How to Contact MathWorks

Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Deep Learning Toolbox™ Reference


The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government’s needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.
<table>
<thead>
<tr>
<th>Date</th>
<th>Printing Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 1992</td>
<td>First printing</td>
<td></td>
</tr>
<tr>
<td>April 1993</td>
<td>Second printing</td>
<td></td>
</tr>
<tr>
<td>January 1997</td>
<td>Third printing</td>
<td></td>
</tr>
<tr>
<td>July 1997</td>
<td>Fourth printing</td>
<td></td>
</tr>
<tr>
<td>January 1998</td>
<td>Fifth printing</td>
<td>Revised for Version 3 (Release 11)</td>
</tr>
<tr>
<td>September 2000</td>
<td>Sixth printing</td>
<td>Revised for Version 4 (Release 12)</td>
</tr>
<tr>
<td>June 2001</td>
<td>Seventh printing</td>
<td>Minor revisions (Release 12.1)</td>
</tr>
<tr>
<td>July 2002</td>
<td>Online only</td>
<td>Minor revisions (Release 13)</td>
</tr>
<tr>
<td>January 2003</td>
<td>Online only</td>
<td>Minor revisions (Release 13SP1)</td>
</tr>
<tr>
<td>June 2004</td>
<td>Online only</td>
<td>Revised for Version 4.0.3 (Release 14)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Online only</td>
<td>Revised for Version 4.0.4 (Release 14SP1)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Eighth printing</td>
<td>Revised for Version 4.0.4</td>
</tr>
<tr>
<td>March 2005</td>
<td>Online only</td>
<td>Revised for Version 4.0.5 (Release 14SP2)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Revised for Version 5.0 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Ninth printing</td>
<td>Minor revisions (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Minor revisions (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for Version 5.1 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for Version 6.0 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for Version 6.0.1 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for Version 6.0.2 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for Version 6.0.3 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for Version 6.0.4 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for Version 7.0 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for Version 7.0.1 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for Version 7.0.2 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for Version 7.0.3 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for Version 8.0 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for Version 8.0.1 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for Version 8.1 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for Version 8.2 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for Version 8.2.1 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for Version 8.3 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for Version 8.4 (Release 2015b)</td>
</tr>
<tr>
<td>March 2016</td>
<td>Online only</td>
<td>Revised for Version 9.0 (Release 2016a)</td>
</tr>
<tr>
<td>September 2016</td>
<td>Online only</td>
<td>Revised for Version 9.1 (Release 2016b)</td>
</tr>
<tr>
<td>March 2017</td>
<td>Online only</td>
<td>Revised for Version 10.0 (Release 2017a)</td>
</tr>
<tr>
<td>September 2017</td>
<td>Online only</td>
<td>Revised for Version 11.0 (Release 2017b)</td>
</tr>
<tr>
<td>March 2018</td>
<td>Online only</td>
<td>Revised for Version 11.1 (Release 2018a)</td>
</tr>
<tr>
<td>September 2018</td>
<td>Online only</td>
<td>Revised for Version 12.0 (Release 2018b)</td>
</tr>
<tr>
<td>March 2019</td>
<td>Online only</td>
<td>Revised for Version 12.1 (Release 2019a)</td>
</tr>
<tr>
<td>September 2019</td>
<td>Online only</td>
<td>Revised for Version 13 (Release 2019b)</td>
</tr>
</tbody>
</table>
Functions — Alphabetical List
Deep Network Designer

Edit and build deep learning networks

Description

The Deep Network Designer app lets you build, visualize, and edit deep learning networks. Using this app, you can:

- Import pretrained networks and edit them for transfer learning.
- Import and edit networks and build new networks.
- Drag and drop to add new layers and create new connections.
- View and edit layer properties.
- Analyze the network to ensure you define the architecture correctly, and detect problems before training.
- Generate MATLAB® code.

After you finish designing a network, you can export it to the workspace, where you can save or train the network.

Open the Deep Network Designer App

- MATLAB Toolstrip: On the Apps tab, under Machine Learning and Deep Learning, click the app icon.
- MATLAB command prompt: Enter `deepNetworkDesigner`.

Examples

Import Pretrained Network

Examine a simple pretrained network in the Deep Network Designer app.

Load a simple pretrained network. If you need to download the network, then the software provides a download link.
net = squeezenet

Open Deep Network Designer.

defaultNetworkDesigner

In the **File** section, click **Import** and choose the network to load from the workspace. Use the plot to explore and visualize the network.

For a list of available networks and how to compare them, see “Pretrained Deep Neural Networks”.

**Edit Pretrained Network for Transfer Learning**

Prepare a network for transfer learning by editing it in the Deep Network Designer app.

Load a pretrained network. If you need to download the network, then the software provides a download link.

net = googlenet

Open Deep Network Designer.

defaultNetworkDesigner

In the **File** section, click **Import** and choose the network to load from the workspace. Use the plot to explore and visualize the network.

Edit the network to specify a new number of classes in your data. Drag a new fully connected layer onto the canvas and set the **OutputSize** property to a new number of classes. Delete the last fully connected layer and connect your new layer instead.

Delete the classification output layer. Then, drag a new classification output layer onto the canvas and connect it instead. The output layer auto settings will learn the number of classes during training.

To check that the network is ready for training, click **Analyze** in the **Analysis** section.

Return to Deep Network Designer. To export the network to the workspace for training, in the **Export** section, click **Export**.
For more information, see “Transfer Learning with Deep Network Designer”.

**Get Help on Layer Properties**

For help understanding and editing layer properties, consult the layer pages.

To find definitions and help on all layer properties, click the layer name in the table “List of Deep Learning Layers”.

In the app, click layers to view and edit properties.

**Generate MATLAB Code**

To recreate the network layers you create in the Deep Network Designer app, generate MATLAB code.

In the app, in the Export section, select Export > Generate Code.

Alternatively, you can recreate your network including any learnable parameters by selecting Export > Generate Code with Pretrained Parameters.

Run the script to recreate the network layers.

For more information, see “Generate MATLAB Code from Deep Network Designer”.

- “Transfer Learning with Deep Network Designer”
- “Build Networks with Deep Network Designer”
- “Generate MATLAB Code from Deep Network Designer”
- “List of Deep Learning Layers”

**See Also**

**Functions**

analyzeNetwork
Topics
“Transfer Learning with Deep Network Designer”
“Build Networks with Deep Network Designer”
“Generate MATLAB Code from Deep Network Designer”
“List of Deep Learning Layers”

Introduced in R2018b
adapt

Adapt neural network to data as it is simulated

Syntax

\[[\text{net},Y,E,Pf,Af,\text{tr}] = \text{adapt}(\text{net},\text{P},\text{T},\text{Pi},\text{Ai})\]

To Get Help

Type `help network/adapt`.

Description

This function calculates network outputs and errors after each presentation of an input.

\[[\text{net},Y,E,Pf,Af,\text{tr}] = \text{adapt}(\text{net},\text{P},\text{T},\text{Pi},\text{Ai})\] takes

<table>
<thead>
<tr>
<th>net</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Network inputs</td>
</tr>
<tr>
<td>T</td>
<td>Network targets (default = zeros)</td>
</tr>
<tr>
<td>Pi</td>
<td>Initial input delay conditions (default = zeros)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay conditions (default = zeros)</td>
</tr>
</tbody>
</table>

and returns the following after applying the adapt function `net.adaptFcn` with the adaption parameters `net.adaptParam`:

<table>
<thead>
<tr>
<th>net</th>
<th>Updated network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Network outputs</td>
</tr>
<tr>
<td>E</td>
<td>Network errors</td>
</tr>
<tr>
<td>Pf</td>
<td>Final input delay conditions</td>
</tr>
</tbody>
</table>
Final layer delay conditions

Training record (epoch and perf)

Note that T is optional and is only needed for networks that require targets. Pi and Pf are also optional and only need to be used for networks that have input or layer delays.

adapt’s signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented,

<table>
<thead>
<tr>
<th>P</th>
<th>Ni-by-TS cell array</th>
<th>Each element P{i,ts} is an Ri-by-Q matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Nt-by-TS cell array</td>
<td>Each element T{i,ts} is a Vi-by-Q matrix.</td>
</tr>
<tr>
<td>Pi</td>
<td>Ni-by-ID cell array</td>
<td>Each element Pi{i,k} is an Ri-by-Q matrix.</td>
</tr>
<tr>
<td>Ai</td>
<td>Nl-by-LD cell array</td>
<td>Each element Ai{i,k} is an Si-by-Q matrix.</td>
</tr>
<tr>
<td>Y</td>
<td>No-by-TS cell array</td>
<td>Each element Y{i,ts} is a Ui-by-Q matrix.</td>
</tr>
<tr>
<td>E</td>
<td>No-by-TS cell array</td>
<td>Each element E{i,ts} is a Ui-by-Q matrix.</td>
</tr>
<tr>
<td>Pf</td>
<td>Ni-by-ID cell array</td>
<td>Each element Pf{i,k} is an Ri-by-Q matrix.</td>
</tr>
<tr>
<td>Af</td>
<td>Nl-by-LD cell array</td>
<td>Each element Af{i,k} is an Si-by-Q matrix.</td>
</tr>
</tbody>
</table>

where

| Ni   | = net.numInputs    |
| Nl   | = net.numLayers    |
| No   | = net.numOutputs   |
| ID   | = net.numInputDelays|
| LD   | = net.numLayerDelays|
TS = Number of time steps
Q = Batch size
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Ui = net.outputs{i}.size

The columns of Pi, Pf, Ai, and Af are ordered from oldest delay condition to most recent:

| Pi{i,k} | = Input i at time ts = k - ID
| Pf{i,k} | = Input i at time ts = TS + k - ID
| Ai{i,k} | = Layer output i at time ts = k - LD
| Af{i,k} | = Layer output i at time ts = TS + k - LD

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

| P | (sum of Ri)-by-Q matrix
| T | (sum of Vi)-by-Q matrix
| Pi | (sum of Ri)-by-(ID*Q) matrix
| Ai | (sum of Si)-by-(LD*Q) matrix
| Y | (sum of Ui)-by-Q matrix
| E | (sum of Ui)-by-Q matrix
| Pf | (sum of Ri)-by-(ID*Q) matrix
| Af | (sum of Si)-by-(LD*Q) matrix

**Examples**

Here two sequences of 12 steps (where T1 is known to depend on P1) are used to define the operation of a filter.
Here *linearlayer* is used to create a layer with an input range of [-1, 1], one neuron, input delays of 0 and 1, and a learning rate of 0.1. The linear layer is then simulated.

```matlab
net = linearlayer([0 1],0.1);
```

Here the network adapts for one pass through the sequence.

The network’s mean squared error is displayed. (Because this is the first call to *adapt*, the default *Pi* is used.)

```matlab
[net,y,e,pf] = adapt(net,p1,t1);
mse(e)
```

Note that the errors are quite large. Here the network adapts to another 12 time steps (using the previous *Pf* as the new initial delay conditions).

```matlab
p2 = {1 -1 -1 1 1 -1 0 0 0 1 -1 -1};
t2 = {2 0 -2 0 2 0 -1 0 0 1 0 -1};
[net,y,e,pf] = adapt(net,p2,t2,pf);
mse(e)
```

Here the network adapts for 100 passes through the entire sequence.

```matlab
p3 = [p1 p2];
t3 = [t1 t2];
for i = 1:100
    [net,y,e] = adapt(net,p3,t3);
end
mse(e)
```

The error after 100 passes through the sequence is very small. The network has adapted to the relationship between the input and target signals.

**Algorithms**

```
adapt` calls the function indicated by `net.adaptFcn`, using the adaption parameter values indicated by `net.adaptParam`.

Given an input sequence with `TS` steps, the network is updated as follows: Each step in the sequence of inputs is presented to the network one at a time. The network’s weight
and bias values are updated after each step, before the next step in the sequence is presented. Thus the network is updated TS times.

See Also
init | revert | sim | train

Introduced before R2006a
adaptwb

Adapt network with weight and bias learning rules

Syntax

\[ [\text{net}, \text{ar}, \text{Ac}] = \text{adapt}(\text{net}, \text{Pd}, \text{T}, \text{Ai}) \]

Description

This function is normally not called directly, but instead called indirectly through the function `adapt` after setting a network's adaptation function (`net.adaptFcn`) to this function.

\[ [\text{net}, \text{ar}, \text{Ac}] = \text{adapt}(\text{net}, \text{Pd}, \text{T}, \text{Ai}) \] takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network</td>
</tr>
<tr>
<td>Pd</td>
<td>Delayed processed input states and inputs</td>
</tr>
<tr>
<td>T</td>
<td>Targets</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network after adaption</td>
</tr>
<tr>
<td>ar</td>
<td>Adaption record</td>
</tr>
<tr>
<td>Ac</td>
<td>Combined initial layer states and layer outputs</td>
</tr>
</tbody>
</table>

Examples

Linear layers use this adaption function. Here a linear layer with input delays of 0 and 1, and a learning rate of 0.5, is created and adapted to produce some target data \( t \) when given some input data \( x \). The response is then plotted, showing the network’s error going down over time.
\begin{verbatim}
x = {-1 0 1 0 1 1 -1 0 -1 1 0 1};
t = {-1 -1 1 1 1 2 0 -1 -1 0 1 1};
net = linearlayer([0 1],0.5);
net.adaptFcn
[net,y,e,xf] = adapt(net,x,t);
plotresponse(t,y)
\end{verbatim}

**See Also**

adapt

*Introduced in R2010b*
adddelay

Add delay to neural network response

Syntax

net = adddelay(net,n)

Description

net = adddelay(net,n) takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of delays</td>
</tr>
</tbody>
</table>

and returns the network with input delay connections increased, and output feedback delays decreased, by the specified number of delays n. The result is a network that behaves identically, except that outputs are produced n timesteps later.

If the number of delays n is not specified, a default of one delay is used.

Examples

Time Delay Network

This example creates, trains, and simulates a time delay network in its original form, on an input time series X and target series T. Then the delay is removed and later added back. The first and third outputs will be identical, while the second result will include a new prediction for the following step.

[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)

net2 = removedelay(net1);
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);
y2 = net2(Xs,Xi);
view(net2)

net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi);
view(net3)
See Also

closeloop | openloop | removedelay

Introduced in R2010b
boxdist

Distance between two position vectors

Syntax

d = boxdist(pos)

Description

boxdist is a layer distance function that is used to find the distances between the layer’s neurons, given their positions.

d = boxdist(pos) takes one argument,

| pos       | N-by-S matrix of neuron positions |

and returns the S-by-S matrix of distances.

boxdist is most commonly used with layers whose topology function is gridtop.

Examples

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

pos = rand(3,10);  
d = boxdist(pos)

Network Use

To change a network so that a layer’s topology uses boxdist, set net.layers{i}.distanceFcn to ‘boxdist’.

In either case, call sim to simulate the network with boxdist.
Algorithms

The box distance $D$ between two position vectors $P_i$ and $P_j$ from a set of $S$ vectors is

$$D_{ij} = \max(\text{abs}(P_i-P_j))$$

See Also

dist | linkdist | mandist | sim

Introduced before R2006a
**bttderiv**

Backpropagation through time derivative function

**Syntax**

\[
\text{bttderiv('dperf_dwb',net,X,T,Xi,Ai,EW)} \\
\text{bttderiv('de_dwb',net,X,T,Xi,Ai,EW)}
\]

**Description**

This function calculates derivatives using the chain rule from a network’s performance back through the network, and in the case of dynamic networks, back through time.

\[
\text{bttderiv('dperf_dwb',net,X,T,Xi,Ai,EW)}
\]
takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network</td>
</tr>
<tr>
<td>X</td>
<td>Inputs, an RxQ matrix (or N\times\text{T} cell array of RixQ matrices)</td>
</tr>
<tr>
<td>T</td>
<td>Targets, an SxQ matrix (or M\times\text{T} cell array of SxQ matrices)</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay states (optional)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states (optional)</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>

and returns the gradient of performance with respect to the network’s weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

\[
\text{bttderiv('de_dwb',net,X,T,Xi,Ai,EW)}
\]
returns the Jacobian of errors with respect to the network’s weights and biases.

**Examples**

Here a feedforward network is trained and both the gradient and Jacobian are calculated.
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
gwb = bttderv('dperf_dwb',net,x,t)
jwb = bttderv('de_dwb',net,x,t)

See Also
defaultderiv | fpderiv | num2deriv | num5deriv | staticderiv

Introduced in R2010b
**cascadeforwardnet**

Cascade-forward neural network

**Syntax**

cascadeforwardnet(hiddenSizes,trainFcn)

**Description**

Cascade-forward networks are similar to feed-forward networks, but include a connection from the input and every previous layer to following layers.

As with feed-forward networks, a two-or more layer cascade-network can learn any finite input-output relationship arbitrarily well given enough hidden neurons.

cascadeforwardnet(hiddenSizes,trainFcn) takes these arguments,

<table>
<thead>
<tr>
<th>hiddenSizes</th>
<th>Row vector of one or more hidden layer sizes (default = 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainFcn</td>
<td>Training function (default = 'trainlm')</td>
</tr>
</tbody>
</table>

and returns a new cascade-forward neural network.

**Examples**

**Create and Train a Cascade Network**

Here a cascade network is created and trained on a simple fitting problem.

```matlab
[x,t] = simplefit_dataset;
net = cascadeforwardnet(10);
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,y,t)
```
perf =

1.9372e-05

See Also
feedforwardnet | network

Topics
“Create, Configure, and Initialize Multilayer Shallow Neural Networks”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b
catelements

Concatenate neural network data elements

**Syntax**

\[ \text{catelements}(x_1, x_2, \ldots, x_n) \]
\[ [x_1; x_2; \ldots; x_n] \]

**Description**

catelements(x_1, x_2, \ldots, x_n) takes any number of neural network data values, and merges them along the element dimension (i.e., the matrix row dimension).

If all arguments are matrices, this operation is the same as \([x_1; x_2; \ldots; x_n]\).

If any argument is a cell array, then all non-cell array arguments are enclosed in cell arrays, and then the matrices in the same positions in each argument are concatenated.

**Examples**

This code concatenates the elements of two matrix data values.

\[
x_1 = \begin{bmatrix} 1 & 2 & 3; & 4 & 7 & 4 \end{bmatrix}
x_2 = \begin{bmatrix} 5 & 8 & 2; & 4 & 7 & 6; & 2 & 9 & 1 \end{bmatrix}
y = \text{catelements}(x_1, x_2)
\]

This code concatenates the elements of two cell array data values.

\[
x_1 = \{[1:3; 4:6]; [7:9; 10:12]; [13:15; 16:18]\}
x_2 = \{[2 1 3]; [4 5 6]; [2 5 4]; [9 7 5]\}
y = \text{catelements}(x_1, x_2)
\]
See Also

catsamples|catsignals|cattimesteps|getelements|nndata|numelements|
setelements

Introduced in R2010b
catsamples

Concatenate neural network data samples

Syntax

catsamples(x1,x2,...,xn)
[x1 x2 ... xn]
catsamples(x1,x2,...,xn,'pad',v)

Description

catsamples(x1,x2,...,xn) takes any number of neural network data values, and merges them along the samples dimension (i.e., the matrix column dimension).

If all arguments are matrices, this operation is the same as [x1 x2 ... xn].

If any argument is a cell array, then all non-cell array arguments are enclosed in cell arrays, and then the matrices in the same positions in each argument are concatenated.

catsamples(x1,x2,...,xn,'pad',v) allows samples with varying numbers of timesteps (columns of cell arrays) to be concatenated by padding the shorter time series with the value v, until they are the same length as the longest series. If v is not specified, then the value NaN is used, which is often used to represent unknown or don't-care inputs or targets.

Examples

This code concatenates the samples of two matrix data values.

x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6]
y = catsamples(x1,x2)

This code concatenates the samples of two cell array data values.
x1 = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
x2 = {[2 1 3; 5 4 1] [4 5 6; 9 4 8]; [2 5 4] [9 7 5]}
y = catsamples(x1,x2)

Here the samples of two cell array data values, with unequal numbers of timesteps, are concatenated.

x1 = {1 2 3 4 5};
x2 = {10 11 12};
y = catsamples(x1,x2,'pad')

See Also
catelements | catsignals | cattimesteps | getsamples | nndata | numsamples | setsamples

Introduced in R2010b
catsignals

Concatenate neural network data signals

Syntax

catsignals(x1,x2,...,xn)
{x1; x2; ...; xn}

Description

catsignals(x1,x2,...,xn) takes any number of neural network data values, and merges them along the element dimension (i.e., the cell row dimension).

If all arguments are matrices, this operation is the same as {x1; x2; ...; xn}.

If any argument is a cell array, then all non-cell array arguments are enclosed in cell arrays, and the cell arrays are concatenated as [x1; x2; ...; xn].

Examples

This code concatenates the signals of two matrix data values.

x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6]
y = catsignals(x1,x2)

This code concatenates the signals of two cell array data values.

x1 = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
x2 = {[2 1 3; 5 4 1] [4 5 6; 9 4 8]; [2 5 4] [9 7 5]}
y = catsignals(x1,x2)
See Also

catelements | catsamples | cattimesteps | getsignals | nndata | numsignals | setsignals

Introduced in R2010b
**cattimesteps**

Concatenate neural network data timesteps

**Syntax**

```matlab
 cattimesteps(x1,x2,...,xn) 
 {x1 x2 ... xn}
```

**Description**

`cattimesteps(x1,x2,...,xn)` takes any number of neural network data values, and merges them along the element dimension (i.e., the cell column dimension).

If all arguments are matrices, this operation is the same as `{x1 x2 ... xn}`.

If any argument is a cell array, all non-cell array arguments are enclosed in cell arrays, and the cell arrays are concatenated as `[x1 x2 ... xn]`.

**Examples**

This code concatenates the elements of two matrix data values.

```matlab
 x1 = [1 2 3; 4 7 4] 
 x2 = [5 8 2; 4 7 6] 
 y = cattimesteps(x1,x2)
```

This code concatenates the elements of two cell array data values.

```matlab
 x1 = {{1:3; 4:6} [7:9; 10:12]; [13:15] [16:18]} 
 x2 = {{2 1 3; 5 4 1} [4 5 6; 9 4 8]; [2 5 4] [9 7 5]} 
 y = cattimesteps(x1,x2)
```
See Also
 catelements | catsamples | catsignals | gettimesteps | nndata | numtimesteps 
 | settimesteps

Introduced in R2010b
cellmat

Create cell array of matrices

Syntax

cellmat(A,B,C,D,v)

Description

cellmat(A,B,C,D,v) takes four integer values and one scalar value v, and returns an A-by-B cell array of C-by-D matrices of value v. If the value v is not specified, zero is used.

Examples

Here two cell arrays of matrices are created.

```matlab
cm1 = cellmat(2,3,5,4)
cm2 = cellmat(3,4,2,2,pi)
```

See Also

nndata

Introduced in R2010b
closeloop

Convert neural network open-loop feedback to closed loop

Syntax

net = closeloop(net)
[net,xi,ai] = closeloop(net,xi,ai)

Description

net = closeloop(net) takes a neural network and closes any open-loop feedback. For each feedback output \( i \) whose property net.outputs\( i \).feedbackMode is 'open', it replaces its associated feedback input and their input weights with layer weight connections coming from the output. The net.outputs\( i \).feedbackMode property is set to 'closed', and the net.outputs\( i \).feedbackInput property is set to an empty matrix. Finally, the value of net.outputs\( i \).feedbackDelays is added to the delays of the feedback layer weights (i.e., to the delays values of the replaced input weights).

[net,xi,ai] = closeloop(net,xi,ai) converts an open-loop network and its current input delay states \( xi \) and layer delay states \( ai \) to closed-loop form.

Examples

Convert NARX Network to Closed-Loop Form

This example shows how to design a NARX network in open-loop form, then convert it to closed-loop form.

\[
[X,T] = \text{simplenarx\_dataset};
net = \text{narxnet}(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = \text{preparets}(net,X,\{\},T);
net = \text{train}(net,Xs,Ts,Xi,Ai);
\text{view}(net)
\]
Yopen = net(Xs,Xi,Ai)
net = closeloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yclosed = net(Xs,Xi,Ai);

**Convert Delay States**

For examples on using closeloop and openloop to implement multistep prediction, see narxnet and narnet.

**See Also**
narnet | narxnet | noloop | openloop

*Introduced in R2010b*
combvec

Create all combinations of vectors

Syntax

combvec(A1,A2,...)

Description

combvec(A1,A2,...) takes any number of inputs, A1, A2, ..., where A1 is a matrix of N1 (column) vectors and A2 is a matrix of N2 (column) vectors. The function returns a matrix of (N1*N2*...) column vectors, where the columns consist of all possibilities of A2 vectors, appended to A1 vectors.

<table>
<thead>
<tr>
<th>A1</th>
<th>Matrix of N1 (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>Matrix of N2 (column) vectors</td>
</tr>
</tbody>
</table>

and returns a matrix of (N1*N2*...) column vectors, where the columns consist of all possibilities of A2 vectors, appended to A1 vectors.

Examples

a1 = [1 2 3; 4 5 6];
a2 = [7 8; 9 10];
a3 = combvec(a1,a2)

a3 =

1  2  3  1  2  3
4  5  6  4  5  6
7  7  7  8  8  8
9  9  9 10 10 10

Introduced before R2006a
**compet**

Competitive transfer function

**Graph and Symbol**

<table>
<thead>
<tr>
<th>Input n</th>
<th>Output a</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ a = \text{compet}(n) \]

Compet Transfer Function

**Syntax**

\[
A = \text{compet}(N,FP)
\]

\[
\text{info} = \text{compet('code')}
\]

**Description**

**compet** is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[
A = \text{compet}(N,FP)
\]

takes \(N\) and optional function parameters,

<table>
<thead>
<tr>
<th>N</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns the S-by-Q matrix \(A\) with a 1 in each column where the same column of \(N\) has its maximum value, and 0 elsewhere.

\[
\text{info} = \text{compet('code')}
\]

returns information according to the code string specified:

\[
\text{compet('name')}
\]

returns the name of this function.
compet('output',FP) returns the [min max] output range.
compet('active',FP) returns the [min max] active input range.
compet('fullderiv') returns 1 or 0, depending on whether dA_dN is S-by-S-by-Q or S-by-Q.
compet('fpnames') returns the names of the function parameters.
compet('fpdefaults') returns the default function parameters.

Examples

Here you define a net input vector N, calculate the output, and plot both with bar graphs.

```matlab
n = [0; 1; -0.5; 0.5];
a = compet(n);
subplot(2,1,1), bar(n), ylabel('n')
subplot(2,1,2), bar(a), ylabel('a')
```

Assign this transfer function to layer i of a network.

```
net.layers{i}.transferFcn = 'compet';
```

See Also

sim | softmax

Introduced before R2006a
**competlayer**

Competitive layer

**Syntax**

competlayer(numClasses,kohonenLR,conscienceLR)

**Description**

Competitive layers learn to classify input vectors into a given number of classes, according to similarity between vectors, with a preference for equal numbers of vectors per class.

`competlayer(numClasses,kohonenLR,conscienceLR)` takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numClasses</code></td>
<td>Number of classes to classify inputs (default = 5)</td>
</tr>
<tr>
<td><code>kohonenLR</code></td>
<td>Learning rate for Kohonen weights (default = 0.01)</td>
</tr>
<tr>
<td><code>conscienceLR</code></td>
<td>Learning rate for conscience bias (default = 0.001)</td>
</tr>
</tbody>
</table>

and returns a competitive layer with `numClasses` neurons.

**Examples**

**Create and Train a Competitive Layer**

Here a competitive layer is trained to classify 150 iris flowers into 6 classes.

```matlab
inputs = iris_dataset;
net = competlayer(6);
net = train(net,inputs);
view(net)
outputs = net(inputs);
classes = vec2ind(outputs);
```
See Also
lvqnet | patternnet | selforgmap

Introduced in R2010b
**con2seq**

Convert concurrent vectors to sequential vectors

**Syntax**

\[
S = \text{con2seq}(b) \\
S = \text{con2seq}(b,\text{TS})
\]

**Description**

Deep Learning Toolbox software arranges concurrent vectors with a matrix, and sequential vectors with a cell array (where the second index is the time step).

\[
\text{con2seq} \quad \text{and} \quad \text{seq2con} \quad \text{allow concurrent vectors to be converted to sequential vectors, and back again.}
\]

\[
S = \text{con2seq}(b) \quad \text{takes one input,}
\]

<table>
<thead>
<tr>
<th>b</th>
<th>R-by-TS matrix</th>
</tr>
</thead>
</table>

and returns one output,

<table>
<thead>
<tr>
<th>S</th>
<th>1-by-TS cell array of R-by-1 vectors</th>
</tr>
</thead>
</table>

\[
S = \text{con2seq}(b,\text{TS}) \quad \text{can also convert multiple batches,}
\]

<table>
<thead>
<tr>
<th>b</th>
<th>N-by-1 cell array of matrices with M*TS columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>S</th>
<th>N-by-TS cell array of matrices with M columns</th>
</tr>
</thead>
</table>
Examples

Here a batch of three values is converted to a sequence.

\[ p1 = [1 4 2] \]
\[ p2 = \text{con2seq}(p1) \]

Here, two batches of vectors are converted to two sequences with two time steps.

\[ p1 = \{[1 3 4 5; 1 1 7 4]; [7 3 4 4; 6 9 4 1]\} \]
\[ p2 = \text{con2seq}(p1,2) \]

See Also
concur | seq2con

Introduced before R2006a
**concur**

Create concurrent bias vectors

**Syntax**

`concur(B,Q)`

**Description**

`concur(B,Q)`

<table>
<thead>
<tr>
<th>B</th>
<th>S-by-1 bias vector (or an Nl-by-1 cell array of vectors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Concurrent size</td>
</tr>
</tbody>
</table>

and returns an S-by-B matrix of copies of B (or an Nl-by-1 cell array of matrices).

**Examples**

Here `concur` creates three copies of a bias vector:

```matlab
b = [1; 3; 2; -1];
concur(b,3)
```

**Network Use**

To calculate a layer's net input, the layer's weighted inputs must be combined with its biases. The following expression calculates the net input for a layer with the `netsum` net input function, two input weights, and a bias:

```matlab
n = netsum(z1,z2,b)
```

The above expression works if Z1, Z2, and B are all S-by-1 vectors. However, if the network is being simulated by `sim` (or `adapt` or `train`) in response to Q concurrent
vectors, then Z1 and Z2 will be S-by-Q matrices. Before B can be combined with Z1 and Z2, you must make Q copies of it.

\[ n = \text{netsum}(z1, z2, \text{concur}(b, q)) \]

**See Also**

con2seq | netprod | netsum | seq2con | sim

**Introduced before R2006a**
**configure**

Configure network inputs and outputs to best match input and target data

**Syntax**

```matlab
net = configure(net,x,t)
net = configure(net,x)
net = configure(net,'inputs',x,i)
net = configure(net,'outputs',t,i)
```

**Description**

Configuration is the process of setting network input and output sizes and ranges, input preprocessing settings and output postprocessing settings, and weight initialization settings to match input and target data.

Configuration must happen before a network’s weights and biases can be initialized. Unconfigured networks are automatically configured and initialized the first time `train` is called. Alternately, a network can be configured manually either by calling this function or by setting a network’s input and output sizes, ranges, processing settings, and initialization settings properties manually.

- `net = configure(net,x,t)` takes input data `x` and target data `t`, and configures the network’s inputs and outputs to match.
- `net = configure(net,x)` configures only inputs.
- `net = configure(net,'inputs',x,i)` configures the inputs specified with the index vector `i`. If `i` is not specified all inputs are configured.
- `net = configure(net,'outputs',t,i)` configures the outputs specified with the index vector `i`. If `i` is not specified all targets are configured.
Examples

Here a feedforward network is created and manually configured for a simple fitting problem (as opposed to allowing train to configure it).

```matlab
[x,t] = simplefit_dataset;
net = feedforwardnet(20); view(net)
net = configure(net,x,t); view(net)
```

See Also
init | isconfigured | train | unconfigure

Introduced in R2010b
confusion

Classification confusion matrix

Syntax

[c, cm, ind, per] = confusion(targets, outputs)

Description

[c, cm, ind, per] = confusion(targets, outputs) takes these values:

<table>
<thead>
<tr>
<th>targets</th>
<th>S-by-Q matrix, where each column vector contains a single 1 value, with all other elements 0. The index of the 1 indicates which of S categories that vector represents.</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputs</td>
<td>S-by-Q matrix, where each column contains values in the range [0, 1]. The index of the largest element in the column indicates which of S categories that vector represents.</td>
</tr>
</tbody>
</table>

and returns these values:

<table>
<thead>
<tr>
<th>c</th>
<th>Confusion value = fraction of samples misclassified</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm</td>
<td>S-by-S confusion matrix, where cm(i, j) is the number of samples whose target is the i th class that was classified as j</td>
</tr>
<tr>
<td>ind</td>
<td>S-by-S cell array, where ind{i, j} contains the indices of samples with the i th target class, but jth output class</td>
</tr>
<tr>
<td>per</td>
<td>S-by-4 matrix, where each row summarizes four percentages associated with the i th class:</td>
</tr>
<tr>
<td></td>
<td>per(i,1) false negative rate  = (false negatives)/(all output negatives)</td>
</tr>
<tr>
<td></td>
<td>per(i,2) false positive rate  = (false positives)/(all output positives)</td>
</tr>
<tr>
<td></td>
<td>per(i,3) true positive rate   = (true positives)/(all output positives)</td>
</tr>
<tr>
<td></td>
<td>per(i,4) true negative rate   = (true negatives)/(all output negatives)</td>
</tr>
</tbody>
</table>
[c, cm, ind, per] = confusion(TARGETS, OUTPUTS) takes these values:

<table>
<thead>
<tr>
<th>targets</th>
<th>1-by-Q vector of 1/0 values representing membership</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputs</td>
<td>S-by-Q matrix, of value in [0, 1] interval, where values greater than or equal to 0.5 indicate class membership</td>
</tr>
</tbody>
</table>

and returns these values:

<table>
<thead>
<tr>
<th>c</th>
<th>Confusion value = fraction of samples misclassified</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm</td>
<td>2-by-2 confusion matrix</td>
</tr>
<tr>
<td>ind</td>
<td>2-by-2 cell array, where ind{i, j} contains the indices of samples whose target is 1 versus 0, and whose output was greater than or equal to 0.5 versus less than 0.5</td>
</tr>
<tr>
<td>per</td>
<td>2-by-4 matrix where each ith row represents the percentage of false negatives, false positives, true positives, and true negatives for the class and out-of-class</td>
</tr>
</tbody>
</table>

**Examples**

```matlab
[x,t] = simpleclass_dataset;
net = patternnet(10);
net = train(net,x,t);
y = net(x);
[c,cm,ind,per] = confusion(t,y)
```

**See Also**

plotconfusion | roc

**Introduced in R2008a**
**convwf**

Convolution weight function

**Syntax**

\[ Z = \text{convwf}(W, P) \]
\[ \text{dim} = \text{convwf}(\text{'size'}, S, R, \text{FP}) \]
\[ \text{dw} = \text{convwf}(\text{'dw'}, W, P, Z, \text{FP}) \]
\[ \text{info} = \text{convwf}(\text{'code'}) \]

**Description**

Weight functions apply weights to an input to get weighted inputs.

\[ Z = \text{convwf}(W, P) \] returns the convolution of a weight matrix \( W \) and an input \( P \).

\[ \text{dim} = \text{convwf}(\text{'size'}, S, R, \text{FP}) \] takes the layer dimension \( S \), input dimension \( R \), and function parameters, and returns the weight size.

\[ \text{dw} = \text{convwf}(\text{'dw'}, W, P, Z, \text{FP}) \] returns the derivative of \( Z \) with respect to \( W \).

\[ \text{info} = \text{convwf}(\text{'code'}) \] returns information about this function. The following codes are defined:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'deriv'</td>
<td>Name of derivative function</td>
</tr>
<tr>
<td>'fullderiv'</td>
<td>Reduced derivative = 2, full derivative = 1, linear derivative = 0</td>
</tr>
<tr>
<td>'pfullderiv'</td>
<td>Input: reduced derivative = 2, full derivative = 1, linear derivative = 0</td>
</tr>
<tr>
<td>'wfullderiv'</td>
<td>Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0</td>
</tr>
<tr>
<td>'name'</td>
<td>Full name</td>
</tr>
<tr>
<td>'fpnames'</td>
<td>Returns names of function parameters</td>
</tr>
</tbody>
</table>
Returns default function parameters

**Examples**

Here you define a random weight matrix $W$ and input vector $P$ and calculate the corresponding weighted input $Z$.

$W = \text{rand}(4,1);$  
$P = \text{rand}(8,1);$  
$Z = \text{convwf}(W,P)$

**Network Use**

To change a network so an input weight uses `convwf`, set  
$\text{net.inputWeights}\{i,j\}.weightFcn$ to 'convwf'. For a layer weight, set  
$\text{net.layerWeights}\{i,j\}.weightFcn$ to 'convwf'.

In either case, call `sim` to simulate the network with `convwf`.

**Introduced in R2006a**
crossentropy

Neural network performance

Syntax

perf = crossentropy(net,targets,outputs,perfWeights)
perf = crossentropy(____,Name,Value)

Description

perf = crossentropy(net,targets,outputs,perfWeights) calculates a network performance given targets and outputs, with optional performance weights and other parameters. The function returns a result that heavily penalizes outputs that are extremely inaccurate (y near 1-t), with very little penalty for fairly correct classifications (y near t). Minimizing cross-entropy leads to good classifiers.

The cross-entropy for each pair of output-target elements is calculated as: ce = -t .* log(y).

The aggregate cross-entropy performance is the mean of the individual values: perf = sum(ce(:))/numel(ce).

Special case (N = 1): If an output consists of only one element, then the outputs and targets are interpreted as binary encoding. That is, there are two classes with targets of 0 and 1, whereas in 1-of-N encoding, there are two or more classes. The binary cross-entropy expression is: ce = -t .* log(y) - (1-t) .* log(1-y).

perf = crossentropy(____,Name,Value) supports customization according to the specified name-value pair arguments.

Examples
Calculate Network Performance

This example shows how to design a classification network with cross-entropy and 0.1 regularization, then calculate performance on the whole dataset.

```matlab
[x,t] = iris_dataset;
net = patternnet(10);
net.performParam.regularization = 0.1;
net = train(net,x,t);
y = net(x);
perf = crossentropy(net,t,y,{1},'regularization',0.1)

perf = 0.0267
```

Set crossentropy as Performance Function

This example shows how to set up the network to use the crossentropy during training.

```matlab
net = feedforwardnet(10);
net.performFcn = 'crossentropy';
net.performParam.regularization = 0.1;
net.performParam.normalization = 'none';
```

Input Arguments

- **net — neural network**
  network object

  Neural network, specified as a network object.

  Example: `net = feedforwardnet(10);`

- **targets — neural network target values**
  matrix or cell array of numeric values

  Neural network target values, specified as a matrix or cell array of numeric values. Network target values define the desired outputs, and can be specified as an N-by-Q matrix of Q N-element vectors, or an M-by-TS cell array where each element is an Ni-by-Q matrix. In each of these cases, N or Ni indicates a vector length, Q the number of samples, M the number of signals for neural networks with multiple outputs, and TS is the

1-49
number of time steps for time series data. targets must have the same dimensions as outputs.

The target matrix columns consist of all zeros and a single 1 in the position of the class being represented by that column vector. When N = 1, the software uses cross entropy for binary encoding, otherwise it uses cross entropy for 1-of-N encoding. NaN values are allowed to indicate unknown or don't-care output values. The performance of NaN target values is ignored.

Data Types: double | cell

outputs — neural network output values
matrix or cell array of numeric values

Neural network output values, specified as a matrix or cell array of numeric values. Network output values can be specified as an N-by-Q matrix of Q N-element vectors, or an M-by-TS cell array where each element is an N_i-by-Q matrix. In each of these cases, N or N_i indicates a vector length, Q the number of samples, M the number of signals for neural networks with multiple outputs and TS is the number of time steps for time series data. outputs must have the same dimensions as targets.

Outputs can include NaN to indicate unknown output values, presumably produced as a result of NaN input values (also representing unknown or don't-care values). The performance of NaN output values is ignored.

General case (N>=2): The columns of the output matrix represent estimates of class membership, and should sum to 1. You can use the softmax transfer function to produce such output values. Use patternnet to create networks that are already set up to use cross-entropy performance with a softmax output layer.

Data Types: double | cell

perfWeights — performance weights
{1} (default) | vector or cell array of numeric values

Performance weights, specified as a vector or cell array of numeric values. Performance weights are an optional argument defining the importance of each performance value, associated with each target value, using values between 0 and 1. Performance values of 0 indicate targets to ignore, values of 1 indicate targets to be treated with normal importance. Values between 0 and 1 allow targets to be treated with relative importance.

Performance weights have many uses. They are helpful for classification problems, to indicate which classifications (or misclassifications) have relatively greater benefits (or
costs). They can be useful in time series problems where obtaining a correct output on
some time steps, such as the last time step, is more important than others. Performance
weights can also be used to encourage a neural network to best fit samples whose targets
are known most accurately, while giving less importance to targets which are known to be
less accurate.

`perfWeights` can have the same dimensions as `targets` and `outputs`. Alternately, each
dimension of the performance weights can either match the dimension of `targets` and
`outputs`, or be 1. For instance, if `targets` is an N-by-Q matrix defining Q samples of N-
element vectors, the performance weights can be N-by-Q indicating a different importance
for each target value, or N-by-1 defining a different importance for each row of the
targets, or 1-by-Q indicating a different importance for each sample, or be the scalar 1
(i.e. 1-by-1) indicating the same importance for all target values.

Similarly, if `outputs` and `targets` are cell arrays of matrices, the `perfWeights` can be
a cell array of the same size, a row cell array (indicating the relative importance of each
time step), a column cell array (indicating the relative importance of each neural network
output), or a cell array of a single matrix or just the matrix (both cases indicating that all
matrices have the same importance values).

For any problem, a `perfWeights` value of `{1}` (the default) or the scalar 1 indicates all
performances have equal importance.

Data Types: `double` | `cell`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the
argument name and `Value` is the corresponding value. `Name` must appear inside quotes.
You can specify several name and value pair arguments in any order as
`Name1,Value1,...,NameN,ValueN`.

Example: `'normalization','standard'` specifies the inputs and targets to be
normalized to the range (-1,+1).

**regularization — proportion of performance attributed to weight/bias values**

0 (default) | numeric value in the range (0,1)

Proportion of performance attributed to weight/bias values, specified as a double between
0 (the default) and 1. A larger value penalizes the network for large weights, and the
more likely the network function will avoid overfitting.
Example: 'regularization', 0
Data Types: single | double

normalization — Normalization mode for outputs, targets, and errors
'default' | 'standard' | 'percent'

Normalization mode for outputs, targets, and errors, specified as 'none', 'standard', or 'percent'. 'none' performs no normalization. 'standard' results in outputs and targets being normalized to (-1, +1), and therefore errors in the range (-2, +2). 'percent' normalizes outputs and targets to (-0.5, 0.5) and errors to (-1, 1).

Example: 'normalization','standard'
Data Types: char

Output Arguments

perf — network performance
double

Network performance, returned as a double in the range (0,1).

See Also

mae | mse | patternnet | sae | softmax | sse

Introduced in R2013b
**defaultderiv**

Default derivative function

**Syntax**

```
defaultderiv('dperf_dwb', net, X, T, Xi, Ai, EW)
defaultderiv('de_dwb', net, X, T, Xi, Ai, EW)
```

**Description**

This function chooses the recommended derivative algorithm for the type of network whose derivatives are being calculated. For static networks, `defaultderiv` calls `staticderiv`; for dynamic networks it calls `bttderiv` to calculate the gradient and `fpderiv` to calculate the Jacobian.

`defaultderiv('dperf_dwb', net, X, T, Xi, Ai, EW)` takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Inputs, an R-by-Q matrix (or N-by-TS cell array of Ri-by-Q matrices)</td>
</tr>
<tr>
<td>T</td>
<td>Targets, an S-by-Q matrix (or M-by-TS cell array of Si-by-Q matrices)</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay states (optional)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states (optional)</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>

and returns the gradient of performance with respect to the network’s weights and biases, where R and S are the number of input and output elements and Q is the number of samples (or N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

`defaultderiv('de_dwb', net, X, T, Xi, Ai, EW)` returns the Jacobian of errors with respect to the network’s weights and biases.
Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```matlab
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
dwb = defaultderiv('dperf_dwb',net,x,t)
```

See Also

bttderv | fpderiv | num2deriv | num5deriv | staticderiv

Introduced in R2010b
**dist**

Euclidean distance weight function

**Syntax**

\[
Z = \text{dist}(W,P,FP) \\
\text{dim} = \text{dist}('\text{size}',S,R,FP) \\
dw = \text{dist}('\text{dw}',W,P,Z,FP) \\
D = \text{dist}(\text{pos}) \\
\text{info} = \text{dist}('\text{code}')
\]

**Description**

Weight functions apply weights to an input to get weighted inputs.

\(Z = \text{dist}(W,P,FP)\) takes these inputs,

<table>
<thead>
<tr>
<th>(W)</th>
<th>S-by-R weight matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>R-by-Q matrix of Q input (column) vectors</td>
</tr>
<tr>
<td>(FP)</td>
<td>Struct of function parameters (optional, ignored)</td>
</tr>
</tbody>
</table>

and returns the S-by-Q matrix of vector distances.

\(\text{dim} = \text{dist}('\text{size}',S,R,FP)\) takes the layer dimension \(S\), input dimension \(R\), and function parameters, and returns the weight size [S-by-R].

\(dw = \text{dist}('\text{dw}',W,P,Z,FP)\) returns the derivative of \(Z\) with respect to \(W\).

**dist** is also a layer distance function which can be used to find the distances between neurons in a layer.

\(D = \text{dist}(\text{pos})\) takes one argument,

| \(\text{pos}\) | N-by-S matrix of neuron positions |

and returns the S-by-S matrix of distances.
info = dist('code') returns information about this function. The following codes are supported:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'deriv'</td>
<td>Name of derivative function</td>
</tr>
<tr>
<td>'fullderiv'</td>
<td>Full derivative = 1, linear derivative = 0</td>
</tr>
<tr>
<td>'pfullderiv'</td>
<td>Input: reduced derivative = 2, full derivative = 1, linear</td>
</tr>
<tr>
<td></td>
<td>derivative = 0</td>
</tr>
<tr>
<td>'name'</td>
<td>Full name</td>
</tr>
<tr>
<td>'fpnames'</td>
<td>Returns names of function parameters</td>
</tr>
<tr>
<td>'fpdefaults'</td>
<td>Returns default function parameters</td>
</tr>
</tbody>
</table>

### Examples

Here you define a random weight matrix W and input vector P and calculate the corresponding weighted input Z.

```matlab
W = rand(4,3);  
P = rand(3,1);  
Z = dist(W,P)
```

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

```matlab
pos = rand(3,10);  
D = dist(pos)
```

### Network Use

You can create a standard network that uses dist by calling newpnn or newgrnn.

To change a network so an input weight uses dist, set net.inputWeights{i,j}.weightFcn to 'dist'. For a layer weight, set net.layerWeights{i,j}.weightFcn to 'dist'.

To change a network so that a layer’s topology uses dist, set net.layers{i}.distanceFcn to 'dist'.

In either case, call sim to simulate the network with dist.
See `newpnn` or `newgrnn` for simulation examples.

**Algorithms**

The Euclidean distance $d$ between two vectors $X$ and $Y$ is

$$d = \sum((x-y)^2)^{0.5}$$

**See Also**

`dotprod` | `linkdist` | `mandist` | `negdist` | `normprod` | `sim`

*Introduced before R2006a*
distdelaynet

Distributed delay network

Syntax

distdelaynet(delays,hiddenSizes,trainFcn)

Description

Distributed delay networks are similar to feedforward networks, except that each input and layer weights has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the time delay neural network (timedelaynet), which only has delays on the input weight.

distdelaynet(delays,hiddenSizes,trainFcn) takes these arguments,

<table>
<thead>
<tr>
<th>delays</th>
<th>Row vector of increasing 0 or positive delays (default = 1:2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hiddenSizes</td>
<td>Row vector of one or more hidden layer sizes (default = 10)</td>
</tr>
<tr>
<td>trainFcn</td>
<td>Training function (default = 'trainlm')</td>
</tr>
</tbody>
</table>

and returns a distributed delay neural network.

Examples

Distributed Delay Network

Here a distributed delay neural network is used to solve a simple time series problem.

```
[X,T] = simpleseries_dataset;
net = distdelaynet({1:2,1:2},10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```
\[ Y = \text{net}(Xs, Xi, Ai); \]
\[ \text{perf} = \text{perform}(\text{net}, Y, Ts) \]

\[ \text{perf} = 0.0323 \]

**See Also**

narnet | narxnet | prepares | removedelay | timedelaynet

**Introduced in R2010b**
divideblock

Divide targets into three sets using blocks of indices

Syntax

[trainInd, valInd, testInd] =
divideblock(Q, trainRatio, valRatio, testRatio)

Description

[trainInd, valInd, testInd] =
divideblock(Q, trainRatio, valRatio, testRatio) separates targets into three
sets: training, validation, and testing. It takes the following inputs:

<table>
<thead>
<tr>
<th>Q</th>
<th>Number of targets to divide up.</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainRatio</td>
<td>Ratio of targets for training. Default = 0.7.</td>
</tr>
<tr>
<td>valRatio</td>
<td>Ratio of targets for validation. Default = 0.15.</td>
</tr>
<tr>
<td>testRatio</td>
<td>Ratio of targets for testing. Default = 0.15.</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>trainInd</th>
<th>Training indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>valInd</td>
<td>Validation indices</td>
</tr>
<tr>
<td>testInd</td>
<td>Test indices</td>
</tr>
</tbody>
</table>

Examples

[trainInd, valInd, testInd] = divideblock(3000, 0.6, 0.2, 0.2);
Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when train is called.

net.divideFcn
net.divideParam
net.divideMode

See Also

divideind | divideint | dividerand | dividetrain

Introduced in R2008a
**divideind**

Divide targets into three sets using specified indices

**Syntax**

\[
\text{[trainInd, valInd, testInd]} = \text{divideind}(Q, \text{trainInd, valInd, testInd})
\]

**Description**

\[
\text{[trainInd, valInd, testInd]} = \text{divideind}(Q, \text{trainInd, valInd, testInd})
\]

separates targets into three sets: training, validation, and testing, according to indices provided. It actually returns the same indices it receives as arguments; its purpose is to allow the indices to be used for training, validation, and testing for a network to be set manually.

It takes the following inputs,

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q)</td>
<td>Number of targets to divide up</td>
</tr>
<tr>
<td>(\text{trainInd})</td>
<td>Training indices</td>
</tr>
<tr>
<td>(\text{valInd})</td>
<td>Validation indices</td>
</tr>
<tr>
<td>(\text{testInd})</td>
<td>Test indices</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{trainInd})</td>
<td>Training indices (unchanged)</td>
</tr>
<tr>
<td>(\text{valInd})</td>
<td>Validation indices (unchanged)</td>
</tr>
<tr>
<td>(\text{testInd})</td>
<td>Test indices (unchanged)</td>
</tr>
</tbody>
</table>

**Examples**

\[
\text{[trainInd, valInd, testInd]} = ...
\]

\[
\text{divideind}(3000, 1:2000, 2001:2500, 2501:3000);
\]
Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

`net.divideFcn`  
`net.divideParam`  
`net.divideMode`

See Also

`divideblock` | `divideint` | `dividerand` | `dividetrain`

Introduced in R2008a
**divideint**

Divide targets into three sets using interleaved indices

**Syntax**

\[
[\text{trainInd}, \text{valInd}, \text{testInd}] = \\
\text{divideint}(Q, \text{trainRatio}, \text{valRatio}, \text{testRatio})
\]

**Description**

\[
[\text{trainInd}, \text{valInd}, \text{testInd}] = \\
\text{divideint}(Q, \text{trainRatio}, \text{valRatio}, \text{testRatio})
\]

separates targets into three sets: training, validation, and testing. It takes the following inputs,

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q)</td>
<td>Number of targets to divide up.</td>
</tr>
<tr>
<td>trainRatio</td>
<td>Ratio of vectors for training. Default = 0.7.</td>
</tr>
<tr>
<td>valRatio</td>
<td>Ratio of vectors for validation. Default = 0.15.</td>
</tr>
<tr>
<td>testRatio</td>
<td>Ratio of vectors for testing. Default = 0.15.</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainInd</td>
<td>Training indices</td>
</tr>
<tr>
<td>valInd</td>
<td>Validation indices</td>
</tr>
<tr>
<td>testInd</td>
<td>Test indices</td>
</tr>
</tbody>
</table>

**Examples**

\[
[\text{trainInd}, \text{valInd}, \text{testInd}] = \text{divideint}(3000, 0.6, 0.2, 0.2);
\]
Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when train is called.

net.divideFcn
net.divideParam
net.divideMode

See Also

divideblock | divideind | dividerand | dividetrain

Introduced in R2008a
dividerand

Divide targets into three sets using random indices

Syntax

[trainInd,valInd,testInd] = dividerand(Q,trainRatio,valRatio,testRatio)

Description

[trainInd,valInd,testInd] = dividerand(Q,trainRatio,valRatio,testRatio) separates targets into three sets: training, validation, and testing. It takes the following inputs,

<table>
<thead>
<tr>
<th>Q</th>
<th>Number of targets to divide up.</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainRatio</td>
<td>Ratio of vectors for training. Default = 0.7.</td>
</tr>
<tr>
<td>valRatio</td>
<td>Ratio of vectors for validation. Default = 0.15.</td>
</tr>
<tr>
<td>testRatio</td>
<td>Ratio of vectors for testing. Default = 0.15.</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>trainInd</th>
<th>Training indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>valInd</td>
<td>Validation indices</td>
</tr>
<tr>
<td>testInd</td>
<td>Test indices</td>
</tr>
</tbody>
</table>

Examples

[trainInd,valInd,testInd] = dividerand(3000,0.6,0.2,0.2);
Network Use

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when train is called.

net.divideFcn
net.divideParam
net.divideMode

See Also
divideblock | divideind | divideint | dividetrain

Introduced in R2008a
**dividetrain**

Assign all targets to training set

**Syntax**

```
[trainInd, valInd, testInd] = dividetrain(Q)
```

**Description**

```
[trainInd, valInd, testInd] = dividetrain(Q)
```

assigns all targets to the training set and no targets to the validation or test sets. It takes the following inputs:

<table>
<thead>
<tr>
<th>Q</th>
<th>Number of targets to divide up.</th>
</tr>
</thead>
</table>

and returns

<table>
<thead>
<tr>
<th>trainInd</th>
<th>Training indices equal to 1:Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>valInd</td>
<td>Empty validation indices, []</td>
</tr>
<tr>
<td>testInd</td>
<td>Empty test indices, []</td>
</tr>
</tbody>
</table>

**Examples**

```
[trainInd, valInd, testInd] = dividetrain(250);
```

**Network Use**

Here are the network properties that define which data division function to use, what its parameters are, and what aspects of targets are divided up, when `train` is called.

- `net.divideFcn`
- `net.divideParam`
- `net.divideMode`
See Also

divideblock | divideind | divideint | dividerand

Introduced in R2010b
**dotprod**

Dot product weight function

**Syntax**

\[
Z = \text{dotprod}(W,P,FP)
\]

\[
dim = \text{dotprod}('size',S,R,FP)
\]

\[
dw = \text{dotprod}('dw',W,P,Z,FP)
\]

\[
info = \text{dotprod}('code')
\]

**Description**

Weight functions apply weights to an input to get weighted inputs.

\[ Z = \text{dotprod}(W,P,FP) \]

takes these inputs,

\[
W \quad \text{S-by-R weight matrix}
\]

\[
P \quad \text{R-by-Q matrix of Q input (column) vectors}
\]

\[
FP \quad \text{Struct of function parameters (optional, ignored)}
\]

and returns the S-by-Q dot product of \( W \) and \( P \).

\[ \text{dim} = \text{dotprod}('size',S,R,FP) \]

takes the layer dimension \( S \), input dimension \( R \), and function parameters, and returns the weight size [S-by-R].

\[ \text{dw} = \text{dotprod}('dw',W,P,Z,FP) \]

returns the derivative of \( Z \) with respect to \( W \).

\[ \text{info} = \text{dotprod}('code') \]

returns information about this function. The following codes are defined:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'deriv'</td>
<td>Name of derivative function</td>
</tr>
<tr>
<td>'pfullderiv'</td>
<td>Input: reduced derivative = 2, full derivative = 1, linear derivative = 0</td>
</tr>
</tbody>
</table>
'wfullderiv'  Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0

'name'  Full name

'fpnames'  Returns names of function parameters

'fpdefaults'  Returns default function parameters

**Examples**

Here you define a random weight matrix $W$ and input vector $P$ and calculate the corresponding weighted input $Z$.

$$W = \text{rand}(4,3);$$
$$P = \text{rand}(3,1);$$
$$Z = \text{dotprod}(W,P)$$

**Network Use**

You can create a standard network that uses `dotprod` by calling `feedforwardnet`.

To change a network so an input weight uses `dotprod`, set `net.inputWeights{i,j}.weightFcn` to `'dotprod'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'dotprod'`.

In either case, call `sim` to simulate the network with `dotprod`.

**See Also**

`dist` | `feedforwardnet` | `negdist` | `normprod` | `sim`

*Introduced before R2006a*
elliotsig

Elliot symmetric sigmoid transfer function

Syntax

\[ A = \text{elliotsig}(N) \]

Description

Transfer functions convert a neural network layer’s net input into its net output.

\[ A = \text{elliotsig}(N) \] takes an \( S \)-by-\( Q \) matrix of \( S \) \( N \)-element net input column vectors and returns an \( S \)-by-\( Q \) matrix \( A \) of output vectors, where each element of \( N \) is squashed from the interval \([-\infty \ \infty]\) to the interval \([-1 \ 1]\) with an “S-shaped” function.

The advantage of this transfer function over other sigmoids is that it is fast to calculate on simple computing hardware as it does not require any exponential or trigonometric functions. Its disadvantage is that it only flattens out for large inputs, so its effect is not as local as other sigmoid functions. This might result in more training iterations, or require more neurons to achieve the same accuracy.

Examples

Calculate a layer output from a single net input vector:

\[ n = [0; 1; -0.5; 0.5]; \]
\[ a = \text{elliotsig}(n); \]

Plot the transfer function:

\[ n = -5:0.01:5; \]
\[ \text{plot}(n, \text{elliotsig}(n)) \]
\[ \text{set(gca,'dataaspectratio',[1 \ 1 \ 1],'xgrid','on','ygrid','on')} \]

For a network you have already defined, change the transfer function for layer \( i \):
net.layers{i}.transferFcn = 'elliot2sig';

See Also
elliot2sig | logsig | tansig

Introduced in R2012b
elliot2sig

Elliot 2 symmetric sigmoid transfer function

Syntax

\[ A = \text{elliot2sig}(N) \]

Description

Transfer functions convert a neural network layer’s net input into its net output. This function is a variation on the original Elliot sigmoid function. It has a steeper slope, closer to \( \text{tansig} \), but is not as smooth at the center.

\[ A = \text{elliot2sig}(N) \] takes an \( S \)-by-\( Q \) matrix of \( S \) \( N \)-element net input column vectors and returns an \( S \)-by-\( Q \) matrix \( A \) of output vectors, where each element of \( N \) is squashed from the interval \([ -\infty \ \infty ]\) to the interval \([-1 \ 1]\) with an “S-shaped” function.

The advantage of this transfer function over other sigmoids is that it is fast to calculate on simple computing hardware as it does not require any exponential or trigonometric functions. Its disadvantage is that it departs from the classic sigmoid shape around zero.

Examples

Calculate a layer output from a single net input vector:

\[
n = [0; 1; -0.5; 0.5];
\]
\[
a = \text{elliot2sig}(n);
\]

Plot the transfer function:

\[
n = -5:0.01:5;
\]
\[
\text{plot}(n, \text{elliot2sig}(n))
\]
\[
\text{set(gca,'dataaspectratio',[1 1 1],'xgrid','on','ygrid','on')}
\]

For a network you have already defined, change the transfer function for layer \( i \):
net.layers{i}.transferFcn = 'elliot2sig';

See Also
elliotsig | logsig | tansig

Introduced in R2012b
elmannet

Elman neural network

Syntax

elmannet(layerdelays,hiddenSizes,trainFcn)

Description

Elman networks are feedforward networks (feedforwardnet) with the addition of layer recurrent connections with tap delays.

With the availability of full dynamic derivative calculations (fpderiv and bttderv), the Elman network is no longer recommended except for historical and research purposes. For more accurate learning try time delay (timedelaynet), layer recurrent (layrecnet), NARX (narxnet), and NAR (narnet) neural networks.

Elman networks with one or more hidden layers can learn any dynamic input-output relationship arbitrarily well, given enough neurons in the hidden layers. However, Elman networks use simplified derivative calculations (using staticderiv, which ignores delayed connections) at the expense of less reliable learning.

elmannet(layerdelays,hiddenSizes,trainFcn) takes these arguments,

<table>
<thead>
<tr>
<th>layerdelays</th>
<th>Row vector of increasing 0 or positive delays (default = 1:2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hiddenSizes</td>
<td>Row vector of one or more hidden layer sizes (default = 10)</td>
</tr>
<tr>
<td>trainFcn</td>
<td>Training function (default = 'trainlm')</td>
</tr>
</tbody>
</table>

and returns an Elman neural network.

Examples

Here an Elman neural network is used to solve a simple time series problem.
[X,T] = simpleseries_dataset;
net = elmannet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y)

See Also
layrecnet | narnet | narxnet | preparets | removedelay | timedelaynet

Introduced in R2010b
errsurf

Error surface of single-input neuron

Syntax

errsurf(P,T,WV,BV,F)

Description

ererrsurf(P,T,WV,BV,F) takes these arguments,

<table>
<thead>
<tr>
<th>P</th>
<th>1-by-Q matrix of input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>1-by-Q matrix of target vectors</td>
</tr>
<tr>
<td>WV</td>
<td>Row vector of values of W</td>
</tr>
<tr>
<td>BV</td>
<td>Row vector of values of B</td>
</tr>
<tr>
<td>F</td>
<td>Transfer function (string)</td>
</tr>
</tbody>
</table>

and returns a matrix of error values over WV and BV.

Examples

```
p = [-6.0 -6.1 -4.1 -4.0 +4.0 +4.1 +6.0 +6.1];
t = [+0.0 +0.0 +.97 +.99 +.01 +.03 +1.0 +1.0];
wv = -1:.1:1; bv = -2.5:.25:2.5;
es = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,es,[60 30])
```

See Also

plotes

1-78
Introduced before R2006a
**extendts**

Extend time series data to given number of timesteps

**Syntax**

`extendts(x,ts,v)`

**Description**

`extendts(x,ts,v)` takes these values,

<table>
<thead>
<tr>
<th>x</th>
<th>Neural network time series data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts</td>
<td>Number of timesteps</td>
</tr>
<tr>
<td>v</td>
<td>Value</td>
</tr>
</tbody>
</table>

and returns the time series data either extended or truncated to match the specified number of timesteps. If the value `v` is specified, then extended series are filled in with that value, otherwise they are extended with random values.

**Examples**

Here, a 20-timestep series is created and then extended to 25 timesteps with the value zero.

```plaintext
x = nndata(5,4,20);
y = extendts(x,25,0)
```

**See Also**

`catsamples` | `nndata` | `preparets`

**Introduced in R2010b**
feedforwardnet

Feedforward neural network

Syntax

feedforwardnet(hiddenSizes,trainFcn)

Description

Feedforward networks consist of a series of layers. The first layer has a connection from the network input. Each subsequent layer has a connection from the previous layer. The final layer produces the network’s output.

Feedforward networks can be used for any kind of input to output mapping. A feedforward network with one hidden layer and enough neurons in the hidden layers, can fit any finite input-output mapping problem.

Specialized versions of the feedforward network include fitting (fitnet) and pattern recognition (patternnet) networks. A variation on the feedforward network is the cascade forward network (cascadeforwardnet) which has additional connections from the input to every layer, and from each layer to all following layers.

feedforwardnet(hiddenSizes,trainFcn) takes these arguments,

| hiddenSizes | Row vector of one or more hidden layer sizes (default = 10) |
| trainFcn    | Training function (default = 'trainlm') |

and returns a feedforward neural network.

Examples

Feedforward Neural Network

This example shows how to use feedforward neural network to solve a simple problem.
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,y,t)

perf =
1.4639e-04

See Also
cascadeforwardnet | fitnet | network | patternnet

Topics
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b
fixunknowns

Process data by marking rows with unknown values

Syntax

\[
[y, ps] = \text{fixunknowns}(X) \\
[y, ps] = \text{fixunknowns}(X, FP) \\
Y = \text{fixunknowns}('apply', X, PS) \\
X = \text{fixunknowns}('reverse', Y, PS) \\
\text{name} = \text{fixunknowns}('name') \\
\text{fp} = \text{fixunknowns}('pdefaults') \\
\text{pd} = \text{fixunknowns}('pdesc') \\
\text{fixunknowns}('pcheck', fp)
\]

Description

fixunknowns processes matrices by replacing each row containing unknown values (represented by NaN) with two rows of information.

The first row contains the original row, with NaN values replaced by the row’s mean. The second row contains 1 and 0 values, indicating which values in the first row were known or unknown, respectively.

\[ [y, ps] = \text{fixunknowns}(X) \] takes these inputs,

\[
<table>
<thead>
<tr>
<th>X</th>
<th>\text{N-by-Q matrix}</th>
</tr>
</thead>
</table>
\]

and returns

\[
| Y | \text{M-by-Q matrix with M - N rows added} |
| PS | \text{Process settings that allow consistent processing of values} |
\]

\[ [y, ps] = \text{fixunknowns}(X, FP) \] takes an empty struct FP of parameters.

\[ Y = \text{fixunknowns}('apply', X, PS) \] returns Y, given X and settings PS.
X = fixunknowns('reverse', Y, PS) returns X, given Y and settings PS.

name = fixunknowns('name') returns the name of this process method.

fp = fixunknowns('pdefaults') returns the default process parameter structure.

pd = fixunknowns('pdesc') returns the process parameter descriptions.

fixunknowns('pcheck', fp) throws an error if any parameter is illegal.

**Examples**

Here is how to format a matrix with a mixture of known and unknown values in its second row:

x1 = [1 2 3 4; 4 NaN 6 5; NaN 2 3 NaN]
[y1, ps] = fixunknowns(x1)

Next, apply the same processing settings to new values:

x2 = [4 5 3 2; NaN 9 NaN 2; 4 9 5 2]
y2 = fixunknowns('apply', x2, ps)

Reverse the processing of y1 to get x1 again.

x1_again = fixunknowns('reverse', y1, ps)

**More About**

**Recode Data with NaNs Using fixunknowns**

If you have input data with unknown values, you can represent them with NaN values. For example, here are five 2-element vectors with unknown values in the first element of two of the vectors:

p1 = [1 NaN 3 2 NaN; 3 1 -1 2 4];

The network will not be able to process the NaN values properly. Use the function fixunknowns to transform each row with NaN values (in this case only the first row) into two rows that encode that same information numerically.
[p2,ps] = fixunknowns(p1);

Here is how the first row of values was recoded as two rows.

\[
\begin{array}{cccc}
1 & 2 & 3 & 2 \\
1 & 0 & 1 & 1 \\
3 & 1 & -1 & 2 \\
\end{array}
\]

The first new row is the original first row, but with the mean value for that row (in this case 2) replacing all NaN values. The elements of the second new row are now either 1, indicating the original element was a known value, or 0 indicating that it was unknown.

The original second row is now the new third row. In this way both known and unknown values are encoded numerically in a way that lets the network be trained and simulated.

Whenever supplying new data to the network, you should transform the inputs in the same way, using the settings ps returned by fixunknowns when it was used to transform the training input data.

\[
p2new = \text{fixunknowns('apply',p1new,ps)};
\]

The function fixunknowns is only recommended for input processing. Unknown targets represented by NaN values can be handled directly by the toolbox learning algorithms. For instance, performance functions used by backpropagation algorithms recognize NaN values as unknown or unimportant values.

See Also
mapminmax | mapstd | processpca

Introduced in R2006a
formwb

Form bias and weights into single vector

Syntax

formwb(net, b, IW, LW)

Description

formwb(net, b, IW, LW) takes a neural network and bias b, input weight IW, and layer weight LW values, and combines the values into a single vector.

Examples

Here a network is created, configured, and its weights and biases formed into a vector.

[x, t] = simplefit_dataset;
net = feedforwardnet(10);
net = configure(net, x, t);
wb = formwb(net, net.b, net.IW, net.LW)

See Also

getwb | separatewb | setwb

Introduced in R2010b
fpderiv

Forward propagation derivative function

Syntax

\[
\text{fpderiv('dperf_dwb',net,X,T,Xi,Ai,EW)} \\
\text{fpderiv('de_dwb',net,X,T,Xi,Ai,EW)}
\]

Description

This function calculates derivatives using the chain rule from inputs to outputs, and in the case of dynamic networks, forward through time.

\text{fpderiv('dperf_dwb',net,X,T,Xi,Ai,EW)} takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Inputs, an R-by-Q matrix (or N-by-TS cell array of Ri-by-Q matrices)</td>
</tr>
<tr>
<td>T</td>
<td>Targets, an S-by-Q matrix (or M-by-TS cell array of Si-by-Q matrices)</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay states (optional)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states (optional)</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>

and returns the gradient of performance with respect to the network’s weights and biases, where R and S are the number of input and output elements and Q is the number of samples (or N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

\text{fpderiv('de_dwb',net,X,T,Xi,Ai,EW)} returns the Jacobian of errors with respect to the network’s weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
gwb = fpderiv('dperf_dwb',net,x,t)
jwb = fpderiv('de_dwb',net,x,t)

See Also
bttderiv|defaultderiv|num2deriv|num5deriv|staticderiv

Introduced in R2010b
fromnndata

Convert data from standard neural network cell array form

Syntax

fromnndata(x,toMatrix,columnSample,cellTime)

Description

fromnndata(x,toMatrix,columnSample,cellTime) takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network</td>
</tr>
<tr>
<td>toMatrix</td>
<td>True if result is to be in matrix form</td>
</tr>
<tr>
<td>columnSample</td>
<td>True if samples are to be represented as columns, false if rows</td>
</tr>
<tr>
<td>cellTime</td>
<td>True if time series are to be represented as a cell array, false if represented with a matrix</td>
</tr>
</tbody>
</table>

and returns the original data reformatted accordingly.

Examples

Here time-series data is converted from a matrix representation to standard cell array representation, and back. The original data consists of a 5-by-6 matrix representing one time-series sample consisting of a 5-element vector over 6 timesteps arranged in a matrix with the samples as columns.

```matlab
x = rands(5,6)
columnSamples = true; % samples are by columns.
cellTime = false; % time-steps in matrix, not cell array.
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
x2 = fromnndata(y,wasMatrix,columnSamples,cellTime)
```
Here data is defined in standard neural network data cell form. Converting this data does not change it. The data consists of three time series samples of 2-element signals over 3 timesteps.

\[
x = \{\text{rands}(2,3); \text{rands}(2,3); \text{rands}(2,3)\}
columnSamples = \text{true};
cellTime = \text{true};
[y,wasMatrix] = \text{tonndata}(x)
x2 = \text{fromnndata}(y,\text{wasMatrix},\text{columnSamples})
\]

**See Also**

tonndata

**Introduced in R2010b**
gadd

Generalized addition

Syntax

gadd(a,b)

Description

gadd(a,b) takes two matrices or cell arrays, and adds them in an element-wise manner.

Examples

Add Matrix and Cell Array Values

This example shows how to add matrix and cell array values.

gadd([1 2 3; 4 5 6],[10;20])

ans = 2×3
   11  12  13
   24  25  26

gadd({1 2; 3 4},{1 3; 5 2})

ans=2×2 cell
     {{2}}    {{5}}
     {{8}}    {{6}}

gadd({1 2 3 4},{10;20;30})

ans=3×4 cell
     {{11}}   {{12}}   {{13}}   {{14}}
See Also

gdivide | gmultiply | gnegate | gsqrt | gsubtract

Introduced in R2010b
**gdivide**

Generalized division

**Syntax**

gdivide(a,b)

**Description**

gdivide(a,b) takes two matrices or cell arrays, and divides them in an element-wise manner.

**Examples**

**Divide Matrix and Cell Array Values**

This example shows how to divide matrix and cell array values.

gdivide([1 2 3; 4 5 6],[10;20])

```matlab
ans =
    0.1000    0.2000    0.3000
    0.2000    0.2500    0.3000
```

gdivide({1 2; 3 4},{1 3; 5 2})

```matlab
ans=2×2 cell
    {1×1 cell}    {1×1 cell}
    [0.6667]    [0.6000]
```

gdivide({1 2 3 4},{10;20;30})

```matlab
ans=3×4 cell
    {1×1 cell}    {1×1 cell}    {1×1 cell}    {1×1 cell}
    [0.1000]    [0.2000]    [0.3000]    [0.4000]
```
See Also
gadd | gmultiply | gnegate | gsqrt | gsubtract

Introduced in R2010b
**gensim**

Generate Simulink block for shallow neural network simulation

**Syntax**

gensim(net,st)

**To Get Help**

Type `help network/gensim`.

**Description**

This function generates a Simulink® block for a shallow neural network. `gensim` does not support deep learning networks such as convolutional or LSTM networks. For more information on code generation for deep learning, see “Deep Learning Code Generation”.

`gensim(net,st)` creates a Simulink system containing a block that simulates neural network `net`.

`gensim(net,st)` takes these inputs:

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>st</td>
<td>Sample time (default = 1)</td>
</tr>
</tbody>
</table>

and creates a Simulink system containing a block that simulates neural network `net` with a sampling time of `st`.

If `net` has no input or layer delays (`net.numInputDelays` and `net.numLayerDelays` are both 0), you can use –1 for `st` to get a network that samples continuously.
Examples

[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
gensim(net)

Introduced before R2006a
genFunction

Generate MATLAB function for simulating shallow neural network

Syntax

```
genFunction(net,pathname)
genFunction(___,'MatrixOnly','yes')
genFunction(___,'ShowLinks','no')
```

Description

This function generates a MATLAB function for simulating a shallow neural network. `genFunction` does not support deep learning networks such as convolutional or LSTM networks. For more information on code generation for deep learning, see “Deep Learning Code Generation”.

`genFunction(net,pathname)` generates a complete stand-alone MATLAB function for simulating a neural network including all settings, weight and bias values, module functions, and calculations in one file. The result is a standalone MATLAB function file. You can also use this function with MATLAB Compiler™ and MATLAB Coder™ tools.

`genFunction(___,'MatrixOnly','yes')` overrides the default cell/matrix notation and instead generates a function that uses only matrix arguments compatible with MATLAB Coder tools. For static networks, the matrix columns are interpreted as independent samples. For dynamic networks, the matrix columns are interpreted as a series of time steps. The default value is 'no'.

`genFunction(___,'ShowLinks','no')` disables the default behavior of displaying links to generated help and source code. The default is 'yes'.

Examples
Create Functions from Static Neural Network

This example shows how to create a MATLAB function and a MEX-function from a static neural network.

First, train a static network and calculate its outputs for the training data.

[x,t] = bodyfat_dataset;
bodyfatNet = feedforwardnet(10);
bodyfatNet = train(bodyfatNet,x,t);
y = bodyfatNet(x);

Next, generate and test a MATLAB function. Then the new function is compiled to a shared/dynamically linked library with mcc.

genFunction(bodyfatNet,'bodyfatFcn');
y2 = bodyfatFcn(x);
accuracy2 = max(abs(y-y2))
mcc -W lib:libBodyfat -T link:lib bodyfatFcn

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB Coder tool codegen to generate a MEX-function, which is also tested.

genFunction(bodyfatNet,'bodyfatFcn','MatrixOnly','yes');
y3 = bodyfatFcn(x);
accuracy3 = max(abs(y-y3))
x1Type = coder.typeof(double(0),[13 Inf]); % Coder type of input 1
codegen bodyfatFcn.m -config:mex -o bodyfatCodeGen -args {x1Type}
y4 = bodyfatCodeGen(x);
accuracy4 = max(abs(y-y4))

Create Functions from Dynamic Neural Network

This example shows how to create a MATLAB function and a MEX-function from a dynamic neural network.

First, train a dynamic network and calculate its outputs for the training data.

[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(maglevNet,x,{},t);
maglevNet = train(maglevNet,X,T,Xi,Ai);
[y,xf,af] = maglevNet(X,Xi,Ai);
Next, generate and test a MATLAB function. Use the function to create a shared/dynamically linked library with `mcc`.

```matlab
genericFunction(maglevNet, 'maglevFcn');
[y2, xf, af] = maglevFcn(X, Xi, Ai);
accuracy2 = max(abs(cell2mat(y)-cell2mat(y2)))
mcc -W lib:libMaglev -T link:lib maglevFcn
```

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB Coder tool `codegen` to generate a MEX-function, which is also tested.

```matlab
genericFunction(maglevNet, 'maglevFcn', 'MatrixOnly', 'yes');
x1 = cell2mat(X(1,:));  % Convert each input to matrix
x2 = cell2mat(X(2,:));
x11 = cell2mat(Xi(1,:));  % Convert each input state to matrix
x12 = cell2mat(Xi(2,:));
[y3, xf1, xf2] = maglevFcn(x1, x2, x11, x12);
accuracy3 = max(abs(cell2mat(y)-y3))
```

```matlab
x1Type = coder.typeof(double(0), [1 Inf]);  % Coder type of input 1
x2Type = coder.typeof(double(0), [1 Inf]);  % Coder type of input 2
x11Type = coder.typeof(double(0), [1 2]);  % Coder type of input 1 states
x12Type = coder.typeof(double(0), [1 2]);  % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen -args {x1Type x2Type x11Type x12Type}
[y4, xf1, xf2] = maglevNetCodeGen(x1, x2, x11, x12);
dynamic_codegen_accuracy = max(abs(cell2mat(y)-y4))
```

### Input Arguments

- **net** — neural network
  
  network object

  Neural network, specified as a network object.

  Example: `net = feedforwardnet(10);`

- **pathname** — location and name of generated function file
  
  (default) | character string

  Location and name of generated function file, specified as a character string. If you do not specify a file name extension of `.m`, it is automatically appended. If you do not specify a path to the file, the default location is the current working folder.

  Example: `'myFcn.m'`

  Data Types: `char`
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You can use genFunction in the Deep Learning Toolbox to generate a standalone MATLAB function for a trained neural network. You can generate C/C++ code from this standalone MATLAB function. To generate Simulink blocks, use the genSim function. See “Deploy Shallow Neural Network Functions”.

See Also
gensim

Topics
“Deploy Shallow Neural Network Functions”

Introduced in R2013b
getelements

Get neural network data elements

Syntax

getelements(x,ind)

Description

getelements(x,ind) returns the elements of neural network data x indicated by the indices ind. The neural network data may be in matrix or cell array form.

If x is a matrix, the result is the ind rows of x.

If x is a cell array, the result is a cell array with as many columns as x, whose elements (1,i) are matrices containing the ind rows of [x(:,i)].

Examples

This code gets elements 1 and 3 from matrix data:

\[
x = \begin{bmatrix}
1 & 2 & 3 \\
4 & 7 & 4
\end{bmatrix}
\]
\[
y = \text{getelements}(x,[1 3])
\]

This code gets elements 1 and 3 from cell array data:

\[
x = \begin{bmatrix}
[1:3; 4:6] \\
[7:9; 10:12] \\
[13:15; 16:18]
\end{bmatrix}
\]
\[
y = \text{getelements}(x,[1 3])
\]

See Also

catelements | getsamples | getsignals | gettimesteps | nndata | numelements | setelements
Introduced in R2010b
getsamples

Get neural network data samples

Syntax

getsamples(x,ind)

Description

getsamples(x,ind) returns the samples of neural network data x indicated by the indices ind. The neural network data may be in matrix or cell array form.

If x is a matrix, the result is the ind columns of x.

If x is a cell array, the result is a cell array the same size as x, whose elements are the ind columns of the matrices in x.

Examples

This code gets samples 1 and 3 from matrix data:

```matlab
x = [1 2 3; 4 7 4]
y = getsamples(x,[1 3])
```

This code gets elements 1 and 3 from cell array data:

```matlab
x = {{[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}}
y = getsamples(x,[1 3])
```

See Also

catsamples|getelements|getsignals|gettimesteps|nndata|numsamples|setsamples
Introduced in R2010b
getsignals

Get neural network data signals

Syntax

getsignals(x,ind)

Description

getsignals(x,ind) returns the signals of neural network data x indicated by the indices ind. The neural network data may be in matrix or cell array form.

If x is a matrix, ind may only be 1, which will return x, or [] which will return an empty matrix.

If x is a cell array, the result is the ind rows of x.

Examples

This code gets signal 2 from cell array data:

\[
x = \{[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]\}
y = getsignals(x,2)
\]

See Also

catsignals | getelements | getsamples | gettimesteps | nndata | numsignals | setsignals

Introduced in R2010b
getsiminit

Get Simulink neural network block initial input and layer delays states

**Syntax**

\[
[xi, ai] = \text{getsiminit}(\text{sysName}, \text{netName}, \text{net})
\]

**Description**

\[
[xi, ai] = \text{getsiminit}(\text{sysName}, \text{netName}, \text{net})
\]

takes these arguments,

<table>
<thead>
<tr>
<th>sysName</th>
<th>The name of the Simulink system containing the neural network block</th>
</tr>
</thead>
<tbody>
<tr>
<td>netName</td>
<td>The name of the Simulink neural network block</td>
</tr>
<tr>
<td>net</td>
<td>The original neural network</td>
</tr>
</tbody>
</table>

and returns,

<table>
<thead>
<tr>
<th>xi</th>
<th>Initial input delay states</th>
</tr>
</thead>
<tbody>
<tr>
<td>ai</td>
<td>Initial layer delay states</td>
</tr>
</tbody>
</table>

**Examples**

Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```matlab
[x,t] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
view(net)
[xs,xi,ai,ts] = preparets(net,x,{},t);
net = train(net,xs,ts,xi,ai);
y = net(xs,xi,ai);
```
Now the network is converted to closed-loop, and the data is reformatted to simulate the network’s closed-loop response.

```matlab
net = closeloop(net);
view(net)
[xs,xi,ai,ts] = preparets(net,x,{},t);
y = net(xs,xi,ai);
```

Here the network is converted to a Simulink system with workspace input and output ports. Its delay states are initialized, inputs X1 defined in the workspace, and it is ready to be simulated in Simulink.

```matlab
[sysName,netName] = gensim(net,'InputMode','Workspace',
    'OutputMode','WorkSpace','SolverMode','Discrete');
setsiminit(sysName,netName,net,xi,ai,1);
x1 = nndata2sim(x,1,1);
```

Finally the initial input and layer delays are obtained from the Simulink model. (They will be identical to the values set with `setsiminit`.)

```matlab
[xi,ai] = getsiminit(sysName,netName,net);
```

### See Also

gensim | nndata2sim | setsiminit | sim2nndata

###Introduced in R2010b
gettimesteps

Get neural network data timesteps

Syntax

gettimesteps(x,ind)

Description

gettimesteps(x,ind) returns the timesteps of neural network data x indicated by the indices ind. The neural network data may be in matrix or cell array form.

If x is a matrix, ind can only be 1, which will return x; or [], which will return an empty matrix.

If x is a cell array the result is the ind columns of x.

Examples

This code gets timestep 2 from cell array data:

```matlab
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
y = gettimesteps(x,2)
```

See Also

cattimesteps | getelements | getsamples | getsignals | nndata | numtimesteps | settimesteps

Introduced in R2010b
getwb

Get network weight and bias values as single vector

Syntax

getwb(net)

Description

getwb(net) returns a neural network’s weight and bias values as a single vector.

Examples

Here a feedforward network is trained to fit some data, then its bias and weight values
are formed into a vector.

[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
w = getwb(net)

See Also
formwb | separatewb | setwb

Introduced in R2010b
gmultiply
Generalized multiplication

Syntax

gmultiply(a,b)

Description

gmultiply(a,b) takes two matrices or cell arrays, and multiplies them in an element-wise manner.

Examples

Multiply Matrix and Cell Array Values

This example shows how to multiply matrix and cell array values.

gmultiply([1 2 3; 4 5 6],[10;20])
ans = 2×3
   10   20   30
   80  100  120

gmultiply({1 2; 3 4},{1 3; 5 2})
ans=2×2 cell
   {{[1]}}    {{[6]}}
   {{[15]}}   {{[8]}}

gmultiply({1 2 3 4},{10;20;30})
ans=3×4 cell
   {{[10]}}    {{[20]}}    {{[30]}}    {{[ 40]}}
See Also

gadd | gdivide | gnegate | gsqrt | gsubtract

Introduced in R2010b
\textbf{gnegate}

Generalized negation

\textbf{Syntax}

gnegate(x)

\textbf{Description}

gnegate(x) takes a matrix or cell array of matrices, and negates their element values.

\textbf{Examples}

\textbf{Negate a Cell Array}

This example shows how to negate a cell array:

\begin{verbatim}
x = {{[1 2; 3 4],[1 -3; -5 2]};
y = gnegate(x);
y{1}, y{2}
\end{verbatim}

\begin{verbatim}
ans = 2x2
   -1   -2
   -3   -4
\end{verbatim}

\begin{verbatim}
ans = 2x2
   -1    3
   5   -2
\end{verbatim}
See Also

gadd | gdivide | gmultiply | gsqrt | gsubtract

Introduced in R2010b
**gpu2nndata**

Reformat neural data back from GPU

**Syntax**

\[ X = \text{gpu2nndata}(Y,Q) \]
\[ X = \text{gpu2nndata}(Y) \]
\[ X = \text{gpu2nndata}(Y,Q,N,TS) \]

**Description**

Training and simulation of neural networks require that matrices be transposed. But they do not require (although they are more efficient with) padding of column length so that each column is memory aligned. This function copies data back from the current GPU and reverses this transform. It can be used on data formatted with `nndata2gpu` or on the results of network simulation.

\[ X = \text{gpu2nndata}(Y,Q) \]

copies the \( QQ \)-by-\( N \) gpuArray \( Y \) into RAM, takes the first \( Q \) rows and transposes the result to get an \( N \)-by-\( Q \) matrix representing \( Q \) \( N \)-element vectors.

\[ X = \text{gpu2nndata}(Y) \]

calculates \( Q \) as the index of the last row in \( Y \) that is not all NaN values (those rows were added to pad \( Y \) for efficient GPU computation by `nndata2gpu`). \( Y \) is then transformed as before.

\[ X = \text{gpu2nndata}(Y,Q,N,TS) \]

takes a \( QQ \)-by-\((N*TS) \) gpuArray where \( N \) is a vector of signal sizes, \( Q \) is the number of samples (less than or equal to the number of rows after alignment padding \( QQ \)), and \( TS \) is the number of time steps.

The gpuArray \( Y \) is copied back into RAM, the first \( Q \) rows are taken, and then it is partitioned and transposed into an \( M \)-by-\( TS \) cell array, where \( M \) is the number of elements in \( N \). Each \( Y\{\text{i,ts}\} \) is an \( N(i) \)-by-\( Q \) matrix.

**Examples**

Copy a matrix to the GPU and back:
x = rand(5,6)
[y,q] = nndata2gpu(x)
x2 = gpu2nndata(y,q)

Copy from the GPU a neural network cell array data representing four time series, each consisting of five time steps of 2-element and 3-element signals.

x = nddata([2;3],4,5)
[y,q,n,ts] = nndata2gpu(x)
x2 = gpu2nndata(y,q,n,ts)

See Also
nndata2gpu

Introduced in R2012b
gridtop

Grid layer topology function

Syntax

gridtop(dimensions)

Description

pos = gridtop calculates neuron positions for layers whose neurons are arranged in an N-dimensional grid.

gridtop(dimensions) takes one argument:

| dimensions | Row vector of dimension sizes |

and returns an N-by-S matrix of N coordinate vectors where N is the number of dimensions and S is the product of dimensions.

Examples

Display Layer with Grid Pattern

This example shows how to display a two-dimensional layer with 40 neurons arranged in an 8-by-5 grid pattern.

pos = gridtop([8 5]);
plotsom(pos)
See Also
hextop | randtop | tritop

Introduced before R2006a


gsqrt
Generalized square root

Syntax

gsqrt(x)

Description

gsqrt(x) takes a matrix or cell array of matrices, and generates the element-wise square root of the matrices.

Examples

Compute Element-Wise Square Root

This example shows how to get the element-wise square root of a cell array:

gsqrt({1 2; 3 4})

ans=2x2 cell
    {[     1]}    {[1.4142]}
    {[1.7321]}    {[     2]}

See Also

gadd | gdivide | gmultiply | gnegate | gsubtract

Introduced in R2010b
gsubtract

Generalized subtraction

Syntax

gsubtract(a,b)

Description

gsubtract(a,b) takes two matrices or cell arrays, and subtracts them in an element-wise manner.

Examples

Subtract Matrix and Cell Array Values

This example shows how to subtract matrix and cell array values.

gsubtract([1 2 3; 4 5 6],[10;20])

ans = 2×3
     -9     -8     -7
    -16    -15    -14


gsubtract({1 2; 3 4},{1 3; 5 2})

ans=2×2 cell
    {[0]}    {[1]}
    {[2]}    {[2]}


gsubtract({1 2 3 4},{10;20;30})

ans=3×4 cell
    {[  -9]}    {[  -8]}    {[  -7]}    {[  -6]}
See Also

gadd | gdivide | gmultiply | gnegate | gsqrt

Introduced in R2010b
**hardlim**

Hard-limit transfer function

**Graph and Symbol**

![Graph and Symbol](image)

\[ a = hardlim(n) \]

Hard-Limit Transfer Function

**Syntax**

\[ A = hardlim(N, FP) \]

**Description**

`hardlim` is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = hardlim(N, FP) \] takes \( N \) and optional function parameters,

<table>
<thead>
<tr>
<th>N</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \( A \), the S-by-Q Boolean matrix with 1s where \( N \geq 0 \).

`info = hardlim('code')` returns information according to the code string specified:

`hardlim('name')` returns the name of this function.
hardlim(‘output’, FP) returns the [min max] output range.

hardlim(‘active’, FP) returns the [min max] active input range.

hardlim(‘fullderiv’) returns 1 or 0, depending on whether \( \frac{dA}{dN} \) is \( S \)-by-\( S \)-by-\( Q \) or \( S \)-by-\( Q \).

hardlim(‘fpnames’) returns the names of the function parameters.

hardlim(‘fpdefaults’) returns the default function parameters.

**Examples**

Here is how to create a plot of the hardlim transfer function.

```matlab
n = -5:0.1:5;
a = hardlim(n);
plot(n,a)
```

Assign this transfer function to layer \( i \) of a network.

```matlab
net.layers{i}.transferFcn = 'hardlim';
```

**Algorithms**

hardlim(\( n \)) = 1 if \( n \geq 0 \)

0 otherwise

**See Also**

hardlims | sim

Introduced before R2006a
hardlims

Symmetric hard-limit transfer function

Graph and Symbol

![Graph and Symbol](image)

Symmetric Hard-Limit Transfer Function

Syntax

A = hardlims(N,FP)

Description

hardlims is a neural transfer function. Transfer functions calculate a layer's output from its net input.

A = hardlims(N,FP) takes N and optional function parameters,

<table>
<thead>
<tr>
<th>N</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns A, the S-by-Q +1/-1 matrix with +1s where N ≥ 0.

info = hardlims('code') returns information according to the code string specified:

hardlims('name') returns the name of this function.
hardlims('output',FP) returns the [min max] output range.

hardlims('active',FP) returns the [min max] active input range.

hardlims('fullderiv') returns 1 or 0, depending on whether $dA_dN$ is $S$-by-$S$-$by$-$Q$ or $S$-by-$Q$.

hardlims('fpnames') returns the names of the function parameters.

hardlims('fpdefaults') returns the default function parameters.

**Examples**

Here is how to create a plot of the hardlims transfer function.

```matlab
n = -5:0.1:5;
a = hardlims(n);
plot(n,a)
```

Assign this transfer function to layer i of a network.

```matlab
net.layers{i}.transferFcn = 'hardlims';
```

**Algorithms**

hardlims(n) = 1 if $n \geq 0$, -1 otherwise.

**See Also**

hardlim | sim

*Introduced before R2006a*
**hextop**

Hexagonal layer topology function

**Syntax**

`hextop(dimensions)`

**Description**

`hextop` calculates the neuron positions for layers whose neurons are arranged in an N-dimensional hexagonal pattern.

`hextop(dimensions)` takes one argument:

<table>
<thead>
<tr>
<th>dimensions</th>
<th>Row vector of dimension sizes</th>
</tr>
</thead>
</table>

and returns an N-by-S matrix of N coordinate vectors where N is the number of dimensions and S is the product of `dimensions`.

**Examples**

**Display Layer with Hexagonal Pattern**

This example shows how to display a two-dimensional layer with 40 neurons arranged in an 8-by-5 hexagonal pattern.

```matlab
pos = hextop([8 5]);
plotsom(pos)
```
See Also
gridtop | randtop | tritop

Introduced before R2006a
**ind2vec**

Convert indices to vectors

**Syntax**

`ind2vec(ind)`
`ind2vec(ind,N)`

**Description**

`ind2vec` and `vec2ind` allow indices to be represented either by themselves, or as vectors containing a 1 in the row of the index they represent.

`ind2vec(ind)` takes one argument,

<table>
<thead>
<tr>
<th>ind</th>
<th>Row vector of indices</th>
</tr>
</thead>
</table>

and returns a sparse matrix of vectors, with one 1 in each column, as indicated by `ind`.

`ind2vec(ind,N)` returns an N-by-M matrix, where N can be equal to or greater than the maximum index.

**Examples**

Here four indices are defined and converted to vector representation.

```matlab
ind = [1 3 2 3];
vec = ind2vec(ind)
```

```matlab
disp(vec)
(1,1) 1
(3,2) 1
(2,3) 1
(3,4) 1
```
Here a vector with all zeros in the last row is converted to indices and back, while preserving the number of rows.

```matlab
vec = [0 0 1 0; 1 0 0 0; 0 1 0 0]'
vec =
    0     1     0
    0     0     1
    1     0     0
    0     0     0

[ind,n] = vec2ind(vec)
ind =
    3     1     2
n =
    4
vec2 = full(ind2vec(ind,n))
vec2 =
    0     1     0
    0     0     1
    1     0     0
    0     0     0
```

**See Also**

`ind2sub` | `sub2ind` | `vec2ind`

*Introduced before R2006a*
init

Initialize neural network

Syntax

net = init(net)

To Get Help

Type help network/init.

Description

net = init(net) returns neural network net with weight and bias values updated according to the network initialization function, indicated by net.initFcn, and the parameter values, indicated by net.initParam.

Examples

Here a perceptron is created, and then configured so that its input, output, weight, and bias dimensions match the input and target data.

\[
x = \begin{bmatrix} 0 & 1 & 0 & 1; 0 & 0 & 1 & 1\end{bmatrix};
\]
\[
t = \begin{bmatrix} 0 & 0 & 0 & 1\end{bmatrix};
\]
net = perceptron;
net = configure(net,x,t);
net.iw{1,1}
net.b{1}

Training the perceptron alters its weight and bias values.

net = train(net,x,t);
net.iw{1,1}
net.b{1}
init reinitializes those weight and bias values.

```matlab
net = init(net);
net.iw{1,1}
net.b{1}
```

The weights and biases are zeros again, which are the initial values used by perceptron networks.

**Algorithms**

init calls `net.initFcn` to initialize the weight and bias values according to the parameter values `net.initParam`.

Typically, `net.initFcn` is set to `'initlay'`, which initializes each layer’s weights and biases according to its `net.layers{i}.initFcn`.

Backpropagation networks have `net.layers{i}.initFcn` set to `'initnw'`, which calculates the weight and bias values for layer \( i \) using the Nguyen-Widrow initialization method.

Other networks have `net.layers{i}.initFcn` set to `'initwb'`, which initializes each weight and bias with its own initialization function. The most common weight and bias initialization function is `rands`, which generates random values between \(-1\) and \(1\).

**See Also**

`adapt` | `initlay` | `initnw` | `initwb` | `rands` | `revert` | `sim` | `train`

*Introduced before R2006a*
initcon

Conscience bias initialization function

Syntax

initcon (S,PR)

Description

initcon is a bias initialization function that initializes biases for learning with the learncon learning function.

initcon (S,PR) takes two arguments,

<table>
<thead>
<tr>
<th></th>
<th>Number of rows (neurons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>PR</td>
<td>R-by-2 matrix of R = [Pmin Pmax] (default = [1 1])</td>
</tr>
</tbody>
</table>

and returns an S-by-1 bias vector.

Note that for biases, R is always 1. initcon could also be used to initialize weights, but it is not recommended for that purpose.

Examples

Here initial bias values are calculated for a five-neuron layer.

b = initcon(5)

Network Use

You can create a standard network that uses initcon to initialize weights by calling competlayer.
To prepare the bias of layer \( i \) of a custom network to initialize with \( \text{initcon} \),

1. Set \( \text{net.initFcn} \) to 'initlay'. (\( \text{net.initParam} \) automatically becomes \( \text{initlay}'s \) default parameters.)
2. Set \( \text{net.layers}\{i\}.\text{initFcn} \) to 'initwb'.
3. Set \( \text{net.biases}\{i\}.\text{initFcn} \) to 'initcon'.

To initialize the network, call \( \text{init} \).

**Algorithms**

\( \text{learncon} \) updates biases so that each bias value \( b(i) \) is a function of the average output \( c(i) \) of the neuron \( i \) associated with the bias.

\( \text{initcon} \) gets initial bias values by assuming that each neuron has responded to equal numbers of vectors in the past.

**See Also**

\( \text{competlayer} | \text{init} | \text{initlay} | \text{initwb} | \text{learncon} \)

**Introduced before R2006a**
initlay

Layer-by-layer network initialization function

Syntax

net = initlay(net)
info = initlay('code')

Description

initlay is a network initialization function that initializes each layer \( i \) according to its own initialization function \( \text{net.layers}\{i\}.\text{initFcn} \).

\[ \text{net} = \text{initlay}(\text{net}) \]
and returns the network with each layer updated.

\[ \text{info} = \text{initlay}(\text{'code'}) \]
returns useful information for each supported code character vector:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pnames'</td>
<td>Names of initialization parameters</td>
</tr>
<tr>
<td>'pdefaults'</td>
<td>Default initialization parameters</td>
</tr>
</tbody>
</table>

initlay does not have any initialization parameters.

Network Use

You can create a standard network that uses initlay by calling feedforwardnet, cascadeforwardnet, and many other network functions.

To prepare a custom network to be initialized with initlay,
1 Set net.initFcn to 'initlay'. This sets net.initParam to the empty matrix [], because initlay has no initialization parameters.

2 Set each net.layers{i}.initFcn to a layer initialization function. (Examples of such functions are initwb and initnw.)

To initialize the network, call init.

Algorithms

The weights and biases of each layer i are initialized according to net.layers{i}.initFcn.

See Also
cascadeforwardnet | feedforwardnet | init | initnw | initwb

Introduced before R2006a
initlvq

LVQ weight initialization function

Syntax

initlvq('configure',x)
initlvq('configure',net,'IW',i,j,settings)
initlvq('configure',net,'LW',i,j,settings)
initlvq('configure',net,'b',i,)

Description

initlvq('configure',x) takes input data x and returns initialization settings for an LVQ weights associated with that input.

initlvq('configure',net,'IW',i,j,settings) takes a network, and indices indicating an input weight to layer i from input j, and that weights settings, and returns new weight values.

initlvq('configure',net,'LW',i,j,settings) takes a network, and indices indicating a layer weight to layer i from layer j, and that weights settings, and returns new weight values.

initlvq('configure',net,'b',i,) takes a network, and an index indicating a bias for layer i, and returns new bias values.

See Also

init|lvqnet

Introduced in R2010b
**initnw**

Nguyen-Widrow layer initialization function

**Syntax**

```
net = initnw(net,i)
```

**Description**

`initnw` is a layer initialization function that initializes a layer’s weights and biases according to the Nguyen-Widrow initialization algorithm. This algorithm chooses values in order to distribute the active region of each neuron in the layer approximately evenly across the layer’s input space. The values contain a degree of randomness, so they are not the same each time this function is called.

`initnw` requires that the layer it initializes have a transfer function with a finite active input range. This includes transfer functions such as `tansig` and `satlin`, but not `purelin`, whose active input range is the infinite interval $[-\infty, \infty]$. Transfer functions, such as `tansig`, will return their active input range as follows:

```
activeInputRange = tansig('active')
activeInputRange =
  -2  2
```

`net = initnw(net,i)` takes two arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Index of a layer</td>
</tr>
</tbody>
</table>

and returns the network with layer i’s weights and biases updated.

There is a random element to Nguyen-Widrow initialization. Unless the default random generator is set to the same seed before each call to `initnw`, it will generate different weight and bias values each time.
Network Use

You can create a standard network that uses initnw by calling feedforwardnet or cascadeforwardnet.

To prepare a custom network to be initialized with initnw,

1. Set net.initFcn to 'initlay'. This sets net.initParam to the empty matrix [], because initlay has no initialization parameters.
2. Set net.layers{i}.initFcn to 'initnw'.

To initialize the network, call init.

Algorithms

The Nguyen-Widrow method generates initial weight and bias values for a layer so that the active regions of the layer’s neurons are distributed approximately evenly over the input space.

Advantages over purely random weights and biases are

- Few neurons are wasted (because all the neurons are in the input space).
- Training works faster (because each area of the input space has neurons). The Nguyen-Widrow method can only be applied to layers
  - With a bias
  - With weights whose weightFcn is dotprod
  - With netInputFcn set to netsum
  - With transferFcn whose active region is finite

If these conditions are not met, then initnw uses rands to initialize the layer’s weights and biases.

See Also

cascadeforwardnet | feedforwardnet | init | initlay | initwb
Introduced before R2006a
**initwb**

By weight and bias layer initialization function

**Syntax**

initwb(net,i)

**Description**

initwb is a layer initialization function that initializes a layer’s weights and biases according to their own initialization functions.

initwb(net,i) takes two arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Index of a layer</td>
</tr>
</tbody>
</table>

and returns the network with layer i’s weights and biases updated.

**Network Use**

You can create a standard network that uses initwb by calling perceptron or linearlayer.

To prepare a custom network to be initialized with initwb,

1. Set net.initFcn to 'initlay'. This sets net.initParam to the empty matrix [], because initlay has no initialization parameters.
2. Set net.layers{i}.initFcn to 'initwb'.
3. Set each net.inputWeights{i,j}.initFcn to a weight initialization function. Set each net.layerWeights{i,j}.initFcn to a weight initialization function. Set each net.biases{i}.initFcn to a bias initialization function. Examples of initialization functions are rands (for weights and biases) and midpoint (for weights only).
To initialize the network, call \texttt{init}.

\section*{Algorithms}

Each weight (bias) in layer \(i\) is set to new values calculated according to its weight (bias) initialization function.

\section*{See Also}
\texttt{init|initlay|initnw|linearlayer|perceptron}

\textit{Introduced before R2006a}
initzero

Zero weight and bias initialization function

Syntax

\[
W = \text{initzero}(S, PR) \\
b = \text{initzero}(S, [1 1])
\]

Description

\(W = \text{initzero}(S, PR)\) takes two arguments,

<table>
<thead>
<tr>
<th>S</th>
<th>Number of rows (neurons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>R-by-2 matrix of input value ranges = [Pmin  Pmax]</td>
</tr>
</tbody>
</table>

and returns an \(S\)-by-\(R\) weight matrix of zeros.

\(b = \text{initzero}(S, [1 1])\) returns an \(S\)-by-1 bias vector of zeros.

Examples

Here initial weights and biases are calculated for a layer with two inputs ranging over \([0 1]\) and \([-2 2]\) and four neurons.

\[
W = \text{initzero}(5, [0 1; -2 2]) \\
b = \text{initzero}(5, [1 1])
\]

See Also

init | initlay | initwb

Introduced before R2006a
isconfigured

Indicate if network inputs and outputs are configured

Syntax

[flag, inputflags, outputflags] = isconfigured(net)

Description

[flag, inputflags, outputflags] = isconfigured(net) takes a neural network and returns three values,

<table>
<thead>
<tr>
<th>flag</th>
<th>True if all network inputs and outputs are configured (have non-zero sizes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputflags</td>
<td>Vector of true/false values for each configured/unconfigured input</td>
</tr>
<tr>
<td>outputflags</td>
<td>Vector of true/false values for each configured/unconfigured output</td>
</tr>
</tbody>
</table>

Examples

Here are the flags returned for a new network before and after being configured:

```plaintext
net = feedforwardnet;
[flag, inputFlags, outputFlags] = isconfigured(net)
[x, t] = simplefit_dataset;
net = configure(net, x, t);
[flag, inputFlags, outputFlags] = isconfigured(net)
```

See Also

configure | unconfigure
Introduced in R2010b
layrecnet

Layer recurrent neural network

Syntax

layrecnet(layerDelays,hiddenSizes,trainFcn)

Description

Layer recurrent neural networks are similar to feedforward networks, except that each layer has a recurrent connection with a tap delay associated with it. This allows the network to have an infinite dynamic response to time series input data. This network is similar to the time delay (timedelaynet) and distributed delay (distdelaynet) neural networks, which have finite input responses.

layrecnet(layerDelays,hiddenSizes,trainFcn) takes these arguments,

<table>
<thead>
<tr>
<th>layerDelays</th>
<th>Row vector of increasing 0 or positive delays (default = 1:2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hiddenSizes</td>
<td>Row vector of one or more hidden layer sizes (default = 10)</td>
</tr>
<tr>
<td>trainFcn</td>
<td>Training function (default = 'trainlm')</td>
</tr>
</tbody>
</table>

and returns a layer recurrent neural network.

Examples

Recurrent Neural Network

Use a layer recurrent neural network to solve a simple time series problem.

[X,T] = simpleseries_dataset;
net = layrecnet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)

perf =
6.1239e-11

See Also
distdelaynet | narnet | narxnet | preparets | removedelay | timedelaynet

Introduced in R2010b
**learncon**

Conscience bias learning function

**Syntax**

```
[dB,LS] = learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learncon('code')
```

**Description**

`learncon` is the conscience bias learning function used to increase the net input to neurons that have the lowest average output until each neuron responds approximately an equal percentage of the time.

```
[dB,LS] = learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,
```

<table>
<thead>
<tr>
<th>B</th>
<th>S-by-1 bias vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>1-by-Q ones vector</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns
Learning occurs according to `learncon`'s learning parameter, shown here with its default value.

```
<table>
<thead>
<tr>
<th>DB</th>
<th>S-by-1 weight (or bias) change matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>New learning state</td>
</tr>
</tbody>
</table>
```

```
LP.lr - 0.001 Learning rate
```

```
info = learncon('code') returns useful information for each supported code character vector:

<table>
<thead>
<tr>
<th>'pnames'</th>
<th>Names of learning parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>
```

Deep Learning Toolbox 2.0 compatibility: The LP.lr described above equals 1 minus the bias time constant used by `trainc` in the Deep Learning Toolbox 2.0 software.

**Examples**

Here you define a random output A and bias vector W for a layer with three neurons. You also define the learning rate LR.

```matlab
a = rand(3,1);
b = rand(3,1);
lp.lr = 0.5;
```

Because `learncon` only needs these values to calculate a bias change (see “Algorithm” below), use them to do so.

```
dW = learncon(b,[],[],[],a,[],[],[],[],[],lp,[])
```

**Network Use**

To prepare the bias of layer i of a custom network to learn with `learncon`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2 Set net.adaptFcn to 'trains'. (net.adaptParam automatically becomes
trains’s default parameters.)
3 Set net.inputWeights{i}.learnFcn to 'learncon'
4 Set each net.layerWeights{i,j}.learnFcn to 'learncon'. (Each weight
learning parameter property is automatically set to learncon’s default parameters.)

To train the network (or enable it to adapt),
1 Set net.trainParam (or net.adaptParam) properties as desired.
2 Call train (or adapt).

**Algorithms**

learncon calculates the bias change db for a given neuron by first updating each
neuron’s conscience, i.e., the running average of its output:

\[ c = (1-lr) \cdot c + lr \cdot a \]

The conscience is then used to compute a bias for the neuron that is greatest for smaller
conscience values.

\[ b = \exp(1-\log(c)) - b \]

(learncon recovers C from the bias values each time it is called.)

**See Also**

adapt | learnk | learnos | train

**Introduced before R2006a**
learngd

Gradient descent weight and bias learning function

Syntax

[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngd('code')

Description

learngd is the gradient descent weight and bias learning function.

[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>S-by-R weight matrix (or S-by-1 bias vector)</td>
</tr>
<tr>
<td>P</td>
<td>R-by-Q input vectors (or ones(1,Q))</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q output gradient with respect to performance x Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be []</td>
</tr>
</tbody>
</table>

and returns
Learning occurs according to `learnmd`'s learning parameter, shown here with its default value.

<table>
<thead>
<tr>
<th>LP.lr - 0.01</th>
<th>Learning rate</th>
</tr>
</thead>
</table>

info = `learnmgd('code')` returns useful information for each supported `code` character vector:

- 'pnames' Names of learning parameters
- 'pdefaults' Default learning parameters
- 'needg' Returns 1 if this function uses gW or gA

**Examples**

Here you define a random gradient $gW$ for a weight going to a layer with three neurons from an input with two elements. Also define a learning rate of 0.5.

```matlab
gW = rand(3,2);
lp.lr = 0.5;
```

Because `learnmd` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnmd([],[],[],[],[],[],gW,[],[],lp,[])
```

**Algorithms**

`learnmd` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$ and error $E$, and the weight (or bias) learning rate $LR$, according to the gradient descent $dw = lr*gW$.

**See Also**

adapt | learnmgm | train
Introduced before R2006a
**learngdm**

Gradient descent with momentum weight and bias learning function

**Syntax**

```
[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngdm('code')
```

**Description**

`learngdm` is the gradient descent with momentum weight and bias learning function.

```
[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,
```

<table>
<thead>
<tr>
<th><strong>W</strong></th>
<th>S-by-R weight matrix (or S-by-1 bias vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P</strong></td>
<td>R-by-Q input vectors (or ones(1,Q))</td>
</tr>
<tr>
<td><strong>Z</strong></td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td><strong>gW</strong></td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td><strong>gA</strong></td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td><strong>LP</strong></td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td><strong>LS</strong></td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns

| **dW** | S-by-R weight (or bias) change matrix |

1-152
Learning occurs according to learnngdm's learning parameters, shown here with their default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP.lr</td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td>LP.mc</td>
<td>0.9</td>
<td>Momentum constant</td>
</tr>
</tbody>
</table>

info = learnngdm('code') returns useful information for each code character vector:

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pnames'</td>
<td>Names of learning parameters</td>
</tr>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

**Examples**

Here you define a random gradient G for a weight going to a layer with three neurons from an input with two elements. Also define a learning rate of 0.5 and momentum constant of 0.8:

```matlab
gW = rand(3,2);
lp.lr = 0.5;
lp.mc = 0.8;
```

Because learnngdm only needs these values to calculate a weight change (see "Algorithm" below), use them to do so. Use the default initial learning state.

```matlab
ls = [];
[dW,ls] = learnngdm([],[],[],[],[],[],[],gW,[],[],lp,ls)
```

learnngdm returns the weight change and a new learning state.

**Algorithms**

learnngdm calculates the weight change dW for a given neuron from the neuron’s input P and error E, the weight (or bias) W, learning rate LR, and momentum constant MC, according to gradient descent with momentum:
\[ dW = mc \cdot dW_{\text{prev}} + (1 - mc) \cdot lr \cdot gW \]

The previous weight change \( dW_{\text{prev}} \) is stored and read from the learning state \( LS \).

**See Also**

adapt | learn | train

*Introduced before R2006a*
learnh

Hebb weight learning rule

Syntax

[dW,LS] = learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnh('code')

Description

learnh is the Hebb weight learning function.

[dW,LS] = learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

<table>
<thead>
<tr>
<th>W</th>
<th>S-by-R weight matrix (or S-by-1 bias vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>R-by-Q input vectors (or ones(1,Q))</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns

| dW   | S-by-R weight (or bias) change matrix       |

1-155
**LS** | New learning state
---|---

Learning occurs according to `learnh`'s learning parameter, shown here with its default value.

**LP.lr - 0.01** | Learning rate
---|---

`info = learnh('code')` returns useful information for each `code` character vector:

- **'pnames'** | Names of learning parameters
- **'pdefaults'** | Default learning parameters
- **'needg'** | Returns 1 if this function uses `gW` or `gA`

**Examples**

Here you define a random input `P` and output `A` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```matlab
p = rand(2,1);
a = rand(3,1);
lp.lr = 0.5;
```

Because `learnh` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnh([],p,[],[],a,[],[],[],[],[],[],lp,[])
```

**Network Use**

To prepare the weights and the bias of layer `i` of a custom network to learn with `learnh`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnh'`.

---

**1-156**
4 Set each `net.layerWeights{i,j}.learnFcn` to 'learnh'. (Each weight learning parameter property is automatically set to `learnh`'s default parameters.)

To train the network (or enable it to adapt),

1 Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2 Call `train` (adapt).

**Algorithms**

`learnh` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$, output $A$, and learning rate $LR$ according to the Hebb learning rule:

$$dW = LR * a * p'$$

**References**


**See Also**

adapt | `learnhd` | train

*Introduced before R2006a*
learnhd

Hebb with decay weight learning rule

Syntax

\[
[dW,LS] = \text{learnhd}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
\]
\[
\text{info} = \text{learnhd('code')}
\]

Description

learnhd is the Hebb weight learning function.

\[
[dW,LS] = \text{learnhd}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
\]
takes several inputs,

| \(W\)   | S-by-R weight matrix (or S-by-1 bias vector) |
| \(P\)   | R-by-Q input vectors (or \text{ones(1,Q)})  |
| \(Z\)   | S-by-Q weighted input vectors               |
| \(N\)   | S-by-Q net input vectors                    |
| \(A\)   | S-by-Q output vectors                       |
| \(T\)   | S-by-Q layer target vectors                 |
| \(E\)   | S-by-Q layer error vectors                  |
| \(gW\)  | S-by-R gradient with respect to performance |
| \(gA\)  | S-by-Q output gradient with respect to performance |
| \(D\)   | S-by-S neuron distances                     |
| \(LP\)  | Learning parameters, none, \(LP = []\)      |
| \(LS\)  | Learning state, initially should be = []    |

and returns

| \(dW\) | S-by-R weight (or bias) change matrix       |
**New learning state**

Learning occurs according to learnhd’s learning parameters, shown here with default values.

<table>
<thead>
<tr>
<th>LS</th>
<th>New learning state</th>
</tr>
</thead>
</table>

LP.dr - 0.01  
LP.lr - 0.1  

Decay rate  
Learning rate

info = learnhd('code') returns useful information for each code character vector:

| 'pnames' | Names of learning parameters |
| 'pdefaults' | Default learning parameters |
| 'needg' | Returns 1 if this function uses gW or gA |

### Examples

Here you define a random input P, output A, and weights W for a layer with a two-element input and three neurons. Also define the decay and learning rates.

```matlab
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.dr = 0.05;
lp.lr = 0.5;
```

Because learnhd only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnhd(w,p,[],[],a,[],[],[],[],[],[],[],lp,[])
```

### Network Use

To prepare the weights and the bias of layer i of a custom network to learn with learnhd,

1. Set net.trainFcn to 'trainr'. (net.trainParam automatically becomes trainr’s default parameters.)
2 Set `net.adaptFcn` to 'trains'. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3 Set each `net.inputWeights{i,j}.learnFcn` to 'learnhd'.
4 Set each `net.layerWeights{i,j}.learnFcn` to 'learnhd'. (Each weight learning parameter property is automatically set to `learnhd`'s default parameters.)

To train the network (or enable it to adapt),
1 Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2 Call `train` (adapt).

#### Algorithms

`learnhd` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$, output $A$, decay rate $DR$, and learning rate $LR$ according to the Hebb with decay learning rule:

$$dw = lr*a*p' - dr*w$$

#### See Also

adapt | learnh | train

*Introduced before R2006a*
**learnis**

Instar weight learning function

**Syntax**

```
[dW,LS] = learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnis('code')
```

**Description**

`learnis` is the instar weight learning function.

```
[dW,LS] = learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>S-by-R weight matrix (or S-by-1 bias vector)</td>
</tr>
<tr>
<td>P</td>
<td>R-by-Q input vectors (or <code>ones(1,Q)</code>)</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dW</td>
<td>S-by-R weight (or bias) change matrix</td>
</tr>
</tbody>
</table>
Learning occurs according to learnis’s learning parameter, shown here with its default value.

<table>
<thead>
<tr>
<th>LS</th>
<th>New learning state</th>
</tr>
</thead>
</table>

| LP.lr - 0.01 | Learning rate |

info = learnis('code') returns useful information for each code character vector:

<table>
<thead>
<tr>
<th>'pnames'</th>
<th>Names of learning parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

**Examples**

Here you define a random input P, output A, and weight matrix W for a layer with a two-element input and three neurons. Also define the learning rate LR.

```matlab
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Because learnis only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnis(w,p,[],[],a,[],[],[],[],[],[],[],[],[],lp,[])
```

**Network Use**

To prepare the weights and the bias of layer i of a custom network so that it can learn with learnis,

1. Set net.trainFcn to 'trainr'. (net.trainParam automatically becomes trainr’s default parameters.)
2. Set net.adaptFcn to 'trains'. (net.adaptParam automatically becomes trains’s default parameters.)
3  Set each `net.inputWeights{i,j}.learnFcn` to `'learnis'`.
4  Set each `net.layerWeights{i,j}.learnFcn` to `'learnis'`. (Each weight learning parameter property is automatically set to `learnis`'s default parameters.)

To train the network (or enable it to adapt),

1  Set `net.trainParam (net.adaptParam)` properties to desired values.
2  Call `train (adapt)`.

**Algorithms**

`learnis` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$, output $A$, and learning rate $LR$ according to the instar learning rule:

$$dw = lr*a*(p'\cdot w)$$

**References**


**See Also**

`adapt | learnk | learnos | train`

**Introduced before R2006a**
learnk

Kohonen weight learning function

Syntax

[dW,LS] = learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnk('code')

Description

learnk is the Kohonen weight learning function.

[dW,LS] = learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>S-by-R weight matrix (or S-by-1 bias vector)</td>
</tr>
<tr>
<td>P</td>
<td>R-by-Q input vectors (or ones(1,Q))</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dW</td>
<td>S-by-R weight (or bias) change matrix</td>
</tr>
</tbody>
</table>
New learning state

Learning occurs according to `learnk`'s learning parameter, shown here with its default value.

| **LP.lr** | 0.01 | Learning rate |

`info = learnk('code')` returns useful information for each `code` character vector:

| 'pnames' | Names of learning parameters |
| 'pdefaults' | Default learning parameters |
| 'needg' | Returns 1 if this function uses `gW` or `gA` |

**Examples**

Here you define a random input `P`, output `A`, and weight matrix `W` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```matlab
p = rand(2,1); a = rand(3,1); w = rand(3,2); lp.lr = 0.5;
```

Because `learnk` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnk(w,p,[],[],a,[],[],[],[],[],lp,[])
```

**Network Use**

To prepare the weights of layer `i` of a custom network to learn with `learnk`,

1. Set `net.trainFcn` to 'trainr'. (`net.trainParam` automatically becomes `trainr`'s default parameters.)
2. Set `net.adaptFcn` to 'trains'. (`net.adaptParam` automatically becomes `trains`'s default parameters.)
3. Set each `net.inputWeights{i,j}.learnFcn` to 'learnk'.

1-165
Set each net.layerWeights{i,j}.learnFcn to 'learnk'. (Each weight learning parameter property is automatically set to learnk’s default parameters.)

To train the network (or enable it to adapt),

1. Set net.trainParam (or net.adaptParam) properties as desired.
2. Call train (or adapt).

**Algorithms**

learnk calculates the weight change $\text{dW}$ for a given neuron from the neuron’s input $P$, output $A$, and learning rate $LR$ according to the Kohonen learning rule:

$$\text{dw} = lr^*(p'-w), \text{if } a \neq 0; = 0, \text{otherwise}$$

**References**


**See Also**

adapt | learnis | learnos | train

*Introduced before R2006a*
learnlv1

LVQ1 weight learning function

Syntax

[dW,LS] = learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnlv1('code')

Description

learnlv1 is the LVQ1 weight learning function.

[dW,LS] = learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

<table>
<thead>
<tr>
<th>W</th>
<th>S-by-R weight matrix (or S-by-1 bias vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>R-by-Q input vectors (or ones(1,Q))</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns

| dW     | S-by-R weight (or bias) change matrix       |
New learning state

Learning occurs according to `learnlv1`'s learning parameter, shown here with its default value.

| LP.lr - 0.01 | Learning rate |

```
info = learnlv1('code')
```

returns useful information for each `code` character vector:

| 'pnames' | Names of learning parameters |
| 'pdefaults' | Default learning parameters |
| 'needg' | Returns 1 if this function uses gW or gA |

**Examples**

Here you define a random input `P`, output `A`, weight matrix `W`, and output gradient `gA` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```
p = rand(2,1);
w = rand(3,2);
a = compet(negdist(w,p));
gA = [-1;1; 1];
lp.lr = 0.5;
```

Because `learnlv1` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```
dW = learnlv1(w,p,[],[],a,[],[],[],gA,[],[],lp,[])`
```

**Network Use**

You can create a standard network that uses `learnlv1` with `lvqnet`. To prepare the weights of layer `i` of a custom network to learn with `learnlv1`,

1. Set `net.trainFcn` to 'trainr'. (net.trainParam automatically becomes trainr’s default parameters.)
2. Set `net.adaptFcn` to 'trains'. (net.adaptParam automatically becomes trains’s default parameters.)
3 Set each `net.inputWeights{i,j}.learnFcn` to 'learnlv1'.
4 Set each `net.layerWeights{i,j}.learnFcn` to 'learnlv1'. (Each weight
   learning parameter property is automatically set to `learnlv1`'s default parameters.)

To train the network (or enable it to adapt),

1 Set `net.trainParam` (or `net.adaptParam`) properties as desired.
2 Call `train` (or `adapt`).

**Algorithms**

`learnlv1` calculates the weight change `dW` for a given neuron from the neuron's input `P`,
output `A`, output gradient `gA`, and learning rate `LR`, according to the LVQ1 rule, given `i`,
the index of the neuron whose output `a(i)` is 1:

\[
dw(i,:) = \begin{cases} 
+lr*(p-w(i,:)) & \text{if } gA(i) = 0 \\
-lr*(p-w(i,:)) & \text{if } gA(i) = -1 
\end{cases}
\]

**See Also**

`adapt` | `learnlv2` | `train`

*Introduced before R2006a*
learnlv2

LVQ2.1 weight learning function

Syntax

info = learnlv2('code')

Description

learnlv2 is the LVQ2 weight learning function.

[dW,LS] = learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

| W    | S-by-R weight matrix (or S-by-1 bias vector) |
| P    | R-by-Q input vectors (or ones(1,Q))          |
| Z    | S-by-Q weighted input vectors               |
| N    | S-by-Q net input vectors                    |
| A    | S-by-Q output vectors                       |
| T    | S-by-Q layer target vectors                 |
| E    | S-by-Q layer error vectors                  |
| gW   | S-by-R weight gradient with respect to performance |
| gA   | S-by-Q output gradient with respect to performance |
| D    | S-by-S neuron distances                     |
| LP   | Learning parameters, none, LP = []           |
| LS   | Learning state, initially should be = []     |

and returns

| dW    | S-by-R weight (or bias) change matrix       |
**New learning state**

Learning occurs according to `learnlv2`'s learning parameter, shown here with its default value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP.lr</td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td>LP.window</td>
<td>0.25</td>
<td>Window size (0 to 1, typically 0.2 to 0.3)</td>
</tr>
</tbody>
</table>

`info = learnlv2('code')` returns useful information for each `code` character vector:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pnames'</td>
<td>Names of learning parameters</td>
</tr>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

**Examples**

Here you define a sample input `P`, output `A`, weight matrix `W`, and output gradient `gA` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```matlab
p = rand(2,1);
w = rand(3,2);
n = negdist(w,p);
a = compet(n);
gA = [-1;1; 1];
lp.lr = 0.5;
```

Because `learnlv2` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnlv2(w,p,[],n,a,[],[],[],gA,[],lp,[])
```

**Network Use**

You can create a standard network that uses `learnlv2` with `lvqnet`.

To prepare the weights of layer `i` of a custom network to learn with `learnlv2`,

1-171
1 Set net.trainFcn to 'trainr'. (net.trainParam automatically becomes trainr’s default parameters.)

2 Set net.adaptFcn to 'trains'. (net.adaptParam automatically becomes trains’s default parameters.)

3 Set each net.inputWeights{i,j}.learnFcn to 'learnlv2'.

4 Set each net.layerWeights{i,j}.learnFcn to 'learnlv2'. (Each weight learning parameter property is automatically set to learnlv2’s default parameters.)

To train the network (or enable it to adapt),

1 Set net.trainParam (or net.adaptParam) properties as desired.

2 Call train (or adapt).

**Algorithms**

learnlv2 implements Learning Vector Quantization 2.1, which works as follows:

For each presentation, if the winning neuron i should not have won, and the runnerup j should have, and the distance $d_i$ between the winning neuron and the input $p$ is roughly equal to the distance $d_j$ from the runnerup neuron to the input $p$ according to the given window,

$$\min(d_i/d_j, d_j/d_i) > (1-\text{window})/(1+\text{window})$$

then move the winning neuron i weights away from the input vector, and move the runnerup neuron j weights toward the input according to

$$dw(i,:) = - \text{lp.lr}*(p'-w(i,:))$$
$$dw(j,:) = + \text{lp.lr}*(p'-w(j,:))$$

**See Also**

adapt | learnlv1 | train

*Introduced before R2006a*
learnos

Outstar weight learning function

Syntax

\[ [dW,LS] = \text{learnos}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \]
\[ \text{info} = \text{learnos}(\text{'code'}) \]

Description

learnos is the outstar weight learning function.

\[ [dW,LS] = \text{learnos}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \] takes several inputs,

| \( W \) | S-by-R weight matrix (or S-by-1 bias vector) |
| \( P \) | R-by-Q input vectors (or \( \text{ones}(1,Q) \)) |
| \( Z \) | S-by-Q weighted input vectors |
| \( N \) | S-by-Q net input vectors |
| \( A \) | S-by-Q output vectors |
| \( T \) | S-by-Q layer target vectors |
| \( E \) | S-by-Q layer error vectors |
| \( gW \) | S-by-R weight gradient with respect to performance |
| \( gA \) | S-by-Q output gradient with respect to performance |
| \( D \) | S-by-S neuron distances |
| \( LP \) | Learning parameters, none, \( LP = [] \) |
| \( LS \) | Learning state, initially should be = [] |

and returns

| \( dW \) | S-by-R weight (or bias) change matrix |
Learning occurs according to `learnos`'s learning parameter, shown here with its default value.

Learning rate info = `learnos('code')` returns useful information for each `code` character vector:

<table>
<thead>
<tr>
<th>'pnames'</th>
<th>Names of learning parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses <code>gW</code> or <code>gA</code></td>
</tr>
</tbody>
</table>

**Examples**

Here you define a random input `P`, output `A`, and weight matrix `W` for a layer with a two-element input and three neurons. Also define the learning rate `LR`.

```matlab
p = rand(2,1);
a = rand(3,1);
w = rand(3,2);
lp.lr = 0.5;
```

Because `learnos` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnos(w,p,[],[],a,[],[],[],[],[],lp,[])
```

**Network Use**

To prepare the weights and the bias of layer `i` of a custom network to learn with `learnos`,

1. Set `net.trainFcn` to `'trainr'`. (`net.trainParam` automatically becomes `trainr`’s default parameters.)
2. Set `net.adaptFcn` to `'trains'`. (`net.adaptParam` automatically becomes `trains`’s default parameters.)
3 Set each `net.inputWeights{i,j}.learnFcn` to 'learnos'.
4 Set each `net.layerWeights{i,j}.learnFcn` to 'learnos'. (Each weight learning parameter property is automatically set to `learnos`'s default parameters.)

To train the network (or enable it to adapt),

1 Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2 Call `train` (adapt).

**Algorithms**

`learnos` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$, output $A$, and learning rate $LR$ according to the outstar learning rule:

$$dw = lr*(a-w)*p'$$

**References**


**See Also**

`adapt` | `learnis` | `learnk` | `train`

**Introduced before R2006a**
**learnp**

Perceptron weight and bias learning function

**Syntax**

\[
[dW,LS] = \text{learnp}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
\]

\[
\text{info} = \text{learnp('code')}
\]

**Description**

`learnp` is the perceptron weight/bias learning function.

\[ [dW,LS] = \text{learnp}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \] takes several inputs,

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>S-by-R weight matrix (or b, and S-by-1 bias vector)</td>
</tr>
<tr>
<td>P</td>
<td>R-by-Q input vectors (or <code>ones(1,Q)</code>)</td>
</tr>
<tr>
<td>Z</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>N</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>A</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>E</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>gW</td>
<td>S-by-R weight gradient with respect to performance</td>
</tr>
<tr>
<td>gA</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>D</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>LP</td>
<td>Learning parameters, none, LP = []</td>
</tr>
<tr>
<td>LS</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dW</td>
<td>S-by-R weight (or bias) change matrix</td>
</tr>
</tbody>
</table>
info = learnp('code') returns useful information for each code character vector:

<table>
<thead>
<tr>
<th>'pnames'</th>
<th>Names of learning parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

**Examples**

Here you define a random input $P$ and error $E$ for a layer with a two-element input and three neurons.

```matlab
p = rand(2,1);
e = rand(3,1);
```

Because `learnp` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnp([],p,[],[],[],[],[],[],[],[],[],e,[],[],[],[],[],[],[],[])```

**Algorithms**

`learnp` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$ and error $E$ according to the perceptron learning rule:

$$dw = \begin{cases} 
0, & \text{if } e = 0 \\
p', & \text{if } e = 1 \\
-p', & \text{if } e = -1
\end{cases}$$

This can be summarized as

$$dw = e*p'$$

**References**

See Also
adapt | learnpn | train

Introduced before R2006a
Learnpn

Normalized perceptron weight and bias learning function

Syntax

\[ [dW,LS] = \text{learnpn}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \]
\[ \text{info} = \text{learnpn('code')} \]

Description

Learnpn is a weight and bias learning function. It can result in faster learning than learnp when input vectors have widely varying magnitudes.

\[ [dW,LS] = \text{learnpn}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \] takes several inputs,

| \( W \) | S-by-R weight matrix (or S-by-1 bias vector) |
| \( P \) | R-by-Q input vectors (or ones(1,Q)) |
| \( Z \) | S-by-Q weighted input vectors |
| \( N \) | S-by-Q net input vectors |
| \( A \) | S-by-Q output vectors |
| \( T \) | S-by-Q layer target vectors |
| \( E \) | S-by-Q layer error vectors |
| \( gW \) | S-by-R weight gradient with respect to performance |
| \( gA \) | S-by-Q output gradient with respect to performance |
| \( D \) | S-by-S neuron distances |
| \( LP \) | Learning parameters, none, \( LP = [\] \) |
| \( LS \) | Learning state, initially should be \( = [\] \) |

and returns
dW  | S-by-R weight (or bias) change matrix
---  | -------------------------------
LS   | New learning state

info = learnpn('code') returns useful information for each code character vector:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pnames'</td>
<td>Names of learning parameters</td>
</tr>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

### Examples

Here you define a random input P and error E for a layer with a two-element input and three neurons.

```matlab
p = rand(2,1);
e = rand(3,1);
```

Because learnpn only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnpn([],p,[],[],[],[],e,[],[],[],[])
```

### Limitations

Perceptrons do have one real limitation. The set of input vectors must be linearly separable if a solution is to be found. That is, if the input vectors with targets of 1 cannot be separated by a line or hyperplane from the input vectors associated with values of 0, the perceptron will never be able to classify them correctly.

### Algorithms

learnpn calculates the weight change dW for a given neuron from the neuron’s input P and error E according to the normalized perceptron learning rule:

\[
pn = p / \sqrt{1 + p(1)^2 + p(2)^2 + \ldots + p(R)^2}
\]

\[
dw = 0, \quad \text{if } e = 0
\]
\[ \begin{align*}
  &= \text{pn}', \text{ if } e = 1 \\
  &= -\text{pn}', \text{ if } e = -1
\end{align*} \]

The expression for \(dW\) can be summarized as

\[ dw = e \times \text{pn}' \]

**See Also**

adapt | learnp | train

**Introduced before R2006a**
**learnsom**

Self-organizing map weight learning function

**Syntax**

\[
[dW,LS] = learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
\]

\[
info = learnsom('code')
\]

**Description**

`learnsom` is the self-organizing map weight learning function.

\[
[dW,LS] = learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
\]

takes several inputs,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>W</strong></td>
<td>S-by-R weight matrix (or S-by-1 bias vector)</td>
</tr>
<tr>
<td><strong>P</strong></td>
<td>R-by-Q input vectors (or <code>ones(1,Q)</code>)</td>
</tr>
<tr>
<td><strong>Z</strong></td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td><strong>gW</strong></td>
<td>S-by-R weight gradient with respect to performance</td>
</tr>
<tr>
<td><strong>gA</strong></td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td><strong>LP</strong></td>
<td>Learning parameters, none, <code>LP = []</code></td>
</tr>
<tr>
<td><strong>LS</strong></td>
<td>Learning state, initially should be = <code>[]</code></td>
</tr>
</tbody>
</table>

and returns

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dW</strong></td>
<td>S-by-R weight (or bias) change matrix</td>
</tr>
</tbody>
</table>
Learning occurs according to `learnsom`'s learning parameters, shown here with their default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LP.order_lr</code></td>
<td>0.9</td>
<td>Ordering phase learning rate</td>
</tr>
<tr>
<td><code>LP.order_steps</code></td>
<td>1000</td>
<td>Ordering phase steps</td>
</tr>
<tr>
<td><code>LP.tune_lr</code></td>
<td>0.02</td>
<td>Tuning phase learning rate</td>
</tr>
<tr>
<td><code>LP.tune_nd</code></td>
<td>1</td>
<td>Tuning phase neighborhood distance</td>
</tr>
</tbody>
</table>

`info = learnsom('code')` returns useful information for each `code` character vector:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pnames'</td>
<td>Names of learning parameters</td>
</tr>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

**Examples**

Here you define a random input `P`, output `A`, and weight matrix `W` for a layer with a two-element input and six neurons. You also calculate positions and distances for the neurons, which are arranged in a 2-by-3 hexagonal pattern. Then you define the four learning parameters.

```matlab
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp.order_lr = 0.9;
lp.order_steps = 1000;
lp.tune_lr = 0.02;
lp.tune_nd = 1;
```

Because `learnsom` only needs these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
ls = [];
[dW,ls] = learnsom(w,p,[],[],a,[],[],[],[],d,lp,ls)
```
Algorithms

`learnsom` calculates the weight change $dW$ for a given neuron from the neuron’s input $P$, activation $A2$, and learning rate $LR$:

$$dw = lr*a2*(p' - w)$$

where the activation $A2$ is found from the layer output $A$, neuron distances $D$, and the current neighborhood size $ND$:

$$a2(i,q) = 1, \text{ if } a(i,q) = 1$$
$$= 0.5, \text{ if } a(j,q) = 1 \text{ and } D(i,j) \leq \text{ nd}$$
$$= 0, \text{ otherwise}$$

The learning rate $LR$ and neighborhood size $NS$ are altered through two phases: an ordering phase and a tuning phase.

The ordering phases lasts as many steps as `LP.order_steps`. During this phase $LR$ is adjusted from `LP.order_lr` down to `LP.tune_lr`, and $ND$ is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase $LR$ decreases slowly from `LP.tune_lr`, and $ND$ is always set to `LP.tune_nd`. During this phase the weights are expected to spread out relatively evenly over the input space while retaining their topological order, determined during the ordering phase.

See Also

adapt | train

Introduced before R2006a
**learnsomb**

Batch self-organizing map weight learning function

**Syntax**

\[
[dW, LS] = \text{learnsomb}(W, P, Z, N, A, T, E, gW, gA, D, LP, LS) \\
\text{info} = \text{learnsomb}('\text{code}')
\]

**Description**

`learnsomb` is the batch self-organizing map weight learning function.

\[
[dW, LS] = \text{learnsomb}(W, P, Z, N, A, T, E, gW, gA, D, LP, LS)
\]

takes several inputs:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(W)</td>
<td>S-by-R weight matrix (or S-by-1 bias vector)</td>
</tr>
<tr>
<td>(P)</td>
<td>R-by-Q input vectors (or \text{ones}(1, Q))</td>
</tr>
<tr>
<td>(Z)</td>
<td>S-by-Q weighted input vectors</td>
</tr>
<tr>
<td>(N)</td>
<td>S-by-Q net input vectors</td>
</tr>
<tr>
<td>(A)</td>
<td>S-by-Q output vectors</td>
</tr>
<tr>
<td>(T)</td>
<td>S-by-Q layer target vectors</td>
</tr>
<tr>
<td>(E)</td>
<td>S-by-Q layer error vectors</td>
</tr>
<tr>
<td>(gW)</td>
<td>S-by-R gradient with respect to performance</td>
</tr>
<tr>
<td>(gA)</td>
<td>S-by-Q output gradient with respect to performance</td>
</tr>
<tr>
<td>(D)</td>
<td>S-by-S neuron distances</td>
</tr>
<tr>
<td>(LP)</td>
<td>Learning parameters, none, (LP = [])</td>
</tr>
<tr>
<td>(LS)</td>
<td>Learning state, initially should be = []</td>
</tr>
</tbody>
</table>

and returns the following:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dW)</td>
<td>S-by-R weight (or bias) change matrix</td>
</tr>
</tbody>
</table>
Learning occurs according to `learnsomb`'s learning parameter, shown here with its default value:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP.init_neighborhood</td>
<td>3</td>
<td>Initial neighborhood size</td>
</tr>
<tr>
<td>LP.steps</td>
<td>100</td>
<td>Ordering phase steps</td>
</tr>
</tbody>
</table>

`info = learnsomb('code')` returns useful information for each `code` character vector:

- `'pnames'` Returns names of learning parameters.
- `'pdefaults'` Returns default learning parameters.
- `'needg'` Returns 1 if this function uses `gW` or `gA`.

### Examples

This example defines a random input `P`, output `A`, and weight matrix `W` for a layer with a 2-element input and 6 neurons. This example also calculates the positions and distances for the neurons, which appear in a 2-by-3 hexagonal pattern.

```matlab
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp = learnsomb('pdefaults');
```

Because `learnsomb` only needs these values to calculate a weight change (see Algorithm).

```matlab
ls = [];
[dW,ls] = learnsomb(w,p,[],[],a,[],[],[],[],d,lp,ls)
```

### Network Use

You can create a standard network that uses `learnsomb` with `selforgmap`. To prepare the weights of layer `i` of a custom network to learn with `learnsomb`:
1 Set NET.trainFcn to 'trainr'. (NET.trainParam automatically becomes trainr's default parameters.)
2 Set NET.adaptFcn to 'trains'. (NET.adaptParam automatically becomes trains's default parameters.)
3 Set each NET.inputWeights{i,j}.learnFcn to 'learnsomb'.
4 Set each NET.layerWeights{i,j}.learnFcn to 'learnsomb'. (Each weight learning parameter property is automatically set to learnsomb's default parameters.)

To train the network (or enable it to adapt):
1 Set NET.trainParam (or NET.adaptParam) properties as desired.
2 Call train (or adapt).

**Algorithms**

learnsomb calculates the weight changes so that each neuron's new weight vector is the weighted average of the input vectors that the neuron and neurons in its neighborhood responded to with an output of 1.

The ordering phase lasts as many steps as LP.steps.

During this phase, the neighborhood is gradually reduced from a maximum size of LP.init_neighborhood down to 1, where it remains from then on.

**See Also**

adapt | selforgmap | train

**Introduced in R2008a**
learnwh

Widrow-Hoff weight/bias learning function

Syntax

\[ [dW,LS] = \text{learnwh}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \]
\[ \text{info} = \text{learnwh('code')} \]

Description

learnwh is the Widrow-Hoff weight/bias learning function, and is also known as the delta or least mean squared (LMS) rule.

\[ [dW,LS] = \text{learnwh}(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) \] takes several inputs,

| \( W \) | S-by-R weight matrix (or \( b \), and S-by-1 bias vector) |
| \( P \) | R-by-Q input vectors (or \( \text{ones}(1,Q) \)) |
| \( Z \) | S-by-Q weighted input vectors |
| \( N \) | S-by-Q net input vectors |
| \( A \) | S-by-Q output vectors |
| \( T \) | S-by-Q layer target vectors |
| \( E \) | S-by-Q layer error vectors |
| \( gW \) | S-by-R weight gradient with respect to performance |
| \( gA \) | S-by-Q output gradient with respect to performance |
| \( D \) | S-by-S neuron distances |
| \( LP \) | Learning parameters, none, \( LP = [ ] \) |
| \( LS \) | Learning state, initially should be = [ ] |

and returns
<table>
<thead>
<tr>
<th>dW</th>
<th>S-by-R weight (or bias) change matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>New learning state</td>
</tr>
</tbody>
</table>

Learning occurs according to the `learnwh` learning parameter, shown here with its default value.

<table>
<thead>
<tr>
<th>LP.lr -</th>
<th>Learning rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>

`info = learnwh('code')` returns useful information for each `code` character vector:

<table>
<thead>
<tr>
<th>'pnames'</th>
<th>Names of learning parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdefaults'</td>
<td>Default learning parameters</td>
</tr>
<tr>
<td>'needg'</td>
<td>Returns 1 if this function uses gW or gA</td>
</tr>
</tbody>
</table>

### Examples

Here you define a random input `P` and error `E` for a layer with a two-element input and three neurons. You also define the learning rate `LR` learning parameter.

```matlab
p = rand(2,1);
e = rand(3,1);
lp.lr = 0.5;
```

Because `learnwh` needs only these values to calculate a weight change (see “Algorithm” below), use them to do so.

```matlab
dW = learnwh([],p,[],[],[],[],e,[],[],[],lp,[])
```

### Network Use

You can create a standard network that uses `learnwh` with `linearlayer`.

To prepare the weights and the bias of layer `i` of a custom network to learn with `learnwh`,

1. Set `net.trainFcn` to `'trainb'`. `net.trainParam` automatically becomes `trainb`'s default parameters.
Set `net.adaptFcn` to 'trains'. `net.adaptParam` automatically becomes `trains`'s default parameters.

Set each `net.inputWeights{i,j}.learnFcn` to 'learnwh'.

Set each `net.layerWeights{i,j}.learnFcn` to 'learnwh'.

Set `net.biases{i}.learnFcn` to 'learnwh'. Each weight and bias learning parameter property is automatically set to the `learnwh` default parameters.

To train the network (or enable it to adapt),

1. Set `net.trainParam` (or `net.adaptParam`) properties to desired values.
2. Call `train` (or `adapt`).

Algorithms

`learnwh` calculates the weight change `dW` for a given neuron from the neuron’s input `P` and error `E`, and the weight (or bias) learning rate `LR`, according to the Widrow-Hoff learning rule:

\[ dw = lr \cdot e \cdot p_n' \]

References


See Also

`adapt` | `linearlayer` | `train`

Introduced before R2006a
**linearlayer**

Linear layer

**Syntax**

`linearlayer(inputDelays,widrowHoffLR)`

**Description**

Linear layers are single layers of linear neurons. They may be static, with input delays of 0, or dynamic, with input delays greater than 0. They can be trained on simple linear time series problems, but often are used adaptively to continue learning while deployed so they can adjust to changes in the relationship between inputs and outputs while being used.

If a network is needed to solve a nonlinear time series relationship, then better networks to try include `timedelaynet`, `narxnet`, and `narnet`.

`linearlayer(inputDelays,widrowHoffLR)` takes these arguments,

<table>
<thead>
<tr>
<th>inputDelays</th>
<th>Row vector of increasing 0 or positive delays (default = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>widrowHoffLR</td>
<td>Widrow-Hoff learning rate (default = 0.01)</td>
</tr>
</tbody>
</table>

and returns a linear layer.

If the learning rate is too small, learning will happen very slowly. However, a greater danger is that it may be too large and learning will become unstable resulting in large changes to weight vectors and errors increasing instead of decreasing. If a data set is available which characterizes the relationship the layer is to learn, the maximum stable learning rate can be calculated with `maxlinlr`. 
Examples

Create and Train a Linear Layer

Here a linear layer is trained on a simple time series problem.

\[
x = \{0 -1 1 1 0 -1 1 0 0 1\};
\]
\[
t = \{0 -1 0 2 1 -1 0 1 0 1\};
\]
\[
\text{net} = \text{linearlayer}(1:2,0.01);
\]
\[
[Xs,Xi,Ai,Ts] = \text{preparets}(\text{net},x,t);
\]
\[
\text{net} = \text{train}(\text{net},Xs,Ts,Xi,Ai);
\]
\[
\text{view}(\text{net})
\]
\[
Y = \text{net}(Xs,Xi);
\]
\[
\text{perf} = \text{perform}(\text{net},Ts,Y)
\]

\[
\text{perf} =
\]

\[
0.2396
\]

See Also

narnet | narxnet | preparets | removedelay | timedelaynet

Introduced in R2010b
**linkdist**

Link distance function

**Syntax**

\[ d = \text{linkdist}(\text{pos}) \]

**Description**

`linkdist` is a layer distance function used to find the distances between the layer’s neurons given their positions.

\[ d = \text{linkdist}(\text{pos}) \] takes one argument,

| pos        | N-by-S matrix of neuron positions |

and returns the S-by-S matrix of distances.

**Examples**

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and find their distances.

\[
\text{pos} = \text{rand}(3,10);
\text{D} = \text{linkdist}(\text{pos})
\]

**Network Use**

You can create a standard network that uses `linkdist` as a distance function by calling `selforgmap`.

To change a network so that a layer’s topology uses `linkdist`, set `net.layers{i}.distanceFcn` to 'linkdist'.
In either case, call `sim` to simulate the network with `dist`.

**Algorithms**

The link distance $D$ between two position vectors $P_i$ and $P_j$ from a set of $S$ vectors is

$$D_{ij} =
\begin{cases} 
0, & \text{if } i == j \\
1, & \text{if } (\text{sum}((P_i-P_j).^2)).^0.5 \leq 1 \\
2, & \text{if } k \text{ exists, } D_{ik} = D_{kj} = 1 \\
3, & \text{if } k_1, k_2 \text{ exist, } D_{ik1} = D_{k1k2} = D_{k2j} = 1 \\
N, & \text{if } k_1..k_N \text{ exist, } D_{ik1} = D_{k1k2} = ... = D_{kNj} = 1 \\
S, & \text{if none of the above conditions apply}
\end{cases}$$

**See Also**

`dist`, `mandist`, `selforgmap`, `sim`

*Introduced before R2006a*
**logsig**

Log-sigmoid transfer function

**Graph and Symbol**

![Log-Sigmoid Transfer Function Graph](image)

\[ a = \text{logsig}(n) \]

Log-Sigmoid Transfer Function

**Syntax**

\[
A = \text{logsig}(N, FP) \\
dA_dN = \text{logsig}('dn',N,A,FP) \\
\text{info} = \text{logsig}('\text{code}')
\]

**Description**

logsig is a transfer function. Transfer functions calculate a layer’s output from its net input.

\[
A = \text{logsig}(N, FP)
\]

<table>
<thead>
<tr>
<th>N</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns A, the S-by-Q matrix of N’s elements squashed into [0, 1].
dA_dN = logsig('dn',N,A,FP) returns the S-by-Q derivative of A with respect to N. If A or FP is not supplied or is set to [], FP reverts to the default parameters, and A is calculated from N.

info = logsig('code') returns useful information for each code character vector:

logsig('name') returns the name of this function.

logsig('output',FP) returns the [min max] output range.

logsig('active',FP) returns the [min max] active input range.

logsig('fullderiv') returns 1 or 0, depending on whether dA_dN is S-by-S-by-Q or S-by-Q.

logsig('fpnames') returns the names of the function parameters.

logsig('fpdefaults') returns the default function parameters.

Examples

Here is the code to create a plot of the logsig transfer function.

n = -5:0.1:5;
A = logsig(n);
plot(n,A)

Assign this transfer function to layer i of a network.

net.layers{i}.transferFcn = 'logsig';

Algorithms

logsig(n) = 1 / (1 + exp(-n))

See Also

sim | tansig
Introduced before R2006a
**lvqnet**

Learning vector quantization neural network

**Syntax**

`lvqnet(hiddenSize, lvqLR, lvqLF)`

**Description**

LVQ (learning vector quantization) neural networks consist of two layers. The first layer maps input vectors into clusters that are found by the network during training. The second layer merges groups of first layer clusters into the classes defined by the target data.

The total number of first layer clusters is determined by the number of hidden neurons. The larger the hidden layer the more clusters the first layer can learn, and the more complex mapping of input to target classes can be made. The relative number of first layer clusters assigned to each target class are determined according to the distribution of target classes at the time of network initialization. This occurs when the network is automatically configured the first time `train` is called, or manually configured with the function `configure`, or manually initialized with the function `init` is called.

`lvqnet(hiddenSize, lvqLR, lvqLF)` takes these arguments,

<table>
<thead>
<tr>
<th>hiddenSize</th>
<th>Size of hidden layer (default = 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvqLR</td>
<td>LVQ learning rate (default = 0.01)</td>
</tr>
<tr>
<td>lvqLF</td>
<td>LVQ learning function (default = 'learnlv1')</td>
</tr>
</tbody>
</table>

and returns an LVQ neural network.

The other option for the `lvq` learning function is `learnlv2`. 

1-198
Examples

Train a Learning Vector Quantization Network

Here, an LVQ network is trained to classify iris flowers.

```matlab
[x,t] = iris_dataset;
net = lvqnet(10);
net.trainParam.epochs = 50;
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,y,t)
classes = vec2ind(y);
```

perf =

```
0.0489
```

See Also

competlayer | patternnet | selforgmap

Introduced in R2010b
lvqoutputs

LVQ outputs processing function

Syntax

[X, settings] = lvqoutputs(X)
X = lvqoutputs('apply', X, PS)
X = lvqoutputs('reverse', X, PS)
dx_dy = lvqoutputs('dx_dy', X, X, PS)

Description

[X, settings] = lvqoutputs(X) returns its argument unchanged, but stores the ratio of target classes in the settings for use by initlvq to initialize weights.

X = lvqoutputs('apply', X, PS) returns X.

X = lvqoutputs('reverse', X, PS) returns X.

dx_dy = lvqoutputs('dx_dy', X, X, PS) returns the identity derivative.

See Also
initlvq | lvqnet

Introduced in R2010b
**mae**

Mean absolute error performance function

**Syntax**

```matlab
perf = mae(E,Y,X,FP)
```

**Description**

`mae` is a network performance function. It measures network performance as the mean of absolute errors.

`perf = mae(E,Y,X,FP)` takes `E` and optional function parameters,

<table>
<thead>
<tr>
<th>E</th>
<th>Matrix or cell array of error vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Matrix or cell array of output vectors (ignored)</td>
</tr>
<tr>
<td>X</td>
<td>Vector of all weight and bias values (ignored)</td>
</tr>
<tr>
<td>FP</td>
<td>Function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns the mean absolute error.

```matlab
dPerf_dx = mae('dx',E,Y,X,perf,FP)```

returns the derivative of `perf` with respect to `X`.

`info = mae('code')` returns useful information for each `code` character vector:

- `mae('name')` returns the name of this function.
- `mae('pnames')` returns the names of the training parameters.
- `mae('pdefaults')` returns the default function parameters.
Examples

Create and configure a perceptron to have one input and one neuron:

```matlab
net = perceptron;
net = configure(net,0,0);
```

The network is given a batch of inputs \( P \). The error is calculated by subtracting the output \( A \) from target \( T \). Then the mean absolute error is calculated.

```matlab
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = net(p)
e = t-y
perf = mae(e)
```

Note that \( \text{mae} \) can be called with only one argument because the other arguments are ignored. \( \text{mae} \) supports those arguments to conform to the standard performance function argument list.

Network Use

You can create a standard network that uses \( \text{mae} \) with \text{perceptron}.

To prepare a custom network to be trained with \( \text{mae} \), set \( \text{net.performFcn} \) to 'mae'. This automatically sets \( \text{net.performParam} \) to the empty matrix [], because \( \text{mae} \) has no performance parameters.

In either case, calling \text{train} or \text{adapt}, results in \( \text{mae} \) being used to calculate performance.

See Also

\( \text{mse} \) | \text{perceptron}

Introduced before R2006a
mandist

Manhattan distance weight function

Syntax

\[ Z = \text{mandist}(W,P) \]
\[ D = \text{mandist}(\text{pos}) \]

Description

\texttt{mandist} is the Manhattan distance weight function. Weight functions apply weights to an input to get weighted inputs.

\[ Z = \text{mandist}(W,P) \] takes these inputs,

| \( W \)  | S-by-R weight matrix |
| \( P \)  | R-by-Q matrix of Q input (column) vectors |

and returns the S-by-Q matrix of vector distances.

\texttt{mandist} is also a layer distance function, which can be used to find the distances between neurons in a layer.

\[ D = \text{mandist}(\text{pos}) \] takes one argument,

| \( \text{pos} \)  | S row matrix of neuron positions |

and returns the S-by-S matrix of distances.

Examples

Here you define a random weight matrix \( W \) and input vector \( P \) and calculate the corresponding weighted input \( Z \).
W = rand(4,3);
P = rand(3,1);
Z = mandist(W,P)

Here you define a random matrix of positions for 10 neurons arranged in three-dimensional space and then find their distances.

pos = rand(3,10);
D = mandist(pos)

### Network Use

To change a network so an input weight uses mandist, set
net.inputWeights{i,j}.weightFcn to 'mandist'. For a layer weight, set
net.layerWeights{i,j}.weightFcn to 'mandist'.

To change a network so a layer’s topology uses mandist, set
net.layers{i}.distanceFcn to 'mandist'.

In either case, call `sim` to simulate the network with `dist`. See `newpnn` or `newgrnn` for simulation examples.

### Algorithms

The Manhattan distance $D$ between two vectors $X$ and $Y$ is

$$D = \text{sum}(|x-y|)$$

### See Also

dist | linkdist | sim

*Introduced before R2006a*
mapminmax

Process matrices by mapping row minimum and maximum values to [-1 1]

Syntax

\[
\begin{align*}
[Y,PS] &= \text{mapminmax}(X,YMIN,YMAX) \\
[Y,PS] &= \text{mapminmax}(X,FP) \\
Y &= \text{mapminmax}('apply',X,PS) \\
X &= \text{mapminmax}('reverse',Y,PS) \\
dx\_dy &= \text{mapminmax}('dx\_dy',X,Y,PS)
\end{align*}
\]

Description

mapminmax processes matrices by normalizing the minimum and maximum values of each row to [YMIN, YMAX].

\[
\begin{array}{|c|c|}
\hline
X & \text{N-by-Q matrix} \\
YMIN & \text{Minimum value for each row of } Y \text{ (default is } -1) \\
YMAX & \text{Maximum value for each row of } Y \text{ (default is } +1) \\
\hline
\end{array}
\]

and returns

\[
\begin{array}{|c|c|}
\hline
Y & \text{N-by-Q matrix} \\
PS & \text{Process settings that allow consistent processing of values} \\
\hline
\end{array}
\]

\[
\begin{align*}
[Y,PS] &= \text{mapminmax}(X,FP) \text{ takes parameters as a struct: } FP.ymin, FP.ymax. \\
Y &= \text{mapminmax}('apply',X,PS) \text{ returns } Y, \text{ given } X \text{ and settings } PS. \\
X &= \text{mapminmax}('reverse',Y,PS) \text{ returns } X, \text{ given } Y \text{ and settings } PS. \\
dx\_dy &= \text{mapminmax}('dx\_dy',X,Y,PS) \text{ returns the reverse derivative.}
\end{align*}
\]
Examples

Here is how to format a matrix so that the minimum and maximum values of each row are mapped to default interval [-1,+1].

\[
x_1 = \begin{bmatrix} 1 & 2 & 4; & 1 & 1 & 1; & 3 & 2 & 2; & 0 & 0 & 0 \end{bmatrix}
\]

\[y_1,PS] = \text{mapminmax}(x_1)\]

Next, apply the same processing settings to new values.

\[
x_2 = \begin{bmatrix} 5 & 2 & 3; & 1 & 1 & 1; & 6 & 7 & 3; & 0 & 0 & 0 \end{bmatrix}
\]

\[y_2 = \text{mapminmax}('\text{apply}',x_2,PS)\]

Reverse the processing of \(y_1\) to get \(x_1\) again.

\[x_1\text{\_again} = \text{mapminmax}('\text{reverse}',y_1,PS)\]

More About

Normalize Inputs and Targets Using \text{mapminmax}

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function \text{mapminmax} scales inputs and targets so that they fall in the range [-1,1]. The following code illustrates how to use this function.

\[
[pn,ps] = \text{mapminmax}(p);
\[tn,ts] = \text{mapminmax}(t);
\text{net} = \text{train(\text{net},pn,tn)};
\]

The original network inputs and targets are given in the matrices \(p\) and \(t\). The normalized inputs and targets \(pn\) and \(tn\) that are returned will all fall in the interval [-1,1]. The structures \(ps\) and \(ts\) contain the settings, in this case the minimum and maximum values of the original inputs and targets. After the network has been trained, the \(ps\) settings should be used to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If \text{mapminmax} is used to scale the targets, then the output of the network will be trained to produce outputs in the range [-1,1]. To convert these outputs back into the same units that were used for the original targets, use the settings \(ts\). The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.
\texttt{an} = \texttt{sim}(\texttt{net}, \texttt{pn});
\texttt{a} = \texttt{mapminmax}(\texttt{\textquoteleft reverse\textquoteright, an, ts});

The network output \texttt{an} corresponds to the normalized targets \texttt{tn}. The unnormalized network output \texttt{a} is in the same units as the original targets \texttt{t}.

If \texttt{mapminmax} is used to preprocess the training set data, then whenever the trained network is used with new inputs they should be preprocessed with the minimum and maximums that were computed for the training set stored in the settings \texttt{ps}. The following code applies a new set of inputs to the network already trained.

\texttt{pnewn} = \texttt{mapminmax}(\texttt{\textquoteleft apply\textquoteright, pnew, ps});
\texttt{anewn} = \texttt{sim}(\texttt{net}, \texttt{pnewn});
\texttt{anew} = \texttt{mapminmax}(\texttt{\textquoteleft reverse\textquoteright, anewn, ts});

For most networks, including \texttt{feedforwardnet}, these steps are done automatically, so that you only need to use the \texttt{sim} command.

\section*{Algorithms}

It is assumed that \(X\) has only finite real values, and that the elements of each row are not all equal. (If \(\text{\texttt{xmax}}=\text{\texttt{xmin}}\) or if either \(\text{\texttt{xmax}}\) or \(\text{\texttt{xmin}}\) are non-finite, then \(y=x\) and no change occurs.)

\[ y = (\text{\texttt{ymax}}-\text{\texttt{ymin}})*(\text{\texttt{x}}-\text{\texttt{xmin}})/(\text{\texttt{xmax}}-\text{\texttt{xmin}}) + \text{\texttt{ymin}}; \]

\section*{See Also}
\texttt{fixunknowns} | \texttt{mapstd} | \texttt{processpca}

\texttt{Introduced in R2006a}
**mapstd**

Process matrices by mapping each row’s means to 0 and deviations to 1

**Syntax**

\[
[Y,PS] = \text{mapstd}(X,\text{ymean},\text{ystd})
\]

\[
[Y,PS] = \text{mapstd}(X,\text{FP})
\]

\[
Y = \text{mapstd}('apply',X,PS)
\]

\[
X = \text{mapstd}('reverse',Y,PS)
\]

\[
dx_dy = \text{mapstd}('dx_dy',X,Y,PS)
\]

**Description**

mapstd processes matrices by transforming the mean and standard deviation of each row to ymean and ystd.

\[
[Y,PS] = \text{mapstd}(X,\text{ymean},\text{ystd})
\]
takes X and optional parameters,

<table>
<thead>
<tr>
<th>X</th>
<th>N-by-Q matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymean</td>
<td>Mean value for each row of Y (default is 0)</td>
</tr>
<tr>
<td>ystd</td>
<td>Standard deviation for each row of Y (default is 1)</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Y</th>
<th>N-by-Q matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>Process settings that allow consistent processing of values</td>
</tr>
</tbody>
</table>

\[
[Y,PS] = \text{mapstd}(X,\text{FP})
\]
takes parameters as a struct: FP.ymean, FP.ystd.

\[
Y = \text{mapstd}('apply',X,PS)
\]
returns Y, given X and settings PS.

\[
X = \text{mapstd}('reverse',Y,PS)
\]
returns X, given Y and settings PS.

\[
dx_dy = \text{mapstd}('dx_dy',X,Y,PS)
\]
returns the reverse derivative.
Examples

Here you format a matrix so that the minimum and maximum values of each row are mapped to default mean and STD of 0 and 1.

\[ x_1 = \begin{bmatrix} 1 & 2 & 4; 1 & 1 & 1; 3 & 2 & 2; 0 & 0 & 0 \end{bmatrix} \]
\[ [y_1,PS] = \text{mapstd}(x1) \]

Next, apply the same processing settings to new values.

\[ x_2 = \begin{bmatrix} 5 & 2 & 3; 1 & 1 & 1; 6 & 7 & 3; 0 & 0 & 0 \end{bmatrix} \]
\[ y_2 = \text{mapstd}('apply',x2,PS) \]

Reverse the processing of \( y_1 \) to get \( x_1 \) again.

\[ x_1\_again = \text{mapstd}('reverse',y1,PS) \]

More About

Normalize Network Inputs and Targets Using \texttt{mapstd}

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. The function \texttt{mapstd} normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of \texttt{mapstd}.

\[
[pn,ps] = \text{mapstd}(p);
[tn,ts] = \text{mapstd}(t);
\]

The original network inputs and targets are given in the matrices \( p \) and \( t \). The normalized inputs and targets \( pn \) and \( tn \) that are returned will have zero means and unity standard deviation. The settings structures \( ps \) and \( ts \) contain the means and standard deviations of the original inputs and original targets. After the network has been trained, you should use these settings to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If \texttt{mapstd} is used to scale the targets, then the output of the network is trained to produce outputs with zero mean and unity standard deviation. To convert these outputs back into the same units that were used for the original targets, use \( ts \). The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.
an = sim(net,pn);
a = mapstd('reverse',an,ts);

The network output an corresponds to the normalized targets tn. The unnormalized network output a is in the same units as the original targets t.

If mapstd is used to preprocess the training set data, then whenever the trained network is used with new inputs, you should preprocess them with the means and standard deviations that were computed for the training set using ps. The following commands apply a new set of inputs to the network already trained:

pnewn = mapstd('apply',pnew,ps);
anewn = sim(net,pnewn);
anew = mapstd('reverse',anewn,ts);

For most networks, including feedforwardnet, these steps are done automatically, so that you only need to use the sim command.

**Algorithms**

It is assumed that X has only finite real values, and that the elements of each row are not all equal.

\[ y = (x-xmean) \cdot \frac{ystd}{xstd} + ymean; \]

**See Also**

fixunknowns | mapminmax | processpca

**Introduced in R2006a**
maxlinlr

Maximum learning rate for linear layer

Syntax

\[
lr = \text{maxlinlr}(P) \\
lr = \text{maxlinlr}(P, 'bias')
\]

Description

maxlinlr is used to calculate learning rates for linearlayer.

\[
lr = \text{maxlinlr}(P)
\]

takes one argument, 

\[
P \quad \text{R-by-Q matrix of input vectors}
\]

and returns the maximum learning rate for a linear layer without a bias that is to be trained only on the vectors in \( P \).

\[
lr = \text{maxlinlr}(P, 'bias')
\]

returns the maximum learning rate for a linear layer with a bias.

Examples

Here you define a batch of four two-element input vectors and find the maximum learning rate for a linear layer with a bias.

\[
P = [1 \ 2 \ -4 \ 7; \ 0.1 \ 3 \ 10 \ 6]; \\
lr = \text{maxlinlr}(P, 'bias')
\]

See Also

learnwh | linearlayer
Introduced before R2006a
meanabs

Mean of absolute elements of matrix or matrices

Syntax

[m,n] = meanabs(x)

Description

[m,n] = meanabs(x) takes a matrix or cell array of matrices and returns,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Mean value of all absolute finite values</td>
</tr>
<tr>
<td>n</td>
<td>Number of finite values</td>
</tr>
</tbody>
</table>

If x contains no finite values, the mean returned is 0.

Examples

m = meanabs([1 2;3 4])
[m,n] = meanabs({[1 2; NaN 4], [4 5; 2 3]})

See Also

meansqr | sumabs | sumsqr

Introduced in R2010b
meansqr
Mean of squared elements of matrix or matrices

Syntax

\[ [m,n] = \text{meansqr}(x) \]

Description

\[ [m,n] = \text{meansqr}(x) \]

\hspace{1cm} \text{takes a matrix or cell array of matrices and returns,}

\[ \begin{array}{|c|c|} \hline
\text{m} & \text{Mean value of all squared finite values} \\
\hline
\text{n} & \text{Number of finite values} \\
\hline
\end{array} \]

If \( x \) contains no finite values, the mean returned is 0.

Examples

\[ m = \text{meansqr}([1 \ 2; 3 \ 4]) \]
\[ [m,n] = \text{meansqr}({[1 \ 2; \text{NaN} \ 4], [4 \ 5; 2 \ 3]}) \]

See Also

\text{meanabs} | \text{sumabs} | \text{sumsqr}

Introduced in R2010b
**midpoint**

Midpoint weight initialization function

**Syntax**

\[ W = \text{midpoint}(S,PR) \]

**Description**

`midpoint` is a weight initialization function that sets weight (row) vectors to the center of the input ranges.

\[ W = \text{midpoint}(S,PR) \]

takes two arguments, 

<table>
<thead>
<tr>
<th>S</th>
<th>Number of rows (neurons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>R-by-Q matrix of input value ranges = [Pmin Pmax]</td>
</tr>
</tbody>
</table>

and returns an S-by-R matrix with rows set to \( (P_{\text{min}}+P_{\text{max}})'/2 \).

**Examples**

Here initial weight values are calculated for a five-neuron layer with input elements ranging over [0 1] and [-2 2].

\[ W = \text{midpoint}(5,[0 1; -2 2]) \]

**See Also**

init | initlay | initwb

**Introduced before R2006a**
**minmax**

Ranges of matrix rows

**Syntax**

```matlab
pr = minmax(P)
```

**Description**

`pr = minmax(P)` takes one argument, `P`, an R-by-Q matrix, and returns the R-by-2 matrix `pr` of minimum and maximum values for each row of `P`.

Alternatively, `P` can be an M-by-N cell array of matrices. Each matrix `P{i,j}` should have Ri rows and Q columns. In this case, `minmax` returns an M-by-1 cell array where the mth element is an Ri-by-2 matrix of the minimum and maximum values of elements for the matrix on the ith row of `P`.

**Examples**

```matlab
x = rands(4,5)
m = minmax(x)
x = nndata([1;2],3,4)
m = minmax(x)
```

*Introduced before R2006a*
mse

Mean squared normalized error performance function

Syntax

perf = mse(net,t,y,ew)

Description

mse is a network performance function. It measures the network’s performance according to the mean of squared errors.

perf = mse(net,t,y,ew) takes these arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network</td>
</tr>
<tr>
<td>t</td>
<td>Matrix or cell array of targets</td>
</tr>
<tr>
<td>y</td>
<td>Matrix or cell array of outputs</td>
</tr>
<tr>
<td>ew</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>

and returns the mean squared error.

This function has two optional parameters, which are associated with networks whose net.trainFcn is set to this function:

- 'regularization' can be set to any value between 0 and 1. The greater the regularization value, the more squared weights and biases are included in the performance calculation relative to errors. The default is 0, corresponding to no regularization.

- 'normalization' can be set to 'none' (the default); 'standard', which normalizes errors between -2 and 2, corresponding to normalizing outputs and targets between -1 and 1; and 'percent', which normalizes errors between -1 and 1. This feature is useful for networks with multi-element outputs. It ensures that the relative accuracy of output elements with differing target value ranges are treated as equally important, instead of prioritizing the relative accuracy of the output element with the largest target value range.
You can create a standard network that uses mse with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with mse, set net.performFcn to 'mse'. This automatically sets net.performParam to a structure with the default optional parameter values.

**Examples**

**Train Neural Network Using mse Performance Function**

This example shows how to train a neural network using the mse performance function.

Here a two-layer feedforward network is created and trained to estimate body fat percentage using the mse performance function and a regularization value of 0.01.

```matlab
[x, t] = bodyfat_dataset;
net = feedforwardnet(10);
net.performParam.regularization = 0.01;

MSE is the default performance function for feedforwardnet.

net.performFcn
```

```matlab
ans =
    'mse'
```

Train the network and evaluate performance.

```matlab
net = train(net, x, t);
y = net(x);
perf = perform(net, t, y)
```

```matlab
perf = 20.7769
```

Alternatively, you can call mse directly.

```matlab
perf = mse(net, t, y, 'regularization', 0.01)
```

```matlab
perf = 20.7769
```
See Also

mae

Introduced before R2006a
narnet

Nonlinear autoregressive neural network

Syntax

narnet(feedbackDelays,hiddenSizes,feedbackMode,trainFcn)

Description

NAR (nonlinear autoregressive) neural networks can be trained to predict a time series from that series past values.

narnet(feedbackDelays,hiddenSizes,feedbackMode,trainFcn) takes these arguments,

<table>
<thead>
<tr>
<th>feedbackDelays</th>
<th>Row vector of increasing 0 or positive delays (default = 1:2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hiddenSizes</td>
<td>Row vector of one or more hidden layer sizes (default = 10)</td>
</tr>
<tr>
<td>feedbackMode</td>
<td>One of 'open', 'closed', or 'none' (default is 'open')</td>
</tr>
<tr>
<td>trainFcn</td>
<td>Training function (default is 'trainlm')</td>
</tr>
</tbody>
</table>

and returns a NAR neural network.

Examples

Train NAR Network and Predict on New Data

Load the simple time-series prediction data and create a NAR network.

```plaintext
T = simplenar_dataset;
net = narnet(1:2,10);
```
Prepare the time series data using `preparets` and train the network.

```matlab
[Xs,Xi,Ai,Ts] = preparets(net,{},{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```

Calculate the network performance.

```matlab
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y)
```

```matlab
perf =

1.0100e-09
```

To predict the output for the next 20 time steps, first simulate the network in closed loop form.

```matlab
[netc,Xic,Aic] = closeloop(net,Xf,Af);
view(netc)
```
The network only has one input. In closed loop mode, this input is joined to the output.

To simulate the network 20 time steps ahead, input an empty cell array of length 20. The network requires only the initial conditions given in $X_{ic}$ and $A_{ic}$.

$$y_2 = \text{netc}([0,20],X_{ic},A_{ic})$$

$y_2 =$

1x20 cell array

Columns 1 through 5

$$\begin{bmatrix}
{[0.8346]} & {[0.3329]} & {[0.9084]} & {[1.0000]} & {[0.3190]}
\end{bmatrix}$$

Columns 6 through 10

$$\begin{bmatrix}
{[0.7329]} & {[0.9801]} & {[0.6409]} & {[0.5146]} & {[0.9746]}
\end{bmatrix}$$

Columns 11 through 15

$$\begin{bmatrix}
{[0.9077]} & {[0.2807]} & {[0.8651]} & {[0.9897]} & {[0.4093]}
\end{bmatrix}$$

Columns 16 through 20

$\begin{bmatrix}
{[1.0000]} & {[0.3190]} & {[0.9746]} & {[0.4093]}
\end{bmatrix}$
See Also
narnet|narxnet|preparets|removedelay|timedelaynet

Introduced in R2010b
**narxnet**

Nonlinear autoregressive neural network with external input

**Syntax**

```matlab
narxnet(inputDelays,feedbackDelays,hiddenSizes,feedbackMode,trainFcn)
```

**Description**

NARX (Nonlinear autoregressive with external input) networks can learn to predict one time series given past values of the same time series, the feedback input, and another time series, called the external or exogenous time series.

`narxnet(inputDelays,feedbackDelays,hiddenSizes,feedbackMode,trainFcn)` takes these arguments,

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputDelays</td>
<td>Row vector of increasing 0 or positive delays (default = 1:2)</td>
</tr>
<tr>
<td>feedbackDelays</td>
<td>Row vector of increasing 0 or positive delays (default = 1:2)</td>
</tr>
<tr>
<td>hiddenSizes</td>
<td>Row vector of one or more hidden layer sizes (default = 10)</td>
</tr>
<tr>
<td>feedbackMode</td>
<td>One of 'open', 'closed', or 'none' (default is 'open')</td>
</tr>
<tr>
<td>trainFcn</td>
<td>Training function (default is 'trainlm')</td>
</tr>
</tbody>
</table>

and returns a NARX neural network.

**Examples**
Train NARX Network and Predict on New Data

Partition the training data. Use \( X_{\text{new}} \) to do prediction in closed loop mode later.

\[
[X,T] = \text{simpleseries\_dataset};
X_{\text{new}} = X(81:100);
X = X(1:80);
T = T(1:80);
\]

Train a network, and simulate it on the first 80 observations

\[
\text{net} = \text{narxnet}(1:2,1:2,10);
[X_s,X_i,A_i,T_s] = \text{preparets}(\text{net},X,{},T);
\text{net} = \text{train}(\text{net},X_s,T_s,X_i,A_i);
\text{view}(\text{net})
\]

Calculate the network performance.

\[
[Y,X_f,A_f] = \text{net}(X_s,X_i,A_i);
\text{perf} = \text{perform}(\text{net},T_s,Y)
\]

\[
\text{perf} =
0.0153
\]

Run the prediction for 20 time steps ahead in closed loop mode.

\[
[\text{netc},X_{ic},A_{ic}] = \text{closeloop}(\text{net},X_f,A_f);
\text{view}(\text{netc})
\]
\( y_2 = \text{netc}(X_{\text{new}}, X_{\text{ic}}, A_{\text{ic}}) \)

\( y_2 = \)

1x20 cell array

Columns 1 through 5

\[
\begin{bmatrix}
[-0.0156] & [0.1133] & [-0.1472] & [-0.0706] & [0.0355]
\end{bmatrix}
\]

Columns 6 through 10

\[
\begin{bmatrix}
[-0.2829] & [0.2047] & [-0.3809] & [-0.2836] & [0.1886]
\end{bmatrix}
\]

Columns 11 through 15

\[
\begin{bmatrix}
[-0.1813] & [0.1373] & [0.2189] & [0.3122] & [0.2346]
\end{bmatrix}
\]

Columns 16 through 20

\[
\begin{bmatrix}
[-0.0156] & [0.0724] & [0.3395] & [0.1940] & [0.0757]
\end{bmatrix}
\]

See Also
closeloop | narnet | openloop | preparets | removedelay | timedelaynet
Introduced in R2010b
nctool

Neural network classification or clustering tool

Syntax

nctool

Description

nctool opens the Neural Net Clustering GUI.

For more information and an example of its usage, see “Cluster Data with a Self-Organizing Map”.

Algorithms

nctool leads you through solving a clustering problem using a self-organizing map. The map forms a compressed representation of the inputs space, reflecting both the relative density of input vectors in that space, and a two-dimensional compressed representation of the input-space topology.

See Also

nftool | nprtool | ntstool

Introduced in R2008a
**negdist**

Negative distance weight function

**Syntax**

\[ Z = \text{negdist}(W,P) \]
\[ \text{dim} = \text{negdist}('\text{size}',S,R,\text{FP}) \]
\[ dw = \text{negdist}('\text{dz}\_\text{dw}',W,P,Z,\text{FP}) \]

**Description**

`negdist` is a weight function. Weight functions apply weights to an input to get weighted inputs.

\[ Z = \text{negdist}(W,P) \]

takes these inputs,

<table>
<thead>
<tr>
<th>W</th>
<th>S-by-R weight matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>R-by-Q matrix of Q input (column) vectors</td>
</tr>
<tr>
<td>FP</td>
<td>Row cell array of function parameters (optional, ignored)</td>
</tr>
</tbody>
</table>

and returns the S-by-Q matrix of negative vector distances.

\[ \text{dim} = \text{negdist}('\text{size}',S,R,\text{FP}) \]

takes the layer dimension \( S \), input dimension \( R \), and function parameters, and returns the weight size [S-by-R].

\[ dw = \text{negdist}('\text{dz}\_\text{dw}',W,P,Z,\text{FP}) \]

returns the derivative of \( Z \) with respect to \( W \).

**Examples**

Here you define a random weight matrix \( W \) and input vector \( P \) and calculate the corresponding weighted input \( Z \).
\[ W = \text{rand}(4,3); \]
\[ P = \text{rand}(3,1); \]
\[ Z = \text{negdist}(W,P) \]

**Network Use**

You can create a standard network that uses `negdist` by calling `competlayer` or `selforgmap`.

To change a network so an input weight uses `negdist`, set `net.inputWeights{i,j}.weightFcn` to `'negdist'`. For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'negdist'`.

In either case, call `sim` to simulate the network with `negdist`.

**Algorithms**

`negdist` returns the negative Euclidean distance:

\[ z = -\sqrt{\sum (w-p)^2} \]

**See Also**

`competlayer` | `dist` | `dotprod` | `selforgmap` | `sim`

**Introduced before R2006a**
**netinv**

Inverse transfer function

**Syntax**

\[ A = \text{netinv}(N,FP) \]

**Description**

`netinv` is a transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \text{netinv}(N,FP) \] takes inputs

<table>
<thead>
<tr>
<th>( N )</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \( 1/N \).

\[ \text{info} = \text{netinv('code')} \] returns information about this function. The following codes are supported:

- `netinv('name')` returns the name of this function.
- `netinv('output',FP)` returns the \([\text{min max}]\) output range.
- `netinv('active',FP)` returns the \([\text{min max}]\) active input range.
- `netinv('fullderiv')` returns 1 or 0, depending on whether \( \frac{dA}{dN} \) is S-by-S-by-Q or S-by-Q.
- `netinv('fpnames')` returns the names of the function parameters.
- `netinv('fpdefaults')` returns the default function parameters.
Examples

Here you define 10 five-element net input vectors N and calculate A.

\[
\begin{align*}
n &= \text{rand}(5,10); \\
a &= \text{netinv}(n);
\end{align*}
\]

Assign this transfer function to layer \(i\) of a network.

\[
\text{net.layers\{i\}.transferFcn} = \text{netinv}';
\]

See Also

logsig | tansig

Introduced in R2006a
netprod

Product net input function

Syntax

N = netprod({Z1,Z2,...,Zn})
info = netprod('code')

Description

netprod is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

N = netprod({Z1,Z2,...,Zn}) takes

\[
\begin{array}{|c|}
\hline
Z_i & S\text{-by-}Q \text{ matrices in a row cell array} \\
\hline
\end{array}
\]

and returns an element-wise product of Z1 to Zn.

info = netprod('code') returns information about this function. The following codes are supported:

\[
\begin{array}{|c|c|}
\hline
\text{code} & \text{Description} \\
\hline
'deriv' & \text{Name of derivative function} \\
'fullderiv' & \text{Full N\text{-by-}S\text{-by-}Q \text{ derivative } = 1, \text{ element-wise } S\text{-by-}Q \text{ derivative } = 0} \\
'name' & \text{Full name} \\
'fpnames' & \text{Returns names of function parameters} \\
'fpdefaults' & \text{Returns default function parameters} \\
\hline
\end{array}
\]

Examples

Here netprod combines two sets of weighted input vectors (user-defined).
Z1 = [1 2 4;3 4 1];
Z2 = [-1 2 2; -5 -6 1];
Z = {Z1,Z2};
N = netprod(Z)

Here netprod combines the same weighted inputs with a bias vector. Because Z1 and Z2 each contain three concurrent vectors, three concurrent copies of B must be created with concur so that all sizes match.

B = [0; -1];
Z = {Z1, Z2, concur(B,3)};
N = netprod(Z)

**Network Use**

You can create a standard network that uses netprod by calling newpnn or newgrnn.

To change a network so that a layer uses netprod, set net.layers{i}.netInputFcn to 'netprod'.

In either case, call sim to simulate the network with netprod. See newpnn or newgrnn for simulation examples.

**See Also**

concur | netsum | sim

**Introduced before R2006a**
**netsum**

Sum net input function

**Syntax**

\[
N = \text{netsum}(\{Z_1, Z_2, \ldots, Z_n\}, \text{FP})
\]

\[
\text{info} = \text{netsum}(\text{'code'})
\]

**Description**

`netsum` is a net input function. Net input functions calculate a layer's net input by combining its weighted inputs and biases.

\[
N = \text{netsum}(\{Z_1, Z_2, \ldots, Z_n\}, \text{FP})
\]

takes \(Z_1\) to \(Z_n\) and optional function parameters, \(Z_i\) \(S\)-by-\(Q\) matrices in a row cell array \(\text{FP}\) Row cell array of function parameters (ignored)

and returns the elementwise sum of \(Z_1\) to \(Z_n\).

\[
\text{info} = \text{netsum}(\text{'code'})
\]

returns information about this function. The following codes are supported:

\[
\text{netsum('name')}
\]

returns the name of this function.

\[
\text{netsum('type')}
\]

returns the type of this function.

\[
\text{netsum('fpnames')}
\]

returns the names of the function parameters.

\[
\text{netsum('fpdefaults')}
\]

returns default function parameter values.

\[
\text{netsum('fpcheck', \text{FP})}
\]

throws an error for illegal function parameters.

\[
\text{netsum('fullderiv')}
\]

returns \(0\) or \(1\), depending on whether the derivative is \(S\)-by-\(Q\) or \(N\)-by-\(S\)-by-\(Q\).
Examples

Here netsum combines two sets of weighted input vectors and a bias. You must use concur to make b the same dimensions as z1 and z2.

\[
z_1 = [1, 2, 4; 3, 4, 1] \\
z_2 = [-1, 2, 2; -5, -6, 1] \\
b = [0; -1] \\
n = netsum([z_1, z_2, conur(b, 3)])
\]

Assign this net input function to the first layer of a network.

```matlab
net = feedforwardnet();
net.layers{1}.netInputFcn = 'netsum';
```

See Also
cascadeforwardnet | feedforwardnet | netinv | netprod

Introduced before R2006a
network

Create custom shallow neural network

Syntax

net = network
net = network(numInputs,numLayers,biasConnect,inputConnect,layerConnect,outputConnect)

To Get Help

Type help network/network.

Tip To learn how to create a deep learning network, see “Specify Layers of Convolutional Neural Network”.

Description

network creates new custom networks. It is used to create networks that are then customized by functions such as feedforwardnet and narxnet.

net = network without arguments returns a new neural network with no inputs, layers or outputs.

net = network(numInputs,numLayers,biasConnect,inputConnect,layerConnect,outputConnect) takes these optional arguments (shown with default values):

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numInputs</td>
<td>Number of inputs, 0</td>
</tr>
<tr>
<td>numLayers</td>
<td>Number of layers, 0</td>
</tr>
<tr>
<td>biasConnect</td>
<td>numLayers-by-1 Boolean vector, zeros</td>
</tr>
</tbody>
</table>
and returns

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>net</strong></td>
<td>New network with the given property values</td>
</tr>
</tbody>
</table>

**Properties**

**Architecture Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.numInputs</td>
<td>0 or a positive integer</td>
<td>Number of inputs.</td>
</tr>
<tr>
<td>net.numLayers</td>
<td>0 or a positive integer</td>
<td>Number of layers.</td>
</tr>
<tr>
<td>net.biasConnect</td>
<td>numLayer-by-1 Boolean vector</td>
<td>If net.biasConnect(i) is 1, then layer i has a bias, and net.biases{i} is a structure describing that bias.</td>
</tr>
<tr>
<td>net.inputConnect</td>
<td>numLayer-by-numInputs Boolean vector</td>
<td>If net.inputConnect(i,j) is 1, then layer i has a weight coming from input j, and net.inputWeights{i,j} is a structure describing that weight.</td>
</tr>
<tr>
<td>net.layerConnect</td>
<td>numLayer-by-numLayers Boolean vector</td>
<td>If net.layerConnect(i,j) is 1, then layer i has a weight coming from layer j, and net.layerWeights{i,j} is a structure describing that weight.</td>
</tr>
<tr>
<td>net.outputConnect</td>
<td>1-by-numLayers Boolean vector</td>
<td>If net.outputConnect(i) is 1, then the network has an output from layer i, and net.outputs{i} is a structure describing that output.</td>
</tr>
<tr>
<td>net.numOutputs</td>
<td>0 or a positive integer (read only)</td>
<td>Number of network outputs according to net.outputConnect.</td>
</tr>
</tbody>
</table>
Subobject Structure Properties

<table>
<thead>
<tr>
<th>net.inputs</th>
<th>numInputs-by-1 cell array</th>
<th>net.inputs{i} is a structure defining input i.</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.layers</td>
<td>numLayers-by-1 cell array</td>
<td>net.layers{i} is a structure defining layer i.</td>
</tr>
<tr>
<td>net.biases</td>
<td>numLayers-by-1 cell array</td>
<td>If net.biasConnect(i) is 1, then net.biases{i} is a structure defining the bias for layer i.</td>
</tr>
<tr>
<td>net.inputWeights</td>
<td>numLayers-by-numInputs cell array</td>
<td>If net.inputConnect(i,j) is 1, then net.inputWeights{i,j} is a structure defining the weight to layer i from input j.</td>
</tr>
<tr>
<td>net.layerWeights</td>
<td>numLayers-by-numLayers cell array</td>
<td>If net.layerConnect(i,j) is 1, then net.layerWeights{i,j} is a structure defining the weight to layer i from layer j.</td>
</tr>
<tr>
<td>net.outputs</td>
<td>1-by-numLayers cell array</td>
<td>If net.outputConnect(i) is 1, then net.outputs{i} is a structure defining the network output from layer i.</td>
</tr>
</tbody>
</table>

Function Properties

<table>
<thead>
<tr>
<th>net.adaptFcn</th>
<th>Name of a network adaption function or ''</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.initFcn</td>
<td>Name of a network initialization function or ''</td>
</tr>
<tr>
<td>net.performFcn</td>
<td>Name of a network performance function or ''</td>
</tr>
<tr>
<td>net.trainFcn</td>
<td>Name of a network training function or ''</td>
</tr>
</tbody>
</table>
**Parameter Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.adaptParam</td>
<td>Network adaption parameters</td>
</tr>
<tr>
<td>net.initParam</td>
<td>Network initialization parameters</td>
</tr>
<tr>
<td>net.performParam</td>
<td>Network performance parameters</td>
</tr>
<tr>
<td>net.trainParam</td>
<td>Network training parameters</td>
</tr>
</tbody>
</table>

**Weight and Bias Value Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.IW</td>
<td>numLayers-by-numInputs cell array of input weight values</td>
</tr>
<tr>
<td>net.LW</td>
<td>numLayers-by-numLayers cell array of layer weight values</td>
</tr>
<tr>
<td>net.b</td>
<td>numLayers-by-1 cell array of bias values</td>
</tr>
</tbody>
</table>

**Other Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.userdata</td>
<td>Structure you can use to store useful values</td>
</tr>
</tbody>
</table>

**Examples**

**Create Network with One Input and Two Layers**

This example shows how to create a network without any inputs and layers, and then set its numbers of inputs and layers to 1 and 2 respectively.

```matlab
net = network
net.numInputs = 1
net.numLayers = 2
```

Alternatively, you can create the same network with one line of code.

```matlab
net = network(1,2)
```
Create Feedforward Network and View Properties

This example shows how to create a one-input, two-layer, feedforward network. Only the first layer has a bias. An input weight connects to layer 1 from input 1. A layer weight connects to layer 2 from layer 1. Layer 2 is a network output and has a target.

\[
\text{net} = \text{network}(1,2,[1;0],[1;0],[0 0; 1 0],[0 1])
\]

You can view the network subobjects with the following code.

\[
\begin{align*}
\text{net.inputs{1}} \\
\text{net.layers{1}}, \text{net.layers{2}} \\
\text{net.biases{1}} \\
\text{net.inputWeights{1,1}}, \text{net.layerWeights{2,1}} \\
\text{net.outputs{2}}
\end{align*}
\]

You can alter the properties of any of the network subobjects. This code changes the transfer functions of both layers:

\[
\begin{align*}
\text{net.layers{1}.transferFcn} &= \text{'tansig'}; \\
\text{net.layers{2}.transferFcn} &= \text{'logsig'};
\end{align*}
\]

You can view the weights for the connection from the first input to the first layer as follows. The weights for a connection from an input to a layer are stored in \text{net.IW}. If the values are not yet set, these result is empty.

\[
\text{net.IW{1,1}}
\]

You can view the weights for the connection from the first layer to the second layer as follows. Weights for a connection from a layer to a layer are stored in \text{net.LW}. Again, if the values are not yet set, the result is empty.

\[
\text{net.LW{2,1}}
\]

You can view the bias values for the first layer as follows.

\[
\text{net.b{1}}
\]

To change the number of elements in input 1 to 2, set each element’s range:

\[
\text{net.inputs{1}.range} = [0 1; -1 1];
\]

To simulate the network for a two-element input vector, the code might look like this:
p = [0.5; -0.1];
y = sim(net,p)

See Also
sim

Topics
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced before R2006a
**newgrnn**

Design generalized regression neural network

**Syntax**

net = newgrnn(P,T,spread)

**Description**

Generalized regression neural networks (grnns) are a kind of radial basis network that is often used for function approximation. grnns can be designed very quickly.

net = newgrnn(P,T,spread) takes three inputs,

<table>
<thead>
<tr>
<th>P</th>
<th>R-by-Q matrix of Q input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>S-by-Q matrix of Q target class vectors</td>
</tr>
<tr>
<td>spread</td>
<td>Spread of radial basis functions (default = 1.0)</td>
</tr>
</tbody>
</table>

and returns a new generalized regression neural network.

The larger the spread, the smoother the function approximation. To fit data very closely, use a spread smaller than the typical distance between input vectors. To fit the data more smoothly, use a larger spread.

**Properties**

newgrnn creates a two-layer network. The first layer has radbas neurons, and calculates weighted inputs with dist and net input with netprod. The second layer has purelin neurons, calculates weighted input with normprod, and net inputs with netsum. Only the first layer has biases.

newgrnn sets the first layer weights to P', and the first layer biases are all set to 0.8326/spread, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- spread. The second layer weights W2 are set to T.
Examples

Here you design a radial basis network, given inputs \( P \) and targets \( T \).

\[
P = [1 \ 2 \ 3];
T = [2.0 \ 4.1 \ 5.9];
net = newgrnn(P,T);
\]

The network is simulated for a new input.

\[
P = 1.5;
Y = sim(net,P)
\]

References


See Also

newpnn | newrb | newrbe | sim

Introduced before R2006a
newlind

Design linear layer

Syntax

`net = newlind(P,T,Pi)`

Description

`net = newlind(P,T,Pi)` takes these input arguments,

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>R-by-Q matrix of Q input vectors</td>
</tr>
<tr>
<td>T</td>
<td>S-by-Q matrix of Q target class vectors</td>
</tr>
<tr>
<td>Pi</td>
<td>1-by-ID cell array of initial input delay states</td>
</tr>
</tbody>
</table>

where each element `Pi{i,k}` is an `Ri`-by-Q matrix, and the default = []; and returns a linear layer designed to output T (with minimum sum square error) given input P.

`newlind(P,T,Pi)` can also solve for linear networks with input delays and multiple inputs and layers by supplying input and target data in cell array form:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Ni-by-TS cell array</td>
</tr>
<tr>
<td>T</td>
<td>Nt-by-TS cell array</td>
</tr>
<tr>
<td>Pi</td>
<td>Ni-by-ID cell array</td>
</tr>
</tbody>
</table>

Each element `P{i,ts}` is an `Ri`-by-Q input matrix, `T{i,ts}` is a `Vi`-by-Q matrix, and `Pi{i,k}` is an `Ri`-by-Q matrix, default = []

and returns a linear network with ID input delays, Ni network inputs, and Nl layers, designed to output T (with minimum sum square error) given input P.

Examples

You want a linear layer that outputs T given P for the following definitions:
P = [1 2 3];
T = [2.0 4.1 5.9];

Use newlind to design such a network and check its response.

net = newlind(P,T);
Y = sim(net,P)

You want another linear layer that outputs the sequence T given the sequence P and two initial input delay states Pi.

P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5.0 6.1 4.0 6.0 6.9 8.0};
net = newlind(P,T,Pi);
Y = sim(net,P,Pi)

You want a linear network with two outputs Y1 and Y2 that generate sequences T1 and T2, given the sequences P1 and P2, with three initial input delay states P11 for input 1 and three initial delays states P12 for input 2.

P1 = {1 2 1 3 3 2}; P11 = {1 3 0};
P2 = {1 2 1 1 2 1}; P12 = {2 1 2};
T1 = {5.0 6.1 4.0 6.0 6.9 8.0};
T2 = {11.0 12.1 10.1 10.9 13.0 13.0};
net = newlind([P1; P2],[T1; T2],[P11; P12]);
Y = sim(net,[P1; P2],[P11; P12]);
Y1 = Y(1,:)
Y2 = Y(2,:)

Algorithms

newlind calculates weight W and bias B values for a linear layer from inputs P and targets T by solving this linear equation in the least squares sense:

\[[W b] * [P; ones] = T\]

See Also

sim
Introduced before R2006a
newpnn

Design probabilistic neural network

Syntax

net = newpnn(P,T,spread)

Description

Probabilistic neural networks (PNN) are a kind of radial basis network suitable for classification problems.

net = newpnn(P,T,spread) takes two or three arguments,

<table>
<thead>
<tr>
<th>P</th>
<th>R-by-Q matrix of Q input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>S-by-Q matrix of Q target class vectors</td>
</tr>
<tr>
<td>spread</td>
<td>Spread of radial basis functions (default = 0.1)</td>
</tr>
</tbody>
</table>

and returns a new probabilistic neural network.

If spread is near zero, the network acts as a nearest neighbor classifier. As spread becomes larger, the designed network takes into account several nearby design vectors.

Examples

Here a classification problem is defined with a set of inputs P and class indices Tc.

P = [1 2 3 4 5 6 7];
Tc = [1 2 3 2 2 3 1];

The class indices are converted to target vectors, and a PNN is designed and tested.

T = ind2vec(Tc)
net = newpnn(P,T);
\[
Y = \text{sim}(\text{net}, P)
\]
\[
Y_c = \text{vec2ind}(Y)
\]

**Algorithms**

`newpnn` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist` and its net input with `netprod`. The second layer has `compet` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Only the first layer has biases.

`newpnn` sets the first-layer weights to \(P'\), and the first-layer biases are all set to \(0.8326 / \text{spread}\), resulting in radial basis functions that cross 0.5 at weighted inputs of +/- spread. The second-layer weights \(W_2\) are set to \(T\).

**References**


**See Also**

`ind2vec` | `newgrnn` | `newrb` | `newrbe` | `sim` | `vec2ind`

Introduced before R2006a
newrb

Design radial basis network

Syntax

net = newrb(P,T,goal,spread,MN,DF)

Description

Radial basis networks can be used to approximate functions. newrb adds neurons to the hidden layer of a radial basis network until it meets the specified mean squared error goal.

net = newrb(P,T,goal,spread,MN,DF) takes two of these arguments,

<table>
<thead>
<tr>
<th>P</th>
<th>R-by-Q matrix of Q input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>S-by-Q matrix of Q target class vectors</td>
</tr>
<tr>
<td>goal</td>
<td>Mean squared error goal (default = 0.0)</td>
</tr>
<tr>
<td>spread</td>
<td>Spread of radial basis functions (default = 1.0)</td>
</tr>
<tr>
<td>MN</td>
<td>Maximum number of neurons (default is Q)</td>
</tr>
<tr>
<td>DF</td>
<td>Number of neurons to add between displays (default = 25)</td>
</tr>
</tbody>
</table>

and returns a new radial basis network.

The larger spread is, the smoother the function approximation. Too large a spread means a lot of neurons are required to fit a fast-changing function. Too small a spread means many neurons are required to fit a smooth function, and the network might not generalize well. Call newrb with different spreads to find the best value for a given problem.

Examples

Here you design a radial basis network, given inputs P and targets T.
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T);

The network is simulated for a new input.

P = 1.5;
Y = sim(net,P)

**Algorithms**

newrb creates a two-layer network. The first layer has radbas neurons, and calculates its weighted inputs with dist and its net input with netprod. The second layer has purelin neurons, and calculates its weighted input with dotprod and its net inputs with netsum. Both layers have biases.

Initially the radbas layer has no neurons. The following steps are repeated until the network’s mean squared error falls below goal.

1. The network is simulated.
2. The input vector with the greatest error is found.
3. A radbas neuron is added with weights equal to that vector.
4. The purelin layer weights are redesigned to minimize error.

**See Also**

newgrnn | newpnn | newrbe | sim

*Introduced before R2006a*
newrbe

Design exact radial basis network

Syntax

net = newrbe(P,T,spread)

Description

Radial basis networks can be used to approximate functions. newrbe very quickly designs a radial basis network with zero error on the design vectors.

net = newrbe(P,T,spread) takes two or three arguments,

<table>
<thead>
<tr>
<th>P</th>
<th>RxQ matrix of Q R-element input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>SxQ matrix of Q S-element target class vectors</td>
</tr>
<tr>
<td>spread</td>
<td>Spread of radial basis functions (default = 1.0)</td>
</tr>
</tbody>
</table>

and returns a new exact radial basis network.

The larger the spread is, the smoother the function approximation will be. Too large a spread can cause numerical problems.

Examples

Here you design a radial basis network given inputs P and targets T.

P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrbe(P,T);

The network is simulated for a new input.

P = 1.5;
Y = sim(net,P)
Algorithms

newrbe creates a two-layer network. The first layer has rdbas neurons, and calculates its weighted inputs with dist and its net input with netprod. The second layer has purelin neurons, and calculates its weighted input with dotprod and its net inputs with netsum. Both layers have biases.

newrbe sets the first-layer weights to P', and the first-layer biases are all set to 0.8326/ spread, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- spread.

The second-layer weights IW{2,1} and biases b{2} are found by simulating the first-layer outputs A{1} and then solving the following linear expression:

\[ [W{2,1} b{2}] * [A{1}; ones] = T \]

See Also

newgrnn | newpnn | newrb | sim

Introduced before R2006a
nftool

Neural Net Fitting tool

Syntax

nftool

Description

nftool opens the Neural Net Fitting GUI.

For more information and an example of its usage, see “Fit Data with a Shallow Neural Network”.

Algorithms

nftool leads you through solving a data fitting problem, solving it with a two-layer feed-forward network trained with Levenberg-Marquardt.

See Also

nctool | npctool | ntstool

Introduced in R2006a
nnccell2mat

Combine neural network cell data into matrix

Syntax

[y, i, j] nncell2mat(x)

Description

[y, i, j] nncell2mat(x) takes a cell array of matrices and returns,

<table>
<thead>
<tr>
<th>y</th>
<th>Cell array formed by concatenating matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Array of row sizes</td>
</tr>
<tr>
<td>ji</td>
<td>Array of column sizes</td>
</tr>
</tbody>
</table>

The row and column sizes returned by nncell2mat can be used to convert the returned matrix back into a cell of matrices with mat2cell.

Examples

Here neural network data is converted to a matrix and back.

c = {rands(2,3) rands(2,3); rands(5,3) rands(5,3)};
[m, i, j] = nncell2mat(c)
c3 = mat2cell(m, i, j)

See Also
nndata | nnsize

Introduced in R2010b
**nncorr**

Cross correlation between neural network time series

**Syntax**

`nncorr(a,b,maxlag,'flag')`

**Description**

`nncorr(a,b,maxlag,'flag')` takes these arguments,

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td>Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of N.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of M.</td>
</tr>
<tr>
<td><code>maxlag</code></td>
<td>Maximum number of time lags</td>
</tr>
<tr>
<td><code>flag</code></td>
<td>Type of normalization (default = 'none')</td>
</tr>
</tbody>
</table>

and returns an N-by-M cell array where each `{i, j}` element is a 2*maxlag+1 length row vector formed from the correlations of `a` elements (i.e., matrix row) `i` and `b` elements (i.e., matrix column) `j`.

If `a` and `b` are specified with row vectors, the result is returned in matrix form.

The options for the normalization `flag` are:

- 'biased' — scales the raw cross-correlation by 1/N.
- 'unbiased' — scales the raw correlation by 1/(N-abs(k)), where `k` is the index into the result.
- 'coeff' — normalizes the sequence so that the correlations at zero lag are 1.0.
- 'none' — no scaling. This is the default.
Examples

Here the autocorrelation of a random 1-element, 1-sample, 20-timestep signal is calculated with a maximum lag of 10.

```matlab
a = nndata(1,1,20)
aa = nncorr(a,a,10)
```

Here the cross-correlation of the first signal with another random 2-element signal are found, with a maximum lag of 8.

```matlab
b = nndata(2,1,20)
ab = nncorr(a,b,8)
```

See Also

confusion | regression

Introduced in R2010b
**nndata**

Create neural network data

**Syntax**

\[ \text{nndata}(N,Q,TS,v) \]

**Description**

\[ \text{nndata}(N,Q,TS,v) \] takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>Vector of ( M ) element sizes</td>
</tr>
<tr>
<td>( Q )</td>
<td>Number of samples</td>
</tr>
<tr>
<td>( TS )</td>
<td>Number of timesteps</td>
</tr>
<tr>
<td>( v )</td>
<td>Scalar value</td>
</tr>
</tbody>
</table>

and returns an \( M \)-by-\( TS \) cell array where each row \( i \) has \( N(i) \)-by-\( Q \) sized matrices of value \( v \). If \( v \) is not specified, random values are returned.

You can access subsets of neural network data with \text{getelements}, \text{getsamples}, \text{gettimesteps}, and \text{getsignals}.

You can set subsets of neural network data with \text{setelements}, \text{setsamples}, \text{settimesteps}, and \text{setsignals}.

You can concatenate subsets of neural network data with \text{catelements}, \text{catsamples}, \text{cattimesteps}, and \text{catsignals}.

**Examples**

Here four samples of five timesteps, for a 2-element signal consisting of zero values is created:

\[ x = \text{nndata}(2,4,5,0) \]
To create random data with the same dimensions:

```matlab
x = nndata(2,4,5)
```

Here static (1 timestep) data of 12 samples of 4 elements is created.

```matlab
x = nndata(4,12)
```

**See Also**

fromnndata | nndata2sim | nnsize | sim2nndata | tonndata

**Introduced in R2010b**
nndata2gpu

Format neural data for efficient GPU training or simulation

Syntax

nndata2gpu(x)
[Y,Q,N,TS] = nndata2gpu(X)
nndata2gpu(X,PRECISION)

Description

nndata2gpu requires Parallel Computing Toolbox™.

nndata2gpu(x) takes an N-by-Q matrix X of Q N-element column vectors, and returns it in a form for neural network training and simulation on the current GPU device.

The N-by-Q matrix becomes a QQ-by-N gpuArray where QQ is Q rounded up to the next multiple of 32. The extra rows (Q+1):QQ are filled with NaN values. The gpuArray has the same precision ('single' or 'double') as X.

[Y,Q,N,TS] = nndata2gpu(X) can also take an M-by-TS cell array of M signals over TS time steps. Each element of X{i,ts} should be an Ni-by-Q matrix of Q Ni-element vectors, representing the ith signal vector at time step ts, across all Q time series. In this case, the gpuArray Y returned is QQ-by-(sum(Ni)*TS). Dimensions Ni, Q, and TS are also returned so they can be used with gpu2nndata to perform the reverse formatting.

nndata2gpu(X,PRECISION) specifies the default precision of the gpuArray, which can be 'double' or 'single'.

Examples

Copy a matrix to the GPU and back:
\begin{verbatim}
x = rand(5,6)
[y,q] = nndata2gpu(x)
x2 = gpu2nndata(y,q)

Copy neural network cell array data, representing four time series, each consisting of five time steps of 2-element and 3-element signals:

x = nnndata([2;3],4,5)
[y,q,n,ts] = nndata2gpu(x)
x2 = gpu2nndata(y,q,n,ts)
\end{verbatim}

See Also

\texttt{gpu2nndata}

\textbf{Introduced in R2012b}
**nndata2sim**

Convert neural network data to Simulink time series

**Syntax**

nndata2sim(x,i,q)

**Description**

nndata2sim(x,i,q) takes these arguments,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Neural network data</td>
</tr>
<tr>
<td>i</td>
<td>Index of signal (default = 1)</td>
</tr>
<tr>
<td>q</td>
<td>Index of sample (default = 1)</td>
</tr>
</tbody>
</table>

and returns time series q of signal i as a Simulink time series structure.

**Examples**

Here random neural network data is created with two signals having 4 and 3 elements respectively, over 10 timesteps. Three such series are created.

```matlab
x = nndata([4;3],3,10);
```

Now the second signal of the first series is converted to Simulink form.

```matlab
y_2_1 = nndata2sim(x,2,1)
```

**See Also**

nndata | nnsim | sim2nndata

**Introduced in R2010b**
**nnsize**

Number of neural data elements, samples, timesteps, and signals

**Syntax**

\[ [N,Q,TS,M] = nnsize(X) \]

**Description**

\[ [N,Q,TS,M] = nnsize(X) \] takes neural network data \( X \) and returns,

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>Vector containing the number of element sizes for each of ( M ) signals</td>
</tr>
<tr>
<td>( Q )</td>
<td>Number of samples</td>
</tr>
<tr>
<td>( TS )</td>
<td>Number of timesteps</td>
</tr>
<tr>
<td>( M )</td>
<td>Number of signals</td>
</tr>
</tbody>
</table>

If \( X \) is a matrix, \( N \) is the number of rows of \( X \), \( Q \) is the number of columns, and both \( TS \) and \( M \) are 1.

If \( X \) is a cell array, \( N \) is an \( S \times 1 \) vector, where \( M \) is the number of rows in \( X \), and \( N(i) \) is the number of rows in \( X\{i,1\} \). \( Q \) is the number of columns in the matrices in \( X \).

**Examples**

This code gets the dimensions of matrix data:

\[
X = \begin{bmatrix} 1 & 2 & 3; & 4 & 7 & 4 \end{bmatrix}
\]

\[
[n,q,ts,s] = nnsize(x)
\]

This code gets the dimensions of cell array data:

\[
X = \{[1:3; 4:6] \ [7:9; 10:12]; \ [13:15] \ [16:18]\}
\]

\[
[n,q,ts,s] = nnsize(x)
\]
See Also

nndata | numelements | numsamples | numsignals | numtimesteps

Introduced in R2010b
nnstart

Neural network getting started GUI

Syntax

nnstart

Description

nnstart opens a window with launch buttons for neural network fitting, pattern recognition, clustering and time series tools. It also provides links to lists of data sets, examples, and other useful information for getting started. See specific topics on “Getting Started with Deep Learning Toolbox”.

See Also

nctool | nftool | nprttool | ntstool

Introduced in R2010b
nntool

Open Network/Data Manager

Syntax

nntool

Description

nntool opens the Network/Data Manager window, which allows you to import, create, use, and export neural networks and data.

Note Although it is still available, nntool is no longer recommended. Instead, use nnstart, which provides graphical interfaces that allow you to design and deploy fitting, pattern recognition, clustering, and time-series neural networks.

See Also

nnstart

Introduced before R2006a
nntraintool

Neural network training tool

Syntax

nntraintool
nntraintool close
nntraintool('close')

Description

nntraintool opens the neural network training GUI.

This function can be called to make the training GUI visible before training has occurred, after training if the window has been closed, or just to bring the training GUI to the front.

Network training functions handle all activity within the training window.

To access additional useful plots, related to the current or last network trained, during or after training, click their respective buttons in the training window.

nntraintool close or nntraintool('close') closes the training window.

Introduced in R2008a
**noloop**

Remove neural network open- and closed-loop feedback

**Syntax**

```matlab
net = noloop(net)
```

**Description**

`net = noloop(net)` takes a neural network and returns the network with open- and closed-loop feedback removed.

For outputs `i`, where `net.outputs{i}.feedbackMode` is 'open', the feedback mode is set to 'none', `outputs{i}.feedbackInput` is set to the empty matrix, and the associated network input is deleted.

For outputs `i`, where `net.outputs{i}.feedbackMode` is 'closed', the feedback mode is set to 'none'.

**Examples**

Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```matlab
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai)
```

Now the network is converted to no loop form. The output and second input are no longer associated.

```matlab
net = noloop(net);
view(net)
```
[Xs,Xi,Ai] = preparets(net,X,T);
Y = net(Xs,Xi,Ai)

See Also
  closeloop | openloop

Introduced in R2010b
normc

Normalize columns of matrix

Syntax

normc(M)

Description

normc(M) normalizes the columns of M to a length of 1.

Examples

m = [1 2; 3 4];
normc(m)
ans =
    0.3162    0.4472
    0.9487    0.8944

See Also

normr

Introduced before R2006a
normprod

Normalized dot product weight function

Syntax

\[ Z = \text{normprod}(W,P,FP) \]
\[ \text{dim} = \text{normprod}('\text{size}',S,R,FP) \]
\[ dw = \text{normprod}('\text{dz}\_\text{dw}',W,P,Z,FP) \]

Description

normprod is a weight function. Weight functions apply weights to an input to get weighted inputs.

\[ Z = \text{normprod}(W,P,FP) \] takes these inputs,

<table>
<thead>
<tr>
<th>( W )</th>
<th>S-by-R weight matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>R-by-Q matrix of Q input (column) vectors</td>
</tr>
<tr>
<td>( FP )</td>
<td>Row cell array of function parameters (optional, ignored)</td>
</tr>
</tbody>
</table>

and returns the S-by-Q matrix of normalized dot products.

\[ \text{dim} = \text{normprod}('\text{size}',S,R,FP) \] takes the layer dimension \( S \), input dimension \( R \), and function parameters, and returns the weight size [S-by-R].

\[ dw = \text{normprod}('\text{dz}\_\text{dw}',W,P,Z,FP) \] returns the derivative of \( Z \) with respect to \( W \).

Examples

Here you define a random weight matrix \( W \) and input vector \( P \) and calculate the corresponding weighted input \( Z \).
W = rand(4,3);
P = rand(3,1);
Z = normprod(W,P)

**Network Use**

You can create a standard network that uses `normprod` by calling `newgrnn`.

To change a network so an input weight uses `normprod`, set
`net.inputWeights{i,j}.weightFcn` to 'normprod'. For a layer weight, set
`net.layerWeights{i,j}.weightFcn` to 'normprod'.

In either case, call `sim` to simulate the network with `normprod`. See `newgrnn` for simulation examples.

**Algorithms**

`normprod` returns the dot product normalized by the sum of the input vector elements.

\[ z = \frac{w*p}{\text{sum}(p)} \]

**See Also**

dotprod

*Introduced before R2006a*
normr

Normalize rows of matrix

**Syntax**

```
normr(M)
```

**Description**

`normr(M)` normalizes the rows of `M` to a length of 1.

**Examples**

```matlab
m = [1 2; 3 4];
normr(m)
ans =
    0.4472    0.8944
    0.6000    0.8000
```

**See Also**

`normc`

*Introduced before R2006a*
nprtool

Neural Net Pattern Recognition tool

Syntax

nprtool

Description

nprtool opens the Neural Net Pattern Recognition tool.

For more information and an example of its usage, see “Classify Patterns with a Shallow Neural Network”.

Algorithms

nprtool leads you through solving a pattern-recognition classification problem using a two-layer feed-forward patternnet network with sigmoid output neurons.

See Also

nctool | nftool | ntstool

Introduced in R2008a
ntstool

Neural network time series tool

Syntax

ntstool
ntstool('close')

Description

ntstool opens the neural network time series tool and leads you through solving a fitting problem using a two-layer feed-forward network.

For more information and an example of its usage, see “Shallow Neural Network Time-Series Prediction and Modeling”.

ntstool('close') closes the tool.

See Also

nctool | nftool | nprtool

Introduced in R2010b
**num2deriv**

Numeric two-point network derivative function

**Syntax**

num2deriv('dperf_dwb',net,X,T,Xi,Ai,EW)
num2deriv('de_dwb',net,X,T,Xi,Ai,EW)

**Description**

This function calculates derivatives using the two-point numeric derivative rule.

\[
\frac{dy}{dx} = \frac{y(x + dx) - y(x)}{dx}
\]

This function is much slower than the analytical (non-numerical) derivative functions, but is provided as a means of checking the analytical derivative functions. The other numerical function, num5deriv, is slower but more accurate.

num2deriv('dperf_dwb',net,X,T,Xi,Ai,EW) takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)</td>
</tr>
<tr>
<td>T</td>
<td>Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay states (optional)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states (optional)</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>

and returns the gradient of performance with respect to the network’s weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).
num2deriv('de_dwb',net,X,T,Xi,Ai,EW) returns the Jacobian of errors with respect to the network's weights and biases.

**Examples**

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```matlab
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
dwb = num2deriv('dperf_dwb',net,x,t)
```

**See Also**

bttderiv | defaultderiv | fpderiv | num5deriv | staticderiv

**Introduced in R2010b**
num5deriv

Numeric five-point stencil neural network derivative function

Syntax

num5deriv('dperf_dwb', net, X, T, Xi, Ai, EW)
num5deriv('de_dwb', net, X, T, Xi, Ai, EW)

Description

This function calculates derivatives using the five-point numeric derivative rule.

\[
\begin{align*}
    y_1 &= y(x + 2dx) \\
    y_2 &= y(x + dx) \\
    y_3 &= y(x - dx) \\
    y_4 &= y(x - 2dx) \\
    \frac{dy}{dx} &= \frac{-y_1 + 8y_2 - 8y_3 + y_4}{12dx}
\end{align*}
\]

This function is much slower than the analytical (non-numerical) derivative functions, but is provided as a means of checking the analytical derivative functions. The other numerical function, num2deriv, is faster but less accurate.

num5deriv('dperf_dwb', net, X, T, Xi, Ai, EW) takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)</td>
</tr>
<tr>
<td>T</td>
<td>Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay states (optional)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states (optional)</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>
and returns the gradient of performance with respect to the network’s weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).

\[
\text{num5deriv('de_dwb',net,X,T,Xi,Ai,EW)}
\]

returns the Jacobian of errors with respect to the network’s weights and biases.

**Examples**

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

```plaintext
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
dwb = num5deriv('dperf_dwb',net,x,t)
```

**See Also**

bttderiv | defaultderiv | fpderiv | num2deriv | staticderiv

*Introduced in R2010b*
**numelements**

Number of elements in neural network data

**Syntax**

```plaintext
numelements(x)
```

**Description**

`numelements(x)` takes neural network data `x` in matrix or cell array form, and returns the number of elements in each signal.

If `x` is a matrix the result is the number of rows of `x`.

If `x` is a cell array the result is an `S`-by-1 vector, where `S` is the number of signals (i.e., rows of `X`), and each element `S(i)` is the number of elements in each signal `i` (i.e., rows of `x{i,1}`).

**Examples**

This code calculates the number of elements represented by matrix data:

```plaintext
x = [1 2 3; 4 7 4]
n = numelements(x)
```

This code calculates the number of elements represented by cell data:

```plaintext
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numelements(x)
```

**See Also**

`catelements` | `getelements` | `nndata` | `nnsize` | `numsamples` | `numsignals` | `numtimesteps` | `setelements`
Introduced in R2010b
**numfinite**

Number of finite values in neural network data

**Syntax**

`numfinite(x)`

**Description**

`numfinite(x)` takes a matrix or cell array of matrices and returns the number of finite elements in it.

**Examples**

```matlab
x = [1 2; 3 NaN]
n = numfinite(x)
```

```matlab
x = {{[1 2; 3 NaN] [5 NaN; NaN 8]}}
n = numfinite(x)
```

**See Also**

`nndata` | `nnsize` | `numnan`

**Introduced in R2010b**
numnan
Number of NaN values in neural network data

Syntax
numnan(x)

Description
numnan(x) takes a matrix or cell array of matrices and returns the number of NaN elements in it.

Examples
x = [1 2; 3 NaN]
   n = numnan(x)

x = {{[1 2; 3 NaN] [5 NaN; NaN 8]}
   n = numnan(x)

See Also
nndata | nnsize | numnan

Introduced in R2010b
numsamples

Number of samples in neural network data

**Syntax**

(numsamples(x)

**Description**

(numsamples(x) takes neural network data x in matrix or cell array form, and returns the number of samples.

If x is a matrix, the result is the number of columns of x.

If x is a cell array, the result is the number of columns of the matrices in x.

**Examples**

This code calculates the number of samples represented by matrix data:

```
x = [1 2 3; 4 7 4]
n = numsamples(x)
```

This code calculates the number of samples represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numsamples(x)
```

**See Also**

catsamples|getsamples|nndata|nnsize|numelements|nums Signals|numtimesteps|setsamples

**Introduced in R2010b**
numsignals

Number of signals in neural network data

Syntax

numsignals(x)

Description

numsignals(x) takes neural network data x in matrix or cell array form, and returns the number of signals.

If x is a matrix, the result is 1.

If x is a cell array, the result is the number of rows in x.

Examples

This code calculates the number of signals represented by matrix data:

```matlab
x = [1 2 3; 4 7 4]
n = numsignals(x)
```

This code calculates the number of signals represented by cell data:

```matlab
x = {{[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}}
n = numsignals(x)
```

See Also

catsignals | getsignals | nndata | nnsizend | numelements | numsamples | numtimesteps | setsignals

Introduced in R2010b
numtimesteps

Number of time steps in neural network data

Syntax

numtimesteps(x)

Description

numtimesteps(x) takes neural network data x in matrix or cell array form, and returns the number of signals.

If x is a matrix, the result is 1.

If x is a cell array, the result is the number of columns in x.

Examples

This code calculates the number of time steps represented by matrix data:

```matlab
x = [1 2 3; 4 7 4]
n = numtimesteps(x)
```

This code calculates the number of time steps represented by cell data:

```matlab
x = {{1:3; 4:6} [7:9; 10:12]; [13:15] [16:18]}
n = numtimesteps(x)
```

See Also

cattimesteps | gettimesteps | nndata | nnsise | numelements | numsamples | numsignals | settimesteps

Introduced in R2010b
**openloop**

Convert neural network closed-loop feedback to open loop

**Syntax**

```matlab
net = openloop(net)
[net,xi,ai] = openloop(net,xi,ai)
```

**Description**

`net = openloop(net)` takes a neural network and opens any closed-loop feedback. For each feedback output `i` whose property `net.outputs{i}.feedbackMode` is 'closed', it replaces its associated feedback layer weights with a new input and input weight connections. The `net.outputs{i}.feedbackMode` property is set to 'open', and the `net.outputs{i}.feedbackInput` property is set to the index of the new input. Finally, the value of `net.outputs{i}.feedbackDelays` is subtracted from the delays of the feedback input weights (i.e., to the delays values of the replaced layer weights).

`[net,xi,ai] = openloop(net,xi,ai)` converts a closed-loop network and its current input delay states `xi` and layer delay states `ai` to open-loop form.

**Examples**

**Convert NARX Network to Open-Loop Form**

Here a NARX network is designed in open-loop form and then converted to closed-loop form, then converted back.

```matlab
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,10);
[Xs,Xi,Al,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Al);
view(net)
Yopen = net(Xs,Xi,Al)
```
net = closeloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yclosed = net(Xs,Xi,Ai);
net = openloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yopen = net(Xs,Xi,Ai)

Convert Delay States

For examples on using closeloop and openloop to implement multistep prediction, see narxnet and narnet.

See Also
closeloop | narnet | narxnet | noloop

Introduced in R2010b
patternnet

Pattern recognition network

Syntax

patternnet(hiddenSizes,trainFcn,performFcn)

Description

Pattern recognition networks are feedforward networks that can be trained to classify inputs according to target classes. The target data for pattern recognition networks should consist of vectors of all zero values except for a 1 in element \( i \), where \( i \) is the class they are to represent.

patternnet(hiddenSizes,trainFcn,performFcn) takes these arguments,

<table>
<thead>
<tr>
<th>hiddenSizes</th>
<th>Row vector of one or more hidden layer sizes (default = 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainFcn</td>
<td>Training function (default = 'trainscg')</td>
</tr>
<tr>
<td>performFcn</td>
<td>Performance function (default = 'crossentropy')</td>
</tr>
</tbody>
</table>

and returns a pattern recognition neural network.

Examples

Pattern Recognition

This example shows how to design a pattern recognition network to classify iris flowers.

[x,t] = iris_dataset;
net = patternnet(10);
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,t,y);
classes = vec2ind(y);
**perceptron**

Perceptron

**Syntax**

```
perceptron(hardlimitTF,perceptronLF)
```

**Description**

Perceptrons are simple single-layer binary classifiers, which divide the input space with a linear decision boundary.

Perceptrons can learn to solve a narrow range of classification problems. They were one of the first neural networks to reliably solve a given class of problem, and their advantage is a simple learning rule.

```
perceptron(hardlimitTF,perceptronLF)
```

takes these arguments,

<table>
<thead>
<tr>
<th>hardlimitTF</th>
<th>Hard limit transfer function (default = 'hardlim')</th>
</tr>
</thead>
<tbody>
<tr>
<td>perceptronLF</td>
<td>Perceptron learning rule (default = 'learnp')</td>
</tr>
</tbody>
</table>

and returns a perceptron.

In addition to the default hard limit transfer function, perceptrons can be created with the hardlims transfer function. The other option for the perceptron learning rule is learnpn.

**Note** Deep Learning Toolbox supports perceptrons for historical interest. For better results, you should instead use `patternnet`, which can solve nonlinearily separable problems. Sometimes the term “perceptrons” refers to feed-forward pattern recognition networks; but the original perceptron, described here, can solve only simple problems.
Examples

Solve Simple Classification Problem Using Perceptron

Use a perceptron to solve a simple classification logical-OR problem.

\[
x = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}; \\
t = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}; \\
net = \text{perceptron}; \\
net = \text{train}(net,x,t); \\
\text{view}(net) \\
y = \text{net}(x);
\]

See Also
narnet | narxnet | patternnet | prepares | removedelay | timedelaynet

Introduced in R2010b
**perform**

Calculate network performance

**Syntax**

`perform(net,t,y,ew)`

**Description**

`perform(net,t,y,ew)` takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Target data</td>
</tr>
<tr>
<td>y</td>
<td>Output data</td>
</tr>
<tr>
<td>ew</td>
<td>Error weights (default = <code>{1}</code>)</td>
</tr>
</tbody>
</table>

and returns network performance calculated according to the `net.performFcn` and `net.performParam` property values.

The target and output data must have the same dimensions. The error weights may be the same dimensions as the targets, in the most general case, but may also have any of its dimensions be 1. This gives the flexibility of defining error weights across any dimension desired.

Error weights can be defined by sample, output element, time step, or network output:

- `ew = [1.0 0.5 0.7 0.2];` % Across 4 samples
- `ew = [0.1; 0.5; 1.0];` % Across 3 elements
- `ew = {0.1 0.2 0.3 0.5 1.0};` % Across 5 timesteps
- `ew = {1.0; 0.5};` % Across 2 outputs

The error weights can also be defined across any combination, such as across two time-series (i.e., two samples) over four timesteps.

- `ew = {[0.5 0.4],[0.3 0.5],[1.0 1.0],[0.7 0.5]};`
In the general case, error weights may have exactly the same dimensions as targets, in which case each target value will have an associated error weight.

The default error weight treats all errors the same.

\[ ew = \{1\} \]

**Examples**

Here a simple fitting problem is solved with a feed-forward network and its performance calculated.

```matlab
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y)

perf =

2.3654e-06
```

**See Also**

configure | init | train

**Introduced in R2010b**
plotconfusion

Plot classification confusion matrix

Syntax

plotconfusion(targets,outputs)
plotconfusion(targets,outputs,name)
plotconfusion(targets1,outputs1,name1,targets2,outputs2,name2,...,targetsn,outputsn,namen)

Description

plotconfusion(targets,outputs) plots a confusion matrix for the true labels targets and predicted labels outputs. Specify the labels as categorical vectors, or in one-of-N (one-hot) form.

On the confusion matrix plot, the rows correspond to the predicted class (Output Class) and the columns correspond to the true class (Target Class). The diagonal cells correspond to observations that are correctly classified. The off-diagonal cells correspond to incorrectly classified observations. Both the number of observations and the percentage of the total number of observations are shown in each cell.

The column on the far right of the plot shows the percentages of all the examples predicted to belong to each class that are correctly and incorrectly classified. These metrics are often called the precision (or positive predictive value) and false discovery rate, respectively. The row at the bottom of the plot shows the percentages of all the examples belonging to each class that are correctly and incorrectly classified. These metrics are often called the recall (or true positive rate) and false negative rate, respectively. The cell in the bottom right of the plot shows the overall accuracy.

plotconfusion(targets,outputs,name) plots a confusion matrix and adds name to the beginning of the plot title.

plotconfusion(targets1,outputs1,name1,targets2,outputs2,name2,...,targetsn,outputsn,namen) plots multiple confusion matrices in one figure and adds the name arguments to the beginnings of the titles of the corresponding plots.
Examples

Plot Confusion Matrix Using Categorical Labels

Load the data consisting of synthetic images of handwritten digits. XTrain is a 28-by-28-by-1-by-5000 array of images and YTrain is a categorical vector containing the image labels.

[XTrain,YTrain] = digitTrain4DArrayData;
whos YTrain

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>YTrain</td>
<td>5000x1</td>
<td>6142</td>
<td>categorical</td>
<td></td>
</tr>
</tbody>
</table>

Define the architecture of a convolutional neural network.

layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,16,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same','Stride',2)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

Specify training options and train the network.

options = trainingOptions('sgdm',...'
    'MaxEpochs',5,...
    'Verbose',false,...
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,layers,options);
Load and classify test data using the trained network.

[XTest,YTest] = digitTest4DArrayData;
YPredicted = classify(net,XTest);

Plot the confusion matrix of the true test labels YTest and the predicted labels YPredicted.

plotconfusion(YTest,YPredicted)
The rows correspond to the predicted class (Output Class) and the columns correspond to the true class (Target Class). The diagonal cells correspond to observations that are correctly classified. The off-diagonal cells correspond to incorrectly classified observations. Both the number of observations and the percentage of the total number of observations are shown in each cell.

The column on the far right of the plot shows the percentages of all the examples predicted to belong to each class that are correctly and incorrectly classified. These metrics are often called the precision (or positive predictive value) and false discovery rate, respectively. The row at the bottom of the plot shows the percentages of all the examples belonging to each class that are correctly and incorrectly classified. These metrics are often called the recall (or true positive rate) and false negative rate, respectively. The cell in the bottom right of the plot shows the overall accuracy.

**Plot Confusion Matrix Using One-of-N Labels**

Load sample data using the `cancer_dataset` function. `XTrain` is a 9-by-699 matrix defining nine attributes of 699 biopsies. `YTrain` is a 2-by-699 matrix where each column indicates the correct category of the corresponding observation. Each column of `YTrain` has one element that equals one in either the first or second row, corresponding to the cancer being benign or malignant, respectively. For more information on this dataset, type `help cancer_dataset` at the command line.

```matlab
rng default
[XTrain,YTrain] = cancer_dataset;
YTrain(:,1:10)
ans = 2×10
     1     1     1     0     1     1     0     0     0     1
     0     0     0     1     0     0     1     1     1     0
```

Create a pattern recognition network and train it using the sample data.

```matlab
net = patternnet(10);
net = train(net,XTrain,YTrain);
```

Estimate the cancer status using the trained network. Each column of the matrix `YPredicted` contains the predicted probabilities of each observation belonging to class 1 and class 2, respectively.
YPredicted = net(XTrain);
YPredicted(:,1:10)

ans = 2×10

0.9999    0.9999    0.9999    0.0578    0.9993    0.9999    0.0012    0.0001    0.0028    0.9999
0.0001    0.0001    0.0001    0.9422    0.0007    0.0001    0.9988    0.9999    0.0001   

Plot the confusion matrix. To create the plot, plotconfusion labels each observation according to the highest class probability.

plotconfusion(YTrain,YPredicted)
In this figure, the first two diagonal cells show the number and percentage of correct classifications by the trained network. For example, 446 biopsies are correctly classified as benign. This corresponds to 63.8% of all 699 biopsies. Similarly, 236 cases are correctly classified as malignant. This corresponds to 33.8% of all biopsies.

5 of the malignant biopsies are incorrectly classified as benign and this corresponds to 0.7% of all 699 biopsies in the data. Similarly, 12 of the benign biopsies are incorrectly classified as malignant and this corresponds to 1.7% of all data.

Out of 451 benign predictions, 98.9% are correct and 1.1% are wrong. Out of 248 malignant predictions, 95.2% are correct and 4.8% are wrong. Out of 458 benign cases, 97.4% are correctly predicted as benign and 2.6% are predicted as malignant. Out of 241 malignant cases, 97.9% are correctly classified as malignant and 2.1% are classified as benign.

Overall, 97.6% of the predictions are correct and 2.4% are wrong.

**Input Arguments**

**targets — True class labels**  
categorical vector | matrix

True class labels, specified one of the following:

- A categorical vector, where each element is the class label of one observation. The outputs and targets arguments must have the same number of elements. If the categorical vectors define underlying classes, then plotconfusion displays all the underlying classes, even if there are no observations of some of the underlying classes. If the arguments are ordinal categorical vectors, then they must both define the same underlying categories, in the same order.

- An N-by-M matrix, where N is the number of classes and M is the number of observations. Each column of the matrix must be in one-of-N (one-hot) form, where a single element equal to 1 indicates the true label and all other elements equal 0.

**outputs — Predicted class labels**  
categorical vector | matrix

Predicted class labels, specified one of the following:
• A categorical vector, where each element is the class label of one observation. The outputs and targets arguments must have the same number of elements. If the categorical vectors define underlying classes, then plotconfusion displays all the underlying classes, even if there are no observations of some of the underlying classes. If the arguments are ordinal categorical vectors, then they must both define the same underlying categories, in the same order.

• An N-by-M matrix, where N is the number of classes and M is the number of observations. Each column of the matrix can be in one-of-N (one-hot) form, where a single element equal to 1 indicates the predicted label, or in the form of probabilities that sum to one.

**name** — Name of the confusion matrix
character array

Name of the confusion matrix, specified as a character array. plotconfusion adds the specified name to the beginning of the plot title.

Data Types: char

**See Also**

trainNetwork | trainingOptions

**Introduced in R2008a**
**plotep**

Plot weight-bias position on error surface

**Syntax**

\[ H = \text{plotep}(W,B,E) \]
\[ H = \text{plotep}(W,B,E,H) \]

**Description**

`plotep` is used to show network learning on a plot created by `plotes`.

\[ H = \text{plotep}(W,B,E) \] takes these arguments,

<table>
<thead>
<tr>
<th>W</th>
<th>Current weight value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Current bias value</td>
</tr>
<tr>
<td>E</td>
<td>Current error</td>
</tr>
</tbody>
</table>

and returns a cell array `H`, containing information for continuing the plot.

\[ H = \text{plotep}(W,B,E,H) \] continues plotting using the cell array `H` returned by the last call to `plotep`.

`H` contains handles to dots plotted on the error surface, so they can be deleted next time; as well as points on the error contour, so they can be connected.

**See Also**

`errsurf` | `plotes`

**Introduced before R2006a**
ploterrcorr

Plot autocorrelation of error time series

Syntax

ploterrcorr(error)
ploterrcorr(errors,'outputIndex',outIdx)

Description

ploterrcorr(error) takes an error time series and plots the autocorrelation of errors across varying lags.

ploterrcorr(errors,'outputIndex',outIdx) uses the optional property name/value pair to define which output error autocorrelation is plotted. The default is 1.

Examples

Plot Autocorrelation of Errors

Here a NARX network is used to solve a time series problem.

[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
E = gsubtract(Ts,Y);
ploterrcorr(E)
See Also
plotinerrcorr | plotresponse

Introduced in R2010b
ploterrhist

Plot error histogram

Syntax

ploterrhist(e)
ploterrhist(e1,'name1',e2,'name2',...)
ploterrhist(...,'bins',bins)

Description

ploterrhist(e) plots a histogram of error values e.

ploterrhist(e1,'name1',e2,'name2',...) takes any number of errors and names and plots each pair.

ploterrhist(...,'bins',bins) takes an optional property name/value pair which defines the number of bins to use in the histogram plot. The default is 20.

Examples

Plot Histogram of Error Values

Here a feedforward network is used to solve a simple fitting problem:

[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
e = t - y;
ploterrhist(e,'bins',30)
See Also
plotconfusion | ploterrcorr | plotinerrcorr

Introduced in R2010b
**plotes**

Plot error surface of single-input neuron

**Syntax**

```
plotes(WV,BV,ES,V)
```

**Description**

`plotes(WV,BV,ES,V)` takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WV</td>
<td>1-by-N row vector of values of W</td>
</tr>
<tr>
<td>BV</td>
<td>1-by-M row vector of values of B</td>
</tr>
<tr>
<td>ES</td>
<td>M-by-N matrix of error vectors</td>
</tr>
<tr>
<td>V</td>
<td>View (default = [-37.5, 30])</td>
</tr>
</tbody>
</table>

and plots the error surface with a contour underneath.

Calculate the error surface ES with `errsurf`.

**Examples**

**Plot Error Surface of Single-Input Neuron**

```matlab
p = [3 2];
t = [0.4 0.8];
wv = -4:0.4:4;
bv = wv;
ES = errsurf(p,t,wv,bv,'logsig');
plotes(wv,bv,ES,[60 30])
```
See Also

errsrf

Introduced before R2006a
plotfit

Plot function fit

Syntax

plotfit(net,inputs,targets)
plotfit(targets1,inputs1,'name1',...)

Description

plotfit(net,inputs,targets) plots the output function of a network across the range of the inputs inputs and also plots target targets and output data points associated with values in inputs. Error bars show the difference between outputs and targets.

The plot appears only for networks with one input.

Only the first output/targets appear if the network has more than one output.

plotfit(targets1,inputs1,'name1',...) displays a series of plots.

Examples

Plot Output and Target Values

This example shows how to use a feed-forward network to solve a simple fitting problem.

[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
plotfit(net,x,t)
See Also

plottrainstate

Introduced in R2008a
**plotinerrcorr**

Plot input to error time-series cross-correlation

**Syntax**

plotinerrcorr(x,e)
plotinerrcorr(...,'inputIndex',inputIndex)
plotinerrcorr(...,'outputIndex',outputIndex)

**Description**

plotinerrcorr(x,e) takes an input time series x and an error time series e, and plots the cross-correlation of inputs to errors across varying lags.

plotinerrcorr(...,'inputIndex',inputIndex) optionally defines which input element is being correlated and plotted. The default is 1.

plotinerrcorr(...,'outputIndex',outputIndex) optionally defines which error element is being correlated and plotted. The default is 1.

**Examples**

**Plot Cross-Correlation of Inputs to Errors**

Here a NARX network is used to solve a time series problem.

```matlab
[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
E = gsubtract(Ts,Y);
plotinerrcorr(Xs,E)
```
See Also
ploterrcorr | ploterrhist | plotresponse

Introduced in R2010b
plotpc

Plot classification line on perceptron vector plot

**Syntax**

plotpc(W,B)
plotpc(W,B,H)

**Description**

plotpc(W,B) takes these inputs,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>S-by-R weight matrix (R must be 3 or less)</td>
</tr>
<tr>
<td>B</td>
<td>S-by-1 bias vector</td>
</tr>
</tbody>
</table>

and returns a handle to a plotted classification line.

plotpc(W,B,H) takes an additional input,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>Handle to last plotted line</td>
</tr>
</tbody>
</table>

and deletes the last line before plotting the new one.

This function does not change the current axis and is intended to be called after plotpv.

**Examples**

**Plot Classification Line**

The code below defines and plots the inputs and targets for a perceptron:

```
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```
The following code creates a perceptron, assigns values to its weights and biases, and plots the resulting classification line.

```matlab
net = perceptron;
net = configure(net,p,t);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```
See Also

plotpv

Introduced before R2006a
plotperform

Plot network performance

Syntax

plotperform(TR)

Description

plotperform(TR) plots error vs. epoch for the training, validation, and test performances of the training record TR returned by the function train.

Examples

Plot Validation Performance of Network

This example shows how to use plotperform to obtain a plot of training record error values against the number of training epochs.

[x,t] = bodyfat_dataset;
net = feedforwardnet(10);
[net,tr] = train(net,x,t);
plotperform(tr)
Generally, the error reduces after more epochs of training, but might start to increase on the validation data set as the network starts overfitting the training data. In the default setup, the training stops after six consecutive increases in validation error, and the best performance is taken from the epoch with the lowest validation error.

See Also
plottrainstate

Introduced in R2008a
**plotpv**

Plot perceptron input/target vectors

**Syntax**

```matlab
plotpv(P,T)
plotpv(P,T,V)
```

**Description**

`plotpv(P,T)` takes these inputs,

<table>
<thead>
<tr>
<th>P</th>
<th>R-by-Q matrix of input vectors (R must be 3 or less)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>S-by-Q matrix of binary target vectors (S must be 3 or less)</td>
</tr>
</tbody>
</table>

and plots column vectors in `P` with markers based on `T`.

`plotpv(P,T,V)` takes an additional input,

```
V                         Graph limits = [x_min x_max y_min y_max]
```

and plots the column vectors with limits set by `V`.

**Examples**

**Plot Inputs and Targets for Perceptron**

This example shows how to define and plot the inputs and targets for a perceptron.

```matlab
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
plotpv(p,t)
```
See Also
plotpc

Introduced before R2006a
**plotregression**

Plot linear regression

**Syntax**

```plaintext
plotregression(targets,outputs)
plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)
```

**Description**

`plotregression(targets,outputs)` plots the linear regression of targets relative to outputs.

`plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)` generates multiple plots.

**Examples**

**Plot Linear Regression**

```plaintext
[x,t] = simplefit_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
y = net(x);
plotregression(t,y,'Regression')
```
See Also

plottrainstate

Introduced in R2008a
plotresponse

Plot dynamic network time series response

Syntax

plotresponse(t,y)
plotresponse(t1,'name',t2,'name2',...,y)
plotresponse(...,'outputIndex',outputIndex)

Description

plotresponse(t,y) takes a target time series t and an output time series y, and plots them on the same axis showing the errors between them.

plotresponse(t1,'name',t2,'name2',...,y) takes multiple target/name pairs, typically defining training, validation and testing targets, and the output. It plots the responses with colors indicating the different target sets.

plotresponse(...,'outputIndex',outputIndex) optionally defines which error element is being correlated and plotted. The default is 1.

Examples

Plot Target and Output Time Series Data

This example shows how to use a NARX network to solve a time series problem.

[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparesets(net,X,{},{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
plotresponse(Ts,Y)
See Also
ploterrcorr | ploterrhist | plotinerrcorr

Introduced in R2010b
plotroc

Plot receiver operating characteristic

Syntax

plotroc(targets,outputs)
plotroc(targets1,outputs2,'name1',...)

Description

plotroc(targets,outputs) plots the receiver operating characteristic for each output class. The more each curve hugs the left and top edges of the plot, the better the classification.

plotroc(targets1,outputs2,'name1',...) generates multiple plots.

Examples

Plot Receiver Operating Characteristic

load simplecluster_dataset
net = patternnet(20);
net = train(net,simpleclusterInputs,simpleclusterTargets);
simpleclusterOutputs = sim(net,simpleclusterInputs);
plotroc(simpleclusterTargets,simpleclusterOutputs)
See Also

roc

Introduced in R2008a
**plotsom**

Plot self-organizing map

**Syntax**

```
plotsom(pos)
plotsom(W,D,ND)
```

**Description**

`plotsom(pos)` takes one argument,

<table>
<thead>
<tr>
<th><strong>POS</strong></th>
<th>N-by-S matrix of S N-dimension neural positions</th>
</tr>
</thead>
</table>

and plots the neuron positions with red dots, linking the neurons within a Euclidean distance of 1.

`plotsom(W,D,ND)` takes three arguments,

<table>
<thead>
<tr>
<th><strong>W</strong></th>
<th>S-by-R weight matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D</strong></td>
<td>S-by-S distance matrix</td>
</tr>
<tr>
<td><strong>ND</strong></td>
<td>Neighborhood distance (default = 1)</td>
</tr>
</tbody>
</table>

and plots the neuron’s weight vectors with connections between weight vectors whose neurons are within a distance of 1.

**Examples**

**Plot Self-Organizing Maps**

These examples generate plots of various layer topologies.
pos = hextop([5 6]);
plotsom(pos)

pos = gridtop([4 5]);
plotsom(pos)
pos = randtop([18 12]);
plotsom(pos)
pos = gridtop([4 5 2]);
plotsom(pos)
pos = hextop([4 4 3]);
plotsom(pos)
See plotsompos for an example of plotting a layer’s weight vectors with the input vectors they map.

**See Also**

learnsom

*Introduced before R2006a*
plotsomhits

Plot self-organizing map sample hits

Syntax

plotsomhits(net,inputs)

Description

plotsomhits(net,inputs) plots a SOM layer, with each neuron showing the number of input vectors that it classifies. The relative number of vectors for each neuron is shown via the size of a colored patch.

This plot supports SOM networks with hextop and gridtop topologies, but not tritop or randtop.

Examples

Plot SOM Sample Hits

```markdown
x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomhits(net,x)
```
See Also
plotsomplanes

Introduced in R2008a
plotsomnc

Plot self-organizing map neighbor connections

Syntax

plotsomnc(net)

Description

plotsomnc(net) plots a SOM layer showing neurons as gray-blue patches and their direct neighbor relations with red lines.

This plot supports SOM networks with hextop and gridtop topologies, but not tritop or randtop.

Examples

Plot SOM Neighbor Connections

x = iris_dataset;
net = selforgmap([8 8]);
net = train(net,x);
plotsomnc(net)
See Also
plotsomhits | plotsomnd | plotsomplanes

Introduced in R2008a
plotsomnd

Plot self-organizing map neighbor distances

Syntax

plotsomnd(net)

Description

plotsomnd(net) plots a SOM layer showing neurons as gray-blue patches and their direct neighbor relations with red lines. The neighbor patches are colored from black to yellow to show how close each neuron’s weight vector is to its neighbors.

This plot supports SOM networks with hextop and gridtop topologies, but not tritop or randtop.

Examples

Plot SOM Neighbor Distances

x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomnd(net)
See Also
plotsomhits|plotsomnc|plotsomplanes

Introduced in R2008a
plotsomplanes

Plot self-organizing map weight planes

Syntax

plotsomplanes(net)

Description

plotsomplanes(net) generates a set of subplots. Each ith subplot shows the weights from the ith input to the layer’s neurons, with the most negative connections shown as black, zero connections as red, and the strongest positive connections as yellow.

The plot is only shown for layers organized in one or two dimensions.

This plot supports SOM networks with hextop and gridtop topologies, but not tritop or randtop.

This function can also be called with standardized plotting function arguments used by the function train.

Examples

Plot SOM Weight Planes

```matlab
x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomplanes(net)
```
See Also
plotsomhits | plotsomnc | plotsomnd

Introduced in R2008a
**plotsompos**

Plot self-organizing map weight positions

**Syntax**

plotsompos(net)
plotsompos(net,inputs)

**Description**

plotsompos(net) plots the input vectors as green dots and shows how the SOM classifies the input space by showing blue-gray dots for each neuron’s weight vector and connecting neighboring neurons with red lines.

plotsompos(net,inputs) plots the input data alongside the weights.

**Examples**

**Plot SOM Weight Positions**

```matlab
x = iris_dataset;
net = selforgmap([10 10]);
net = train(net,x);
plotsompos(net,x)
```
See Also
plotsomhits | plotsomnd | plotsomplanes

Introduced in R2008a
plotsomtop

Plot self-organizing map topology

Syntax

plotsomtop(net)

Description

plotsomtop(net) plots the topology of a SOM layer.

This plot supports SOM networks with hextop and gridtop topologies, but not tritop or randtop.

Examples

Plot SOM Topology

x = iris_dataset;
net = selforgmap([8 8]);
plotsomtop(net)
See Also
plotsomhits | plotsomnd | plotsomplanes

Introduced in R2008a
plottrainstate

Plot training state values

Syntax

plottrainstate(tr)

Description

plottrainstate(tr) plots the training state from a training record tr returned by train.

Examples

Plot Training State Values

This example shows how to plot training state values using plottrainstate.

[x, t] = bodyfat_dataset;
net = feedforwardnet(10);
[net, tr] = train(net, x, t);
plottrainstate(tr)
See Also
plotfit|plotperform|plotregression

Introduced in R2008a
**plotv**

Plot vectors as lines from origin

**Syntax**

`plotv(M, T)`

**Description**

`plotv(M, T)` takes two inputs,

<table>
<thead>
<tr>
<th>M</th>
<th>R-by-Q matrix of Q column vectors with R elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The line plotting type (optional; default = ' - ')</td>
</tr>
</tbody>
</table>

and plots the column vectors of M.

R must be 2 or greater. If R is greater than 2, only the first two rows of M are used for the plot.

**Examples**

**Plot Vectors**

This example shows how to plot three 2-element vectors.

```matlab
M = [-0.4 0.7 0.2 ; ... 
    -0.5 0.1 0.5];
plotv(M, ' - ')
```
Introduced before R2006a
**plotvec**

Plot vectors with different colors

**Syntax**

plotvec(X,C,M)

**Description**

plotvec(X,C,M) takes these inputs,

<table>
<thead>
<tr>
<th>X</th>
<th>Matrix of (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Row vector of color coordinates</td>
</tr>
<tr>
<td>M</td>
<td>Marker (default = '+' )</td>
</tr>
</tbody>
</table>

and plots each i th vector in X with a marker M, using the i th value in C as the color coordinate.

plotvec(X) only takes a matrix X and plots each i th vector in X with marker '+' using the index i as the color coordinate.

**Examples**

**Plot Vectors with Different Colors**

This example shows how to plot four 2-element vectors.

```matlab
x = [ 0 1 0.5 0.7 ; ... -1 2 0.5 0.1];
c = [1 2 3 4];
plotvec(x,c)
```
Introduced before R2006a
**plotwb**

Plot Hinton diagram of weight and bias values

**Syntax**

plotwb(net)
plotwb(IW,LW,B)
plotwb(...,'toLayers',toLayers)
plotwb(...,'fromInputs',fromInputs)
plotwb(...,'fromLayers',fromLayers)
plotwb(...,'root',root)

**Description**

plotwb(net) takes a neural network and plots all its weights and biases.

plotwb(IW,LW,B) takes a neural networks input weights, layer weights and biases and plots them.

plotwb(...,'toLayers',toLayers) optionally defines which destination layers whose input weights, layer weights and biases will be plotted.

plotwb(...,'fromInputs',fromInputs) optionally defines which inputs will have their weights plotted.

plotwb(...,'fromLayers',fromLayers) optionally defines which layers will have weights coming from them plotted.

plotwb(...,'root',root) optionally defines the root used to scale the weight/bias patch sizes. The default is 2, which makes the 2-dimensional patch sizes scale directly with absolute weight and bias sizes. Larger values of root magnify the relative patch sizes of smaller weights and biases, making differences in smaller values easier to see.
Examples

Plot Weights and Biases

Here a cascade-forward network is configured for particular data and its weights and biases are plotted in several ways.

```matlab
[x,t] = simplefit_dataset;
net = cascadeforwardnet([15 5]);
net = configure(net,x,t);
plotwb(net)
plotwb(net,'root',3)
```

![Plot of weights and biases](image)

```matlab
plotwb(net,'root',3)
```
plotwb(net,'root',4)
plotwb(net,'toLayers',2)
plotwb(net,'fromLayers',1)
plotwb(net,'toLayers',2,'fromInputs',1)
See Also
plotsomplanes

Introduced in R2010b
pnormc

Pseudonormalize columns of matrix

Syntax

pnormc(X,R)

Description

pnormc(X,R) takes these arguments,

<table>
<thead>
<tr>
<th>X</th>
<th>M-by-N matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>(Optional) radius to normalize columns to (default = 1)</td>
</tr>
</tbody>
</table>

and returns X with an additional row of elements, which results in new column vector lengths of R.

Caution For this function to work properly, the columns of X must originally have vector lengths less than R.

Examples

x = [0.1 0.6; 0.3 0.1];
y = pnormc(x)

See Also

normc | normr

Introduced before R2006a
poslin

Positive linear transfer function

Graph and Symbol

![Graph of the positive linear transfer function](image)

Positive Linear Transfer Function

Syntax

\[
A = \text{poslin}(N, \text{FP})
\]

\[
\text{info} = \text{poslin}('\text{code}')
\]

Description

poslin is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[
A = \text{poslin}(N, \text{FP})
\]

takes \(N\) and optional function parameters,

<table>
<thead>
<tr>
<th>(N)</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{FP})</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \(A\), the S-by-Q matrix of \(N\)’s elements clipped to \([0, \infty]\).

\[
\text{info} = \text{poslin}('\text{code}')
\]

returns information about this function. The following codes are supported:
poslin('name') returns the name of this function.

poslin('output', FP) returns the [min max] output range.

poslin('active', FP) returns the [min max] active range.

poslin('fullderiv') returns 1 or 0, depending on whether $dA_dN$ is $S$-by-$S$-by-$Q$ or $S$-by-$Q$.

poslin('fpnames') returns the names of the function parameters.

poslin('fpdefaults') returns the default function parameters.

**Examples**

Here is the code to create a plot of the poslin transfer function.

```matlab
n = -5:0.1:5;
a = poslin(n);
plot(n,a)
```

Assign this transfer function to layer i of a network.

```matlab
net.layers{i}.transferFcn = 'poslin';
```

**Network Use**

To change a network so that a layer uses poslin, set `net.layers{i}.transferFcn` to 'poslin'.

Call `sim` to simulate the network with poslin.

**Algorithms**

The transfer function poslin returns the output $n$ if $n$ is greater than or equal to zero and 0 if $n$ is less than or equal to zero.

$$\text{poslin}(n) = \begin{cases} n, & \text{if } n \geq 0 \\ 0, & \text{if } n \leq 0 \end{cases}$$
See Also
purelin | satlin | satlins | sim

Introduced before R2006a
**preparets**

Prepare input and target time series data for network simulation or training

**Syntax**

\[
[Xs,Xi,Ai,Ts,EWs,shift] = \text{preparets}(\text{net},Xnf,Tnf,Tf,EW)
\]

**Description**

This function simplifies the normally complex and error prone task of reformatting input and target time series. It automatically shifts input and target time series as many steps as are needed to fill the initial input and layer delay states. If the network has open-loop feedback, then it copies feedback targets into the inputs as needed to define the open-loop inputs.

Each time a new network is designed, with different numbers of delays or feedback settings, preparets can reformat input and target data accordingly. Also, each time a network is transformed with openloop, closeloop, removedelay or addedelay, this function can reformat the data accordingly.

\[
[Xs,Xi,Ai,Ts,EWs,shift] = \text{preparets}(\text{net},Xnf,Tnf,Tf,EW)
\]

takes these arguments,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xnf</td>
<td>Non-feedback inputs</td>
</tr>
<tr>
<td>Tnf</td>
<td>Non-feedback targets</td>
</tr>
<tr>
<td>Tf</td>
<td>Feedback targets</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (default = {1})</td>
</tr>
</tbody>
</table>

and returns,

<table>
<thead>
<tr>
<th>Xs</th>
<th>Shifted inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xi</td>
<td>Initial input delay states</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states</td>
</tr>
<tr>
<td>Ts</td>
<td>Shifted targets</td>
</tr>
<tr>
<td>EWs</td>
<td>Shifted error weights</td>
</tr>
<tr>
<td>shift</td>
<td>The number of timesteps truncated from the front of (X) and (T) in order to properly fill (Xi) and (Ai).</td>
</tr>
</tbody>
</table>

## Examples

### Prepare Data for Open- and Closed-Loop Networks

Here a time-delay network with 20 hidden neurons is created, trained and simulated.

```matlab
[X,T] = simpleseries_dataset;
net = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts);
view(net)
Y = net(Xs,Xi,Ai);
```

Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```matlab
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
y = net(Xs,Xi,Ai);
```
Now the network is converted to closed loop, and the data is reformatted to simulate the network’s closed-loop response.

```matlab
net = closeloop(net);
view(net)
[Xs,Xi,Ai] = preparets(net,X,{},T);
y = net(Xs,Xi,Ai);
```

**See Also**
adddelay | closeloop | narnet | narxnet | openloop | removedelay | timedelaynet
Introduced in R2010b
**processpca**

Process columns of matrix with principal component analysis

**Syntax**

\[[Y,PS] = \text{processpca}(X,\text{maxfrac})\]  
\[[Y,PS] = \text{processpca}(X,\text{FP})\]  
\[Y = \text{processpca}('apply',X,PS)\]  
\[X = \text{processpca}('reverse',Y,PS)\]  
\[\text{name} = \text{processpca}('\text{name}')\]  
\[fp = \text{processpca}('\text{pdefaults}')\]  
\[\text{names} = \text{processpca}('\text{pdesc}')\]  
\[\text{processpca}('\text{pcheck}',fp);\]

**Description**

`processpca` processes matrices using principal component analysis so that each row is uncorrelated, the rows are in the order of the amount they contribute to total variation, and rows whose contribution to total variation are less than `maxfrac` are removed.

\[[Y,PS] = \text{processpca}(X,\text{maxfrac})\] takes `X` and an optional parameter,

<table>
<thead>
<tr>
<th>X</th>
<th>N-by-Q matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxfrac</td>
<td>Maximum fraction of variance for removed rows (default is 0)</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Y</th>
<th>M-by-Q matrix with N - M rows deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>Process settings that allow consistent processing of values</td>
</tr>
</tbody>
</table>

\[[Y,PS] = \text{processpca}(X,FP)\] takes parameters as a struct: `FP.maxfrac`.

\[Y = \text{processpca}('apply',X,PS)\] returns `Y`, given `X` and settings `PS`.

\[X = \text{processpca}('reverse',Y,PS)\] returns `X`, given `Y` and settings `PS`. 

1-369
name = processpca('name') returns the name of this process method.

fp = processpca('pdefaults') returns default process parameter structure.

names = processpca('pdesc') returns the process parameter descriptions.

processpca('pcheck',fp); throws an error if any parameter is illegal.

**Examples**

Here is how to format a matrix with an independent row, a correlated row, and a completely redundant row so that its rows are uncorrelated and the redundant row is dropped.

```matlab
x1_independent = rand(1,5)
x1_correlated = rand(1,5) + x1_independent;
x1_redundant = x1_independent + x1_correlated
x1 = [x1_independent; x1_correlated; x1_redundant]
[y1,ps] = processpca(x1)
```

Next, apply the same processing settings to new values.

```matlab
x2_independent = rand(1,5)
x2_correlated = rand(1,5) + x1_independent;
x2_redundant = x1_independent + x1_correlated
x2 = [x2_independent; x2_correlated; x2_redundant];
y2 = processpca('apply',x2,ps)
```

Reverse the processing of y1 to get x1 again.

```matlab
x1_again = processpca('reverse',y1,ps)
```

**More About**

**Reduce Input Dimensionality Using processpca**

In some situations, the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis. This technique has three effects: it orthogonalizes the
components of the input vectors (so that they are uncorrelated with each other), it orders
the resulting orthogonal components (principal components) so that those with the
largest variation come first, and it eliminates those components that contribute the least
to the variation in the data set. The following code illustrates the use of `processpca`,
which performs a principal-component analysis using the processing setting `maxfrac` of
0.02.

```matlab
[pn,ps1] = mapstd(p);
[ptrans,ps2] = processpca(pn,0.02);
```

The input vectors are first normalized, using `mapstd`, so that they have zero mean and
unity variance. This is a standard procedure when using principal components. In this
example, the second argument passed to `processpca` is 0.02. This means that
`processpca` eliminates those principal components that contribute less than 2% to the
total variation in the data set. The matrix `ptrans` contains the transformed input vectors.
The settings structure `ps2` contains the principal component transformation matrix. After
the network has been trained, these settings should be used to transform any future
inputs that are applied to the network. It effectively becomes a part of the network, just
like the network weights and biases. If you multiply the normalized input vectors `pn` by
the transformation matrix `transMat`, you obtain the transformed input vectors `ptrans`.

If `processpca` is used to preprocess the training set data, then whenever the trained
network is used with new inputs, you should preprocess them with the transformation
matrix that was computed for the training set, using `ps2`. The following code applies a
new set of inputs to a network already trained.

```matlab
pnewn = mapstd('apply',pnew,ps1);
pnewtrans = processpca('apply',pnewn,ps2);
a = sim(net,pnewtrans);
```

Principal component analysis is not reliably reversible. Therefore it is only recommended
for input processing. Outputs require reversible processing functions.

Principal component analysis is not part of the default processing for `feedforwardnet`.
You can add this with the following command:

```matlab
net.inputs{1}.processFcns{end+1} = 'processpca';
```

**Algorithms**

Values in rows whose elements are not all the same value are set to
\[ y = 2\frac{(x - \text{minx})}{(\text{maxx} - \text{minx})} - 1; \]

Values in rows with all the same value are set to 0.

**See Also**

fixunknowns | mapminmax | mapstd

**Introduced in R2006a**
**prune**

Delete neural inputs, layers, and outputs with sizes of zero

**Syntax**

\[ [\text{net}, \text{pi}, \text{pl}, \text{po}] = \text{prune}(\text{net}) \]

**Description**

This function removes zero-sized inputs, layers, and outputs from a network. This leaves a network which may have fewer inputs and outputs, but which implements the same operations, as zero-sized inputs and outputs do not convey any information.

One use for this simplification is to prepare a network with zero sized subobjects for Simulink, where zero sized signals are not supported.

The companion function **prunedata** can prune data to remain consistent with the transformed network.

\[ [\text{net}, \text{pi}, \text{pl}, \text{po}] = \text{prune}(\text{net}) \] takes a neural network and returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>The same network with zero-sized subobjects removed</td>
</tr>
<tr>
<td>pi</td>
<td>Indices of pruned inputs</td>
</tr>
<tr>
<td>pl</td>
<td>Indices of pruned layers</td>
</tr>
<tr>
<td>po</td>
<td>Indices of pruned outputs</td>
</tr>
</tbody>
</table>

**Examples**

Here a NARX dynamic network is created which has one external input and a second input which feeds back from the output.

```matlab
net = narxnet(20);
view(net)
```
The network is then trained on a single random time-series problem with 50 timesteps. The external input happens to have no elements.

```matlab
X = nndata(0,1,50);
T = nndata(1,1,50);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts);
```

The network and data are then pruned before generating a Simulink diagram and initializing its input and layer states.

```matlab
[net2,pi,pl,po] = prune(net);
view(net)
[Xs2,Xi2,Ai2,Ts2] = prunedata(net,pi,pl,po,Xs,Xi,Ai,Ts)
[sysName,netName] = gensim(net);
setsiminit(sysName,netName,Xi2,Ai2)
```

**See Also**
gensim | prunedata

*Introduced in R2010b*
prunedata

Prune data for consistency with pruned network

Syntax

```matlab
[Xp,Xip,Aip,Tp] = prunedata(pi,pl,po,X,Xi,Ai,T)
```

Description

This function prunes data to be consistent with a network whose zero-sized inputs, layers, and outputs have been removed with `prune`.

One use for this simplification is to prepare a network with zero-sized subobjects for Simulink, where zero-sized signals are not supported.

```
[Xp,Xip,Aip,Tp] = prunedata(pi,pl,po,X,Xi,Ai,T) takes these arguments,
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pi</code></td>
<td>Indices of pruned inputs</td>
</tr>
<tr>
<td><code>pl</code></td>
<td>Indices of pruned layers</td>
</tr>
<tr>
<td><code>po</code></td>
<td>Indices of pruned outputs</td>
</tr>
<tr>
<td><code>X</code></td>
<td>Input data</td>
</tr>
<tr>
<td><code>Xi</code></td>
<td>Initial input delay states</td>
</tr>
<tr>
<td><code>Ai</code></td>
<td>Initial layer delay states</td>
</tr>
<tr>
<td><code>T</code></td>
<td>Target data</td>
</tr>
</tbody>
</table>

and returns the pruned inputs, input and layer delay states, and targets.

Examples

Here a NARX dynamic network is created which has one external input and a second input which feeds back from the output.
net = narxnet(20);
view(net)

The network is then trained on a single random time-series problem with 50 timesteps. The external input happens to have no elements.

X = nndata(0,1,50);
T = nndata(1,1,50);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts);

The network and data are then pruned before generating a Simulink diagram and initializing its input and layer states.

[net2,pi,pl,po] = prune(net);
view(net)
[Xs2,Xi2,Ai2,Ts2] = prunedata(net,pi,pl,po,Xs,Xi,Ai,Ts)
[sysName,netName] = gensim(net);
setsiminit(sysName,netName,Xi2,Ai2)

See Also
gensim | prune

Introduced in R2010b
purelin

Linear transfer function

Graph and Symbol

![Graph and Symbol](image)

\[ a = \text{purelin}(n) \]

Linear Transfer Function

Syntax

\[ A = \text{purelin}(N,FP) \]
\[ \text{info} = \text{purelin}('\text{code}') \]

Description

purelin is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \text{purelin}(N,FP) \] takes \( N \) and optional function parameters,

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>S-by-Q matrix of net input (column) vectors</td>
</tr>
<tr>
<td>( FP )</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \( A \), an S-by-Q matrix equal to \( N \).

\[ \text{info} = \text{purelin}('\text{code}') \] returns useful information for each supported \textit{code} character vector:
purelin('name') returns the name of this function.

purelin('output',FP) returns the [min max] output range.

purelin('active',FP) returns the [min max] active input range.

purelin('fullderiv') returns 1 or 0, depending on whether $\frac{dA}{dN}$ is $S$-by-$S$-by-$Q$ or $S$-by-$Q$.

purelin('fpnames') returns the names of the function parameters.

purelin('fpdefaults') returns the default function parameters.

**Examples**

Here is the code to create a plot of the purelin transfer function.

```matlab
n = -5:0.1:5;
a = purelin(n);
plot(n,a)
```

Assign this transfer function to layer $i$ of a network.

```matlab
net.layers{i}.transferFcn = 'purelin';
```

**Algorithms**

```matlab
a = purelin(n) = n
```

**See Also**

satlin|satlins|sim

**Introduced before R2006a**
quant

Discretize values as multiples of quantity

Syntax

quant(X,Q)

Description

quant(X,Q) takes two inputs,

<table>
<thead>
<tr>
<th>X</th>
<th>Matrix, vector, or scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Minimum value</td>
</tr>
</tbody>
</table>

and returns values from X rounded to nearest multiple of Q.

Examples

x = [1.333 4.756 -3.897];
y = quant(x,0.1)

Introduced before R2006a
radbas
Radial basis transfer function

Graph and Symbol

Radial Basis Function

Syntax

\[ A = \text{radbas}(N,FP) \]

Description

radbas is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \text{radbas}(N,FP) \] takes one or two inputs,

<table>
<thead>
<tr>
<th>( N )</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( FP )</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \( A \), an \( S \)-by-\( Q \) matrix of the radial basis function applied to each element of \( N \).
Examples

Here you create a plot of the \texttt{radbas} transfer function.

\begin{verbatim}
n = -5:0.1:5;  
a = \texttt{radbas}(n);  
plot(n,a)
\end{verbatim}

Assign this transfer function to layer \texttt{i} of a network.

\begin{verbatim}
net.layers{\texttt{i}}.transferFcn = '\texttt{radbas}';
\end{verbatim}

Algorithms

\begin{verbatim}
a = \texttt{radbas}(n) = \exp(-n^2)
\end{verbatim}

See Also

\texttt{radbasn} | \texttt{sim} | \texttt{tribas}

Introduced before \texttt{R2006a}
radbasn

Normalized radial basis transfer function

Graph and Symbol

Radial Basis Function

Syntax

A = radbasn(N,FP)

Description

radbasn is a neural transfer function. Transfer functions calculate a layer's output from its net input. This function is equivalent to radbas, except that output vectors are normalized by dividing by the sum of the pre-normalized values.

A = radbasn(N,FP) takes one or two inputs,

| N         | S-by-Q matrix of net input (column) vectors |
| FP        | Struct of function parameters (ignored)    |

and returns A, an S-by-Q matrix of the radial basis function applied to each element of N.
Examples

Here six random 3-element vectors are passed through the radial basis transform and normalized.

\[ n = \text{rand}(3,6) \]
\[ a = \text{radbasn}(n) \]

Assign this transfer function to layer \( i \) of a network.

\[ \text{net.layers}\{i\}.\text{transferFcn} = '\text{radbasn}' \; \]

Algorithms

\[ a = \text{radbasn}(n) = \exp(-n^2) / \sum(\exp(-n^2)) \]

See Also

radbas | sim | tribas

Introduced in R2010b
randnc

Normalized column weight initialization function

Syntax

\[ W = \text{randnc}(S,PR) \]

Description

randnc is a weight initialization function.

\[ W = \text{randnc}(S,PR) \] takes two inputs,

<table>
<thead>
<tr>
<th>( S )</th>
<th>Number of rows (neurons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PR )</td>
<td>R-by-2 matrix of input value ranges = ([P_{\text{min}} \ P_{\text{max}}])</td>
</tr>
</tbody>
</table>

and returns an \( S \)-by-\( R \) random matrix with normalized columns.

You can also call this in the form \( \text{randnc}(S,R) \).

Examples

A random matrix of four normalized three-element columns is generated:

\[ M = \text{randnc}(3,4) \]

\[
M =
\begin{bmatrix}
-0.6007 & -0.4715 & -0.2724 & 0.5596 \\
-0.7628 & -0.6967 & -0.9172 & 0.7819 \\
-0.2395 & 0.5406 & -0.2907 & 0.2747
\end{bmatrix}
\]

See Also

randnr
Introduced before R2006a
randnr

Normalized row weight initialization function

Syntax

\[ W = \text{randnr}(S,PR) \]

Description

\( \text{randnr} \) is a weight initialization function.

\( W = \text{randnr}(S,PR) \) takes two inputs,

<table>
<thead>
<tr>
<th>S</th>
<th>Number of rows (neurons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>R-by-2 matrix of input value ranges = [Pmin Pmax]</td>
</tr>
</tbody>
</table>

and returns an S-by-R random matrix with normalized rows.

You can also call this in the form \( \text{randnr}(S,R) \).

Examples

A matrix of three normalized four-element rows is generated:

\[ M = \text{randnr}(3,4) \]

\[
M =
\begin{bmatrix}
0.9713 & 0.0800 & -0.1838 & -0.1282 \\
0.8228 & 0.0338 & 0.1797 & 0.5381 \\
-0.3042 & -0.5725 & 0.5436 & 0.5331
\end{bmatrix}
\]

See Also

randnc
Introduced before R2006a
rands

Symmetric random weight/bias initialization function

Syntax

\[
\begin{align*}
W &= \text{rands}(S, PR) \\
M &= \text{rands}(S, R) \\
v &= \text{rands}(S)
\end{align*}
\]

Description

\textit{rands} is a weight/bias initialization function.

\[W = \text{rands}(S, PR)\] takes

\[
\begin{array}{|c|}
\hline
S & \text{Number of neurons} \\
PR & \text{R-by-2 matrix of R input ranges} \\
\hline
\end{array}
\]

and returns an S-by-R weight matrix of random values between -1 and 1.

\[M = \text{rands}(S, R)\] returns an S-by-R matrix of random values. \[v = \text{rands}(S)\] returns an S-by-1 vector of random values.

Examples

Here, three sets of random values are generated with \textit{rands}.

\[
\begin{align*}
\text{rands}(4, [0 1; -2 2]) \\
\text{rands}(4) \\
\text{rands}(2, 3)
\end{align*}
\]
Network Use

To prepare the weights and the bias of layer i of a custom network to be initialized with rands,

1. Set net.initFcn to 'initlay'. (net.initParam automatically becomes initlay's default parameters.)
2. Set net.layers{i}.initFcn to 'initwb'.
3. Set each net.inputWeights{i,j}.initFcn to 'rands'.
4. Set each net.layerWeights{i,j}.initFcn to 'rands'.
5. Set each net.biases{i}.initFcn to 'rands'.

To initialize the network, call init.

See Also
init | initlay | initwb | randnc | randnr | randsmall

Introduced before R2006a
randsmall

Small random weight/bias initialization function

Syntax

\[
W = \text{randsmall}(S, PR) \\
M = \text{rands}(S, R) \\
v = \text{rands}(S)
\]

Description

randsmall is a weight/bias initialization function.

\[
W = \text{randsmall}(S, PR)\] takes

<table>
<thead>
<tr>
<th>S</th>
<th>Number of neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>R-by-2 matrix of R input ranges</td>
</tr>
</tbody>
</table>

and returns an S-by-R weight matrix of small random values between -0.1 and 0.1.

\[
M = \text{rands}(S, R)\] returns an S-by-R matrix of random values. \( v = \text{rands}(S)\) returns an S-by-1 vector of random values.

Examples

Here three sets of random values are generated with \text{rands}.

\[
\text{randsmall}(4,[0 1; -2 2]) \\
\text{randsmall}(4) \\
\text{randsmall}(2,3)
\]
Network Use

To prepare the weights and the bias of layer i of a custom network to be initialized with rands,

1. Set net.initFcn to 'initlay'。(net.initParam automatically becomes initlay's default parameters.)
2. Set net.layers{i}.initFcn to 'initwb'.
3. Set each net.inputWeights{i,j}.initFcn to 'randsmall'.
4. Set each net.layerWeights{i,j}.initFcn to 'randsmall'.
5. Set each net.biases{i}.initFcn to 'randsmall'.

To initialize the network, call init.

See Also
init | initlay | initwb | randnc | randnr | rands

Introduced in R2010b
randtop

Random layer topology function

Syntax

pos = randtop(dimensions)

Description

randtop calculates the neuron positions for layers whose neurons are arranged in an N-dimensional random pattern.

pos = randtop(dimensions) takes one argument:

<table>
<thead>
<tr>
<th>dimensions</th>
<th>Row vector of dimension sizes</th>
</tr>
</thead>
</table>

and returns an N-by-S matrix of N coordinate vectors, where N is the number of dimensions and S is the product of dimensions.

Examples

Display Layer with Random Pattern

This shows how to display a two-dimensional layer with neurons arranged in a random pattern.

pos = randtop([18 12]);
plotsom(pos)
See Also
gridtop | hextop | tritop

Introduced before R2006a
regression

Linear regression

Syntax

\[
[r,m,b] = \text{regression}(t,y) \\
[r,m,b] = \text{regression}(t,y,'one')
\]

Description

\[
[r,m,b] = \text{regression}(t,y)
\]
takes these arguments,

<table>
<thead>
<tr>
<th>t</th>
<th>Target matrix or cell array data with a total of N matrix rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Output matrix or cell array data of the same size</td>
</tr>
</tbody>
</table>

and returns these outputs,

<table>
<thead>
<tr>
<th>r</th>
<th>Regression values for each of the N matrix rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Slope of regression fit for each of the N matrix rows</td>
</tr>
<tr>
<td>b</td>
<td>Offset of regression fit for each of the N matrix rows</td>
</tr>
</tbody>
</table>

\[
[r,m,b] = \text{regression}(t,y,'one')
\]
combines all matrix rows before regressing, and returns single scalar regression, slope, and offset values.

Examples

Fit Regression Model and Plot Fitted Values versus Targets

Train a feedforward network, then calculate and plot the regression between its targets and outputs.

\[
[x,t] = \text{simplefit}\_\text{dataset}; \\
\text{net} = \text{feedforwardnet}(20);
\]
net = train(net,x,t);
y = net(x);
[r,m,b] = regression(t,y)

r = 1.0000
m = 1.0000
b = 1.0878e-04

plotregression(t,y)
See Also
confusion | plotregression

Introduced in R2010b
removeconstantrows

Process matrices by removing rows with constant values

Syntax

\[
\begin{align*}
[Y,PS] &= \text{removeconstantrows}(X,\text{max\_range}) \\
[Y,PS] &= \text{removeconstantrows}(X,\text{FP}) \\
Y &= \text{removeconstantrows('apply',X,PS)} \\
X &= \text{removeconstantrows('reverse',Y,PS)}
\end{align*}
\]

Description

removeconstantrows processes matrices by removing rows with constant values.

\[
[Y,PS] = \text{removeconstantrows}(X,\text{max\_range})\]

takes \(X\) and an optional parameter, \(X\) \(N\)-by-\(Q\) matrix \(\text{max\_range}\) Maximum range of values for row to be removed (default is 0) and returns

\[
\begin{align*}
X & \quad \text{N-by-Q matrix} \\
\text{max\_range} & \quad \text{Maximum range of values for row to be removed (default is 0)} \\
Y & \quad \text{M-by-Q matrix with N - M rows deleted} \\
PS & \quad \text{Process settings that allow consistent processing of values}
\end{align*}
\]

\[
[Y,PS] = \text{removeconstantrows}(X,\text{FP})\]

takes parameters as a struct: \(FP.\text{max\_range}\).

\[
Y = \text{removeconstantrows('apply',X,PS)}\]

returns \(Y\), given \(X\) and settings \(PS\).

\[
X = \text{removeconstantrows('reverse',Y,PS)}\]

returns \(X\), given \(Y\) and settings \(PS\).

Any NaN values in the input matrix are treated as missing data, and are not considered as unique values. So, for example, removeconstantrows removes the first row from the matrix \([1\ 1\ 1\ \text{NaN};\ 1\ 1\ 1\ 2]\).
**Examples**

Format a matrix so that the rows with constant values are removed.

\[
x1 = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 1 & 1 \\ 3 & 2 & 2 \\ 0 & 0 & 0 \end{bmatrix};
\]
\[
[y1,PS] = \text{removeconstantrows}(x1);
\]

\[
y1 =
\begin{bmatrix}
1 & 2 & 4 \\
3 & 2 & 2
\end{bmatrix}
\]

\[
PS =
\begin{align*}
\text{max\_range:} &\ 0 \\
\text{keep:} &\ [1\ 3] \\
\text{remove:} &\ [2\ 4] \\
\text{value:} &\ [2\times1\ \text{double}] \\
\text{xrows:} &\ 4 \\
\text{yrows:} &\ 2 \\
\text{constants:} &\ [2\times1\ \text{double}] \\
\text{no\_change:} &\ 0
\end{align*}
\]

Next, apply the same processing settings to new values.

\[
x2 = \begin{bmatrix} 5 & 2 & 3 \\ 1 & 1 & 1 \\ 6 & 7 & 3 \\ 0 & 0 & 0 \end{bmatrix};
\]
\[
y2 = \text{removeconstantrows('apply',x2,PS)}
\]

\[
5\ 2\ 3
6\ 7\ 3
\]

Reverse the processing of \(y1\) to get the original \(x1\) matrix.

\[
x1\_again = \text{removeconstantrows('reverse',y1,PS)}
\]

\[
1\ 2\ 4
1\ 1\ 1
3\ 2\ 2
0\ 0\ 0
\]

**See Also**

fixunknowns | mapminmax | mapstd | processpca

**Introduced in R2006a**
**removedelay**

Remove delay to neural network's response

**Syntax**

```matlab
net = removedelay(net,n)
```

**Description**

`net = removedelay(net,n)` takes these arguments,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network</td>
</tr>
<tr>
<td>n</td>
<td>Number of delays</td>
</tr>
</tbody>
</table>

and returns the network with input delay connections decreased, and output feedback delays increased, by the specified number of delays `n`. The result is a network which behaves identically, except that outputs are produced `n` timesteps earlier.

If the number of delays `n` is not specified, a default of one delay is used.

**Examples**

**Remove and Add Delay to Network**

This example creates, trains, and simulates a time delay network in its original form, on an input time series `X` and target series `T`. Then the delay is removed and later added back. The first and third outputs will be identical, while the second result will include a new prediction for the following step.

```matlab
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
```
y1 = net1(Xs,Xi);
view(net1)

net2 = removedelay(net1);
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);
y2 = net2(Xs,Xi);
view(net2)

net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi);
view(net3)
See Also
adddelay | closeloop | openloop

Introduced in R2010b
removerows

Process matrices by removing rows with specified indices

Syntax

[Y,PS] = removerows(X,'ind',ind)
[Y,PS] = removerows(X,FP)
Y = removerows('apply',X,PS)
X = removerows('reverse',Y,PS)
dx_dy = removerows('dx',X,Y,PS)
dx_dy = removerows('dx',X,[],PS)
name = removerows('name')
fp = removerows('pdefaults')
names = removerows('pdesc')
removerows('pcheck',FP)

Description

removerows processes matrices by removing rows with the specified indices.

[Y,PS] = removerows(X,'ind',ind) takes X and an optional parameter, 

<table>
<thead>
<tr>
<th>X</th>
<th>N-by-Q matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ind</td>
<td>Vector of row indices to remove (default is [])</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Y</th>
<th>M-by-Q matrix, where M == N-length(ind)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>Process settings that allow consistent processing of values</td>
</tr>
</tbody>
</table>

[Y,PS] = removerows(X,FP) takes parameters as a struct: FP.ind.

Y = removerows('apply',X,PS) returns Y, given X and settings PS.

X = removerows('reverse',Y,PS) returns X, given Y and settings PS.
dx_dy = removerows('dx',X,Y,PS) returns the M-by-N-by-Q derivative of Y with respect to X.

dx_dy = removerows('dx',X,[],PS) returns the derivative, less efficiently.

name = removerows('name') returns the name of this process method.

fp = removerows('pdefaults') returns the default process parameter structure.

names = removerows('pdesc') returns the process parameter descriptions.

removerows('pcheck',FP) throws an error if any parameter is illegal.

**Examples**

Here is how to format a matrix so that rows 2 and 4 are removed:

x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0]
[y1,ps] = removerows(x1,'ind',[2 4])

Next, apply the same processing settings to new values.

x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0]
y2 = removerows('apply',x2,ps)

Reverse the processing of y1 to get x1 again.

x1_again = removerows('reverse',y1,ps)

**Algorithms**

In the reverse calculation, the unknown values of replaced rows are represented with NaN values.

**See Also**

fixunknowns | mapminmax | mapstd | processpca

**Introduced in R2006a**
**revert**

Change network weights and biases to previous initialization values

**Syntax**

```matlab
net = revert (net)
```

**Description**

`net = revert (net)` returns neural network `net` with weight and bias values restored to the values generated the last time the network was initialized.

If the network is altered so that it has different weight and bias connections or different input or layer sizes, then `revert` cannot set the weights and biases to their previous values and they are set to zeros instead.

**Examples**

Here a perceptron is created with input size set to 2 and number of neurons to 1.

```matlab
net = perceptron;
net.inputs{1}.size = 2;
net.layers{1}.size = 1;
```

The initial network has weights and biases with zero values.

```matlab
net.iw{1,1}, net.b{1}
```

Change these values as follows:

```matlab
net.iw{1,1} = [1 2];
net.b{1} = 5;
net.iw{1,1}, net.b{1}
```

You can recover the network's initial values as follows:
net = revert(net);
net.iw{1,1}, net.b{1}

See Also
adapt | init | sim | train

Introduced before R2006a
**roc**

Receiver operating characteristic

**Syntax**

```matlab
[tpr,fpr,thresholds] = roc(targets,outputs)
```

**Description**

The *receiver operating characteristic* is a metric used to check the quality of classifiers. For each class of a classifier, roc applies threshold values across the interval \([0, 1]\) to outputs. For each threshold, two values are calculated, the True Positive Ratio (TPR) and the False Positive Ratio (FPR). For a particular class \(i\), TPR is the number of outputs whose actual and predicted class is class \(i\), divided by the number of outputs whose predicted class is class \(i\). FPR is the number of outputs whose actual class is not class \(i\), but predicted class is class \(i\), divided by the number of outputs whose predicted class is not class \(i\).

You can visualize the results of this function with `plotroc`.

**Syntax**

```matlab
[tpr,fpr,thresholds] = roc(targets,outputs)
```

takes these arguments:

<table>
<thead>
<tr>
<th>targets</th>
<th>S-by-Q matrix, where each column vector contains a single 1 value, with all other elements 0. The index of the 1 indicates which of S categories that vector represents.</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputs</td>
<td>S-by-Q matrix, where each column contains values in the range ([0, 1]). The index of the largest element in the column indicates which of S categories that vector presents. Alternately, 1-by-Q vector, where values greater or equal to (0.5) indicate class membership, and values below (0.5), nonmembership.</td>
</tr>
</tbody>
</table>

and returns these values:
roc(targets,outputs) takes these arguments:

| targets | 1-by-Q matrix of Boolean values indicating class membership. |
| outputs | S-by-Q matrix, of values in \([0,1]\) interval, where values greater than or equal to 0.5 indicate class membership. |

and returns these values:

| tpr | 1-by-N vector of true-positive/positive ratios. |
| fpr | 1-by-N vector of false-positive/negative ratios. |
| thresholds | 1-by-N vector of thresholds over interval \([0,1]\). |

**Examples**

```matlab
load iris_dataset
net = patternnet(20);
net = train(net,irisInputs,irisTargets);
irisOutputs = sim(net,irisInputs);
[tpr,fpr,thresholds] = roc(irisTargets,irisOutputs)
```

**See Also**

confusion | plotroc

**Introduced in R2008a**
sae

Sum absolute error performance function

Syntax

perf = sae(net,t,y,ew)
[...] = sae(...,'regularization',regularization)
[...] = sae(...,'normalization',normalization)
[...] = sae(...,'squaredWeighting',squaredWeighting)
[...] = sae(...,FP)

Description

sae is a network performance function. It measures performance according to the sum of squared errors.

perf = sae(net,t,y,ew) takes these input arguments and optional function parameters,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Matrix or cell array of target vectors</td>
</tr>
<tr>
<td>y</td>
<td>Matrix or cell array of output vectors</td>
</tr>
<tr>
<td>ew</td>
<td>Error weights (default = {1})</td>
</tr>
</tbody>
</table>

and returns the sum squared error.

This function has three optional function parameters that can be defined with parameter name/pair arguments, or as a structure FP argument with fields having the parameter name and assigned the parameter values:

[...] = sae(...,'regularization',regularization)
[...] = sae(...,'normalization',normalization)
[...] = sae(...,'squaredWeighting',squaredWeighting)
• **regularization** — can be set to any value between the default of 0 and 1. The greater the regularization value, the more squared weights and biases are taken into account in the performance calculation.

• **normalization** — can be set to the default 'absolute', or 'normalized' (which normalizes errors to the [ +2 -2 ] range consistent with normalized output and target ranges of [ -1 1 ]) or 'percent' (which normalizes errors to the range [ -1 +1 ]).

• **squaredWeighting** — can be set to the default false, for applying error weights to absolute errors, or false for applying error weights to the squared errors before squaring.

### Examples

Here a network is trained to fit a simple data set and its performance calculated

```matlab
[x,t] = simplefit_dataset;
net = fitnet(10,'trainscg');
net.performFcn = 'sae';
net = train(net,x,t)
y = net(x)
e = t-y
perf = sae(net,t,y)
```

### Network Use

To prepare a custom network to be trained with sae, set `net.performFcn` to 'sae'. This automatically sets `net.performParam` to the default function parameters.

Then calling `train`, `adapt` or `perform` will result in `sae` being used to calculate performance.

*Introduced in R2010b*
satlin

Saturating linear transfer function

Graph and Symbol

![Graph](image)

\[ a = \text{satlin}(n) \]

Satlin Transfer Function

Syntax

\[ A = \text{satlin}(N, \text{FP}) \]

Description

\text{satlin} is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \text{satlin}(N, \text{FP}) \] takes one input,

\begin{array}{|l|l|}
\hline
N & S-by-Q matrix of net input (column) vectors \\
FP & Struct of function parameters (ignored) \\
\hline
\end{array}

and returns \( A \), the S-by-Q matrix of \( N \)'s elements clipped to \([0, 1]\).

\text{info} = \text{satlin('code')} returns useful information for each supported \textit{code} character vector:

\text{satlin('name')} returns the name of this function.
satlin('output',FP) returns the [min max] output range.
satlin('active',FP) returns the [min max] active input range.
satlin('fullderiv') returns 1 or 0, depending on whether \(\frac{dA}{dN}\) is \(S\)-by-\(S\)-by-\(Q\) or \(S\)-by-\(Q\).
satlin('fpnames') returns the names of the function parameters.
satlin('fpdefaults') returns the default function parameters.

**Examples**

Here is the code to create a plot of the satlin transfer function.

```matlab
n = -5:0.1:5;
a = satlin(n);
plot(n,a)
```

Assign this transfer function to layer \(i\) of a network.

```matlab
net.layers{i}.transferFcn = 'satlin';
```

**Algorithms**

\[
a = \begin{cases} 
0, & \text{if } n \leq 0 \\
 n, & \text{if } 0 \leq n \leq 1 \\
1, & \text{if } 1 \leq n 
\end{cases}
\]

**See Also**

poslin|purelin|satlins|sim

**Introduced before R2006a**
satlins

Symmetric saturating linear transfer function

Graph and Symbol

\[ a = \text{satlins}(n) \]

Satlins Transfer Function

Syntax

\[ A = \text{satlins}(N, \text{FP}) \]

Description

satlins is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

A = satlins(N,FP) takes N and an optional argument,

<table>
<thead>
<tr>
<th>N</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (optional, ignored)</td>
</tr>
</tbody>
</table>

and returns A, the S-by-Q matrix of N’s elements clipped to [ -1, 1 ].

info = satlins('code') returns useful information for each supported code character vector:

satlins('name') returns the name of this function.
satlins('output',FP) returns the [min max] output range.
satlins('active',FP) returns the [min max] active input range.
satlins('fullderiv') returns 1 or 0, depending on whether dA_dN is S-by-S-by-Q or S-by-Q.
satlins('fpnames') returns the names of the function parameters.
satlins('fpdefaults') returns the default function parameters.

Examples

Here is the code to create a plot of the satlins transfer function.

n = -5:0.1:5;
a = satlins(n);
plot(n,a)

Algorithms

satlins(n) = -1, if n <= -1
n, if -1 <= n <= 1
1, if 1 <= n

See Also

poslin|purelin|satlin|sim

Introduced before R2006a
scalprod

Scalar product weight function

Syntax

\[ Z = \text{scalprod}(W,P) \]
\[ \text{dim} = \text{scalprod}('\text{size}',S,R,FP) \]
\[ \text{dw} = \text{scalprod}('\text{dw}',W,P,Z,FP) \]

Description

scalprod is the scalar product weight function. Weight functions apply weights to an input to get weighted inputs.

\[ Z = \text{scalprod}(W,P) \] takes these inputs,

<table>
<thead>
<tr>
<th>W</th>
<th>1-by-1 weight matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>R-by-Q matrix of Q input (column) vectors</td>
</tr>
</tbody>
</table>

and returns the R-by-Q scalar product of \( W \) and \( P \) defined by \( Z = w*P \).

\[ \text{dim} = \text{scalprod}('\text{size}',S,R,FP) \] takes the layer dimension \( S \), input dimension \( R \), and function parameters, and returns the weight size [1-by-1].

\[ \text{dw} = \text{scalprod}('\text{dw}',W,P,Z,FP) \] returns the derivative of \( Z \) with respect to \( W \).

Examples

Here you define a random weight matrix \( W \) and input vector \( P \) and calculate the corresponding weighted input \( Z \).

\[
W = \text{rand}(1,1); \\
P = \text{rand}(3,1); \\
Z = \text{scalprod}(W,P)
\]
**Network Use**

To change a network so an input weight uses `scalprod`, set `net.inputWeights{i,j}.weightFcn` to `'scalprod'`.

For a layer weight, set `net.layerWeights{i,j}.weightFcn` to `'scalprod'`.

In either case, call `sim` to simulate the network with `scalprod`.

**See Also**

`dist | dotprod | negdist | normprod | sim`

*Introduced in R2006a*
selforgmap

Self-organizing map

Syntax

selforgmap(dimensions, coverSteps, initNeighbor, topologyFcn, distanceFcn)

Description

Self-organizing maps learn to cluster data based on similarity, topology, with a preference (but no guarantee) of assigning the same number of instances to each class.

Self-organizing maps are used both to cluster data and to reduce the dimensionality of data. They are inspired by the sensory and motor mappings in the mammal brain, which also appear to automatically organizing information topologically.

selforgmap(dimensions, coverSteps, initNeighbor, topologyFcn, distanceFcn) takes these arguments,

| dimensions | Row vector of dimension sizes (default = [8 8]) |
| coverSteps | Number of training steps for initial covering of the input space (default = 100) |
| initNeighbor | Initial neighborhood size (default = 3) |
| topologyFcn | Layer topology function (default = 'hextop') |
| distanceFcn | Neuron distance function (default = 'linkdist') |

and returns a self-organizing map.
Examples

Use Self-Organizing Map to Cluster Data

Here a self-organizing map is used to cluster a simple set of data.

```matlab
x = simplecluster_dataset;
net = selforgmap([8 8]);
net = train(net,x);
view(net)
y = net(x);
classes = vec2ind(y);
```

See Also

competlayer | lvqnet | nctool

Introduced in R2010b
separatewb

Separate biases and weight values from weight/bias vector

Syntax

\[ [b, IW, LW] = \text{separatewb}(\text{net}, \text{wb}) \]

Description

\[ [b, IW, LW] = \text{separatewb}(\text{net}, \text{wb}) \] takes two arguments, 

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb</td>
<td>Weight/bias vector</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>b</th>
<th>Cell array of bias vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>IW</td>
<td>Cell array of input weight matrices</td>
</tr>
<tr>
<td>LW</td>
<td>Cell array of layer weight matrices</td>
</tr>
</tbody>
</table>

Examples

Here a feedforward network is trained to fit some data, then its bias and weight values formed into a vector. The single vector is then redivided into the original biases and weights.

\[
[x, t] = \text{simplefit dataset};  
\text{net} = \text{feedforwardnet}(20);  
\text{net} = \text{train}(\text{net}, \text{x}, \text{t});  
\text{wb} = \text{formwb}(\text{net}, \text{net}.b, \text{net}.iw, \text{net}.lw)  
\text{[b, iw, lw]} = \text{separatewb}(\text{net}, \text{wb})
\]
See Also
formwb | getwb | setwb

Introduced in R2010b
**seq2con**

Convert sequential vectors to concurrent vectors

**Syntax**

\[
b = \text{seq2con}(s)
\]

**Description**

Deep Learning Toolbox software represents batches of vectors with a matrix, and sequences of vectors with multiple columns of a cell array.

\texttt{seq2con} and \texttt{con2seq} allow concurrent vectors to be converted to sequential vectors, and back again.

\[
b = \text{seq2con}(s)
\]

takes one input,

\[
s \quad \text{N-by-TS cell array of matrices with M columns}
\]

and returns

\[
b \quad \text{N-by-1 cell array of matrices with M*TS columns}
\]

**Examples**

Here three sequential values are converted to concurrent values.

\[
p1 = \{1 \ 4 \ 2\}
p2 = \text{seq2con}(p1)
\]

Here two sequences of vectors over three time steps are converted to concurrent vectors.

\[
p1 = \{[1; \ 1] \ [5; \ 4] \ [1; \ 2]; \ [3; \ 9] \ [4; \ 1] \ [9; \ 8]\}
p2 = \text{seq2con}(p1)
\]
See Also

con2seq | concur

Introduced before R2006a
setelements

Set neural network data elements

Syntax

setelements(x,i,v)

Description

setelements(x,i,v) takes these arguments,

<table>
<thead>
<tr>
<th>x</th>
<th>Neural network matrix or cell array data</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indices</td>
</tr>
<tr>
<td>v</td>
<td>Neural network data to store into x</td>
</tr>
</tbody>
</table>

and returns the original data x with the data v stored in the elements indicated by the indices i.

Examples

This code sets elements 1 and 3 of matrix data:

\[
x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 7 & 4 \end{bmatrix}
\]
\[
v = \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix};
\]
\[
y = \text{setelements}(x,[1 3],v)
\]

This code sets elements 1 and 3 of cell array data:

\[
x = \{[1:3; 4:6] \ [7:9; 10:12]; \ [13:15] \ [16:18]\}
\]
\[
v = \{[20 21 22; 23 24 25] \ [26 27 28; 29 30 31]\}
\]
\[
y = \text{setelements}(x,[1 3],v)
\]
See Also

catelements | getelements | nndata | numelements | setsamples | setsignals | settimesteps

Introduced in R2010b
setsamples

Set neural network data samples

Syntax

setsamples(x,i,v)

Description

setsamples(x,i,v) takes these arguments,

<table>
<thead>
<tr>
<th>x</th>
<th>Neural network matrix or cell array data</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indices</td>
</tr>
<tr>
<td>v</td>
<td>Neural network data to store into x</td>
</tr>
</tbody>
</table>

and returns the original data x with the data v stored in the samples indicated by the indices i.

Examples

This code sets samples 1 and 3 of matrix data:

\[
\begin{align*}
x & = \begin{bmatrix} 1 & 2 & 3; & 4 & 7 & 4 \end{bmatrix} \\
v & = \begin{bmatrix} 10 & 11; & 12 & 13 \end{bmatrix} \\
y & = \text{setsamples}(x,[1 \ 3],v)
\end{align*}
\]

This code sets samples 1 and 3 of cell array data:

\[
\begin{align*}
x & = \{[[1:3; \ 4:6] \ [7:9; \ 10:12]; \ [13:15] \ [16:18]]\} \\
v & = \{[[20 \ 21; \ 22 \ 23] \ [24 \ 25; \ 26 \ 27]; \ [28 \ 29] \ [30 \ 31]]\} \\
y & = \text{setsamples}(x,[1 \ 3],v)
\end{align*}
\]
See Also
catsamples | getsamples | nndata | numsamples | setelements | setsignals | settimesteps

Introduced in R2010b
setsignals

Set neural network data signals

Syntax

setsignals(x,i,v)

Description

setsignals(x,i,v) takes these arguments,

<table>
<thead>
<tr>
<th>x</th>
<th>Neural network matrix or cell array data</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indices</td>
</tr>
<tr>
<td>v</td>
<td>Neural network data to store into x</td>
</tr>
</tbody>
</table>

and returns the original data x with the data v stored in the signals indicated by the indices i.

Examples

This code sets signal 2 of cell array data:

```matlab
x = {{1:3; 4:6} [7:9; 10:12]; [13:15] [16:18]}
v = {{20:22} [23:25]}
y = setsignals(x,2,v)
```

See Also

catsignals | getsignals | nndata | numsignals | setelements | setsamples | settimesteps

Introduced in R2010b
**setsiminit**

Set neural network Simulink block initial conditions

**Syntax**

`setsiminit(sysName,netName,net,xi,ai,Q)`

**Description**

`setsiminit(sysName,netName,net,xi,ai,Q)` takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sysName</code></td>
<td>The name of the Simulink system containing the neural network block</td>
</tr>
<tr>
<td><code>netName</code></td>
<td>The name of the Simulink neural network block</td>
</tr>
<tr>
<td><code>net</code></td>
<td>The original neural network</td>
</tr>
<tr>
<td><code>xi</code></td>
<td>Initial input delay states</td>
</tr>
<tr>
<td><code>ai</code></td>
<td>Initial layer delay states</td>
</tr>
<tr>
<td><code>Q</code></td>
<td>Sample number (default is 1)</td>
</tr>
</tbody>
</table>

and sets the Simulink neural network blocks initial conditions as specified.

**Examples**

Here a NARX network is designed. The NARX network has a standard input and an open loop feedback output to an associated feedback input.

```matlab
[x,t] = simplenarx_dataset;
net = narxnet(1:2,1:2,20);
view(net)
[xs,xi,ai,ts] = preparets(net,x,{},t);
net = train(net,xs,ts,xi,ai);
[y] = net(xs,xi,ai);
```
Now the network is converted to closed loop, and the data is reformatted to simulate the network’s closed loop response.

\[
\text{net} = \text{closeloop}(	ext{net});
\]
\[
\text{view}(	ext{net})
\]
\[
[xs, xi, ai, ts] = \text{preparets}(	ext{net}, x, \{\}, t);
\]
\[
y = \text{net}(xs, xi, ai);
\]

Here the network is converted to a Simulink system with workspace input and output ports. Its delay states are initialized, inputs \(X_1\) defined in the workspace, and it is ready to be simulated in Simulink.

\[
[\text{sysName}, \text{netName}] = \text{gensim}(	ext{net}, \text{'InputMode'}, \text{'Workspace'}, \ldots
    \text{'OutputMode'}, \text{'Workspace'}, \text{'SolverMode'}, \text{'Discrete'});
\]
\[
\text{setsiminit}(	ext{sysName}, \text{netName}, \text{net}, xi, ai, 1);
\]
\[
x1 = \text{nndata2sim}(x, 1, 1);
\]

Finally the initial input and layer delays are obtained from the Simulink model. (They will be identical to the values set with \text{setsiminit}.)

\[
[\text{xi}, \text{ai}] = \text{getsiminit}(	ext{sysName}, \text{netName}, \text{net});
\]

**See Also**
gensim | getsiminit | nndata2sim | sim2nndata

**Introduced in R2010b**
settimesteps

Set neural network data timesteps

**Syntax**

settimesteps(x,i,v)

**Description**

settimesteps(x,i,v) takes these arguments,

<table>
<thead>
<tr>
<th>x</th>
<th>Neural network matrix or cell array data</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indices</td>
</tr>
<tr>
<td>v</td>
<td>Neural network data to store into x</td>
</tr>
</tbody>
</table>

and returns the original data \(x\) with the data \(v\) stored in the timesteps indicated by the indices \(i\).

**Examples**

This code sets timestep 2 of cell array data:

\[
x = \{[[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]] \}
\[
v = \{[[20:22; 23:25]; [25:27]] \}
\[
y = \text{settimesteps}(x, 2, v)
\]

**See Also**

cattimesteps | gettimesteps | nndata | numtimesteps | setelements | setsamples | setsignals

Introduced in R2010b
**setwb**

Set all network weight and bias values with single vector

**Syntax**

\[ \text{net} = \text{setwb} (\text{net}, \text{wb}) \]

**Description**

This function sets a network’s weight and biases to a vector of values.

\[ \text{net} = \text{setwb} (\text{net}, \text{wb}) \] takes the following inputs:

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>wb</td>
<td>Vector of weight and bias values</td>
</tr>
</tbody>
</table>

**Examples**

**Set Network's Weights and Biases**

This example shows how to set and view a network’s weight and bias values.

Create and configure a network.

\[
[x, t] = \text{simplefit\_dataset}; \\
\text{net} = \text{feedforwardnet}(3); \\
\text{net} = \text{configure}(\text{net}, x, t); \\
\text{view}(\text{net})
\]
This network has three weights and three biases in the first layer, and three weights and one bias in the second layer. So, the total number of weight and bias values in the network is 10. Set the weights and biases to random values.

```matlab
net = setwb(net,rand(10,1));
```

View the weight and bias values

```matlab
net.IW{1,1}
net.b{1}
```

```
ans =
    0.1576
    0.9706
    0.9572
```

```
ans =
    0.5469
    0.9575
    0.9649
```

See Also
formwb | getwb | separatewb

Introduced in R2010b
**sim**

Simulate neural network

**Syntax**

\[
[Y, Xf, Af] = \text{sim}(net, X, Xi, Ai, T)
\]

\[
[Y, Xf, Af] = \text{sim}(net, \{Q, TS\}, Xi, Ai)
\]

\[
[Y, ...] = \text{sim}(\text{net}, ..., 'useParallel', ...)
\]

\[
[Y, ...] = \text{sim}(\text{net}, ..., 'useGPU', ...)
\]

\[
[Y, ...] = \text{sim}(\text{net}, ..., 'showResources', ...)
\]

\[
[Y\text{composite}, ...] = \text{sim}(\text{net}, X\text{composite}, ...)
\]

\[
[Y\text{gpu}, ...] = \text{sim}(\text{net}, X\text{gpu}, ...)
\]

**To Get Help**

Type `help network/sim`.

**Description**

`sim` simulates neural networks.

\[
[Y, Xf, Af] = \text{sim}(\text{net}, X, Xi, Ai, T)
\]
takes

<table>
<thead>
<tr>
<th>net</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Network inputs</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay conditions (default = zeros)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay conditions (default = zeros)</td>
</tr>
<tr>
<td>T</td>
<td>Network targets (default = zeros)</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>Y</th>
<th>Network outputs</th>
</tr>
</thead>
</table>
**Xf** | Final input delay conditions
---|---
**Af** | Final layer delay conditions

`sim` is usually called implicitly by calling the neural network as a function. For instance, these two expressions return the same result:

\[
y = \text{sim}(\text{net}, x, xi, ai) \\
y = \text{net}(x, xi, ai)
\]

Note that arguments `Xi`, `Ai`, `Xf`, and `Af` are optional and need only be used for networks that have input or layer delays.

The signal arguments can have two formats: cell array or matrix.

The cell array format is easiest to describe. It is most convenient for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:

<table>
<thead>
<tr>
<th><strong>X</strong></th>
<th>Ni-by-TS cell array</th>
<th>Each element $X{i, ts}$ is an Ri-by-Q matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xi</strong></td>
<td>Ni-by-ID cell array</td>
<td>Each element $Xi{i, k}$ is an Ri-by-Q matrix.</td>
</tr>
<tr>
<td><strong>Ai</strong></td>
<td>Nl-by-LD cell array</td>
<td>Each element $Ai{i, k}$ is an Si-by-Q matrix.</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>No-by-TS cell array</td>
<td>Each element $X{i, ts}$ is a Ui-by-Q matrix.</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>No-by-TS cell array</td>
<td>Each element $Y{i, ts}$ is a Ui-by-Q matrix.</td>
</tr>
<tr>
<td><strong>Xf</strong></td>
<td>Ni-by-ID cell array</td>
<td>Each element $Xf{i, k}$ is an Ri-by-Q matrix.</td>
</tr>
<tr>
<td><strong>Af</strong></td>
<td>Nl-by-LD cell array</td>
<td>Each element $Af{i, k}$ is an Si-by-Q matrix.</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th><strong>Ni</strong></th>
<th>(=)</th>
<th>net.numInputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nl</strong></td>
<td>(=)</td>
<td>net.numLayers</td>
</tr>
<tr>
<td><strong>No</strong></td>
<td>(=)</td>
<td>net.numOutputs</td>
</tr>
</tbody>
</table>
ID = net.numInputDelays
LD = net.numLayerDelays
TS = Number of time steps
Q = Batch size
Ri = net.inputs{i}.size
Si = net.layers{i}.size
Ui = net.outputs{i}.size

The columns of Xi, Ai, Xf, and Af are ordered from oldest delay condition to most recent:

<table>
<thead>
<tr>
<th>Xi{i,k} = Input i at time ts = k - ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xf{i,k} = Input i at time ts = TS + k - ID</td>
</tr>
<tr>
<td>Ai{i,k} = Layer output i at time ts = k - LD</td>
</tr>
<tr>
<td>Af{i,k} = Layer output i at time ts = TS + k - LD</td>
</tr>
</tbody>
</table>

The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can also be used with networks that have more.

Each matrix argument is found by storing the elements of the corresponding cell array argument in a single matrix:

| X = (sum of Ri)-by-Q matrix |
| Xi = (sum of Ri)-by-(ID*Q) matrix |
| Ai = (sum of Si)-by-(LD*Q) matrix |
| T = (sum of Ui)-by-Q matrix |
| Y = (sum of Ui)-by-Q matrix |
| Xf = (sum of Ri)-by-(ID*Q) matrix |
| Af = (sum of Si)-by-(LD*Q) matrix |

[Y,Xf,Af] = sim(net,{Q TS},Xi,Ai) is used for networks that do not have an input when cell array notation is used.
[Y,...] = sim(net,...,'useParallel',...), [Y,...] = sim(net,...,'useGPU',...), or [Y,...] = sim(net,...,'showResources',...) (or the network called as a function) accepts optional name/value pair arguments to control how calculations are performed. Two of these options allow training to happen faster or on larger datasets using parallel workers or GPU devices if Parallel Computing Toolbox is available. These are the optional name/value pairs:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'useParallel','no'</td>
<td>Calculations occur on normal MATLAB thread. This is the default 'useParallel' setting.</td>
</tr>
<tr>
<td>'useParallel','yes'</td>
<td>Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.</td>
</tr>
<tr>
<td>'useGPU','no'</td>
<td>Calculations occur on the CPU. This is the default 'useGPU' setting.</td>
</tr>
<tr>
<td>'useGPU','yes'</td>
<td>Calculations occur on the current gpuDevice if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current gpuDevice is not supported, calculations remain on the CPU. If 'useParallel' is also 'yes' and a parallel pool is open, then each worker with a unique GPU uses that GPU, other workers run calculations on their respective CPU cores.</td>
</tr>
<tr>
<td>'useGPU','only'</td>
<td>If no parallel pool is open, then this setting is the same as 'yes'. If a parallel pool is open, then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.</td>
</tr>
<tr>
<td>'showResources','no'</td>
<td>Do not display computing resources used at the command line. This is the default setting.</td>
</tr>
<tr>
<td>'showResources','yes'</td>
<td>Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker's computation mode is described, including workers in the pool that are not used.</td>
</tr>
</tbody>
</table>

[Ycomposite,...] = sim(net,Xcomposite,...) takes Composite data and returns Composite results. If Composite data is used, then 'useParallel' is automatically set to 'yes'.

[Ygpu,...] = sim(net,Xgpu,...) takes gpuArray data and returns gpuArray results. If gpuArray data is used, then 'useGPU' is automatically set to 'yes'.
Examples

In the following examples, the `sim` function is called implicitly by calling the neural network object (`net`) as a function.

Simulate Feedforward Networks

This example loads a dataset that maps anatomical measurements $x$ to body fat percentages $t$. A feedforward network with 10 neurons is created and trained on that data, then simulated.

```matlab
[x,t] = bodyfat_dataset;
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
```

Simulate NARX Time Series Networks

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current $x$ and the magnet’s vertical position response $t$, then simulates the network. The function `preparets` prepares the data before training and simulation. It creates the open-loop network’s combined inputs $x_0$, which contains both the external input $x$ and previous values of position $t$. It also prepares the delay states $x_i$.

```matlab
[x,t] = maglev_dataset;
net = narxnet(10);
[xo,xi,~,to] = preparets(net,x,{},t);
net = train(net,xo,to,xi);
y = net(xo,xi)
```

This same system can also be simulated in closed-loop form.

```matlab
netc = closeloop(net);
view(netc)
[xc,xi,ai,tc] = preparets(netc,x,{},t);
yc = netc(xc,xi,ai);
```
Simulate in Parallel on a Parallel Pool

With Parallel Computing Toolbox you can simulate and train networks faster and on larger datasets than can fit on one PC. Here training and simulation happens across parallel MATLAB workers.

parpool
[X,T] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,X,T,'useParallel','yes','showResources','yes');
Y = net(X,'useParallel','yes');

Simulate on GPUs

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded as it is distributed, then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

Xc = Composite;
for i=1:numel(Xc)
    Xc{i} = X+rand(size(X))*0.1;  % Use real data instead of random
end
Yc = net(Xc,'showResources','yes');

Networks can be simulated using the current GPU device, if it is supported by Parallel Computing Toolbox.

gpuDevice % Check if there is a supported GPU
Y = net(X,'useGPU','yes','showResources','yes');

To put the data on a GPU manually, and get the results on the GPU:

Xgpu = gpuArray(X);
Ygpu = net(Xgpu,'showResources','yes');
Y = gather(Ygpu);

To run in parallel, with workers associated with unique GPUs taking advantage of that hardware, while the rest of the workers use CPUs:

Y = net(X,'useParallel','yes','useGPU','yes','showResources','yes');

Using only workers with unique GPUs might result in higher speeds, as CPU workers might not keep up.
Y = net(X,'useParallel','yes','useGPU','only','showResources','yes');

Algorithms

sim uses these properties to simulate a network net.

net.numInputs, net.numLayers
net.outputConnect, net.biasConnect
net.inputConnect, net.layerConnect

These properties determine the network’s weight and bias values and the number of delays associated with each weight:

net.IW{i,j}
net.LW{i,j}
net.b{i}
net.inputWeights{i,j}.delays
net.layerWeights{i,j}.delays

These function properties indicate how sim applies weight and bias values to inputs to get each layer’s output:

net.inputWeights{i,j}.weightFcn
net.layerWeights{i,j}.weightFcn
net.layers{i}.netInputFcn
net.layers{i}.transferFcn

See Also
adapt | init | revert | train

Introduced before R2006a
sim2nndata

Convert Simulink time series to neural network data

Syntax

sim2nndata(x)

Description

sim2nndata(x) takes either a column vector of values or a Simulink time series structure and converts it to a neural network data time series.

Examples

Here a random Simulink 20-step time series is created and converted.

simts = rands(20,1);
nnts = sim2nndata(simts)

Here a similar time series is defined with a Simulink structure and converted.

simts.time = 0:19
simts.signals.values = rands(20,1);
simts.dimensions = 1;
nnts = sim2nndata(simts)

See Also

nndata | nndata2sim

Introduced in R2010b
**softmax**

Soft max transfer function

**Graph and Symbol**

![Graph and Symbol](image)

**Syntax**

\[ A = \text{softmax}(N, FP) \]

**Description**

*softmax* is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \text{softmax}(N, FP) \]

 takes \( N \) and optional function parameters, \( N \) `-by-` \( Q \) matrix of net input (column) vectors, \( FP \) Struct of function parameters (ignored), and returns \( A \), the \( S \)-by-\( Q \) matrix of the softmax competitive function applied to each column of \( N \).

\[
\begin{array}{|l|}
\hline
N & S\text{-by-}Q \text{ matrix of net input (column) vectors} \\
FP & \text{Struct of function parameters (ignored)} \\
\hline
\end{array}
\]

and returns \( A \), the \( S\)-by-\( Q \) matrix of the softmax competitive function applied to each column of \( N \).

\text{info} = \text{softmax}('code') \) returns information about this function. The following codes are defined:

\text{softmax}('name') \) returns the name of this function.
softmax('output',FP) returns the [min  max] output range.
softmax('active',FP) returns the [min  max] active input range.
softmax('fullderiv') returns 1 or 0, depending on whether dA_dN is S-by-S-by-Q or S-by-Q.
softmax('fpnames') returns the names of the function parameters.
softmax('fpdefaults') returns the default function parameters.

Examples
Here you define a net input vector N, calculate the output, and plot both with bar graphs.

\[ n = [0; 1; -0.5; 0.5]; \]
\[ a = \text{softmax}(n); \]
\[ \text{subplot}(2,1,1), \text{bar}(n), \text{ylabel}('n') \]
\[ \text{subplot}(2,1,2), \text{bar}(a), \text{ylabel}('a') \]

Assign this transfer function to layer i of a network.

\[ \text{net.layers}\{i\}.\text{transferFcn} = '\text{softmax}'; \]

Algorithms
\[ a = \text{softmax}(n) = \exp(n)/\sum(\exp(n)) \]

See Also
compet | sim

Introduced before R2006a
srchbac

1-D minimization using backtracking

Syntax

[a,gX,perf,retcode,delta,tol] = srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)

Description

srchbac is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called backtracking.

[a,gX,perf,retcode,delta,tol] = srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf) takes these inputs,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Vector containing current values of weights and biases</td>
</tr>
<tr>
<td>Pd</td>
<td>Delayed input vectors</td>
</tr>
<tr>
<td>Tl</td>
<td>Layer target vectors</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial input delay conditions</td>
</tr>
<tr>
<td>Q</td>
<td>Batch size</td>
</tr>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
<tr>
<td>dX</td>
<td>Search direction vector</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient vector</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at current X</td>
</tr>
<tr>
<td>dperf</td>
<td>Slope of performance value at current X in direction of dX</td>
</tr>
<tr>
<td>delta</td>
<td>Initial step size</td>
</tr>
<tr>
<td>tol</td>
<td>Tolerance on search</td>
</tr>
</tbody>
</table>
and returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch_perf</td>
<td>Change in performance on previous step</td>
</tr>
<tr>
<td>a</td>
<td>Step size that minimizes performance</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient at new minimum point</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at new minimum point</td>
</tr>
<tr>
<td>retcode</td>
<td>Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.</td>
</tr>
<tr>
<td>delta</td>
<td>New initial step size, based on the current step size</td>
</tr>
<tr>
<td>tol</td>
<td>New tolerance on search</td>
</tr>
</tbody>
</table>

Parameters used for the backstepping algorithm are

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>Scale factor that determines sufficient reduction in perf</td>
</tr>
<tr>
<td>beta</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td>low_lim</td>
<td>Lower limit on change in step size</td>
</tr>
<tr>
<td>up_lim</td>
<td>Upper limit on change in step size</td>
</tr>
<tr>
<td>maxstep</td>
<td>Maximum step length</td>
</tr>
<tr>
<td>minstep</td>
<td>Minimum step length</td>
</tr>
<tr>
<td>scale_tol</td>
<td>Parameter that relates the tolerance tol to the initial step size delta, usually set to 20</td>
</tr>
</tbody>
</table>

The defaults for these parameters are set in the training function that calls them. See traincfgf, traincgb, traincgf, trainbfg, and trainoss.

Dimensions for these variables are
<table>
<thead>
<tr>
<th>Pd</th>
<th>No-by-Ni-by-TS cell array</th>
<th>Each element $P_{i,j,ts}$ is a $D_{ij}$-by-$Q$ matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_l$</td>
<td>$N_l$-by-TS cell array</td>
<td>Each element $P_{i,ts}$ is a $V_i$-by-$Q$ matrix.</td>
</tr>
<tr>
<td>V</td>
<td>$N_l$-by-LD cell array</td>
<td>Each element $A_{i,k}$ is an $S_i$-by-$Q$ matrix.</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th>$N_i$</th>
<th>$= \text{net.numInputs}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_l$</td>
<td>$= \text{net.numLayers}$</td>
</tr>
<tr>
<td>$L_D$</td>
<td>$= \text{net.numLayerDelays}$</td>
</tr>
<tr>
<td>$R_i$</td>
<td>$= \text{net.inputs{i}.size}$</td>
</tr>
<tr>
<td>$S_i$</td>
<td>$= \text{net.layers{i}.size}$</td>
</tr>
<tr>
<td>$V_i$</td>
<td>$= \text{net.targets{i}.size}$</td>
</tr>
<tr>
<td>$D_{ij}$</td>
<td>$= R_i \times \text{length(net.inputWeights{i,j}.delays)}$</td>
</tr>
</tbody>
</table>

**More About**

**Backtracking Search**

The backtracking search routine `srchbac` is best suited to use with the quasi-Newton optimization algorithms. It begins with a step multiplier of 1 and then backtracks until an acceptable reduction in the performance is obtained. On the first step it uses the value of performance at the current point and a step multiplier of 1. It also uses the value of the derivative of performance at the current point to obtain a quadratic approximation to the performance function along the search direction. The minimum of the quadratic approximation becomes a tentative optimum point (under certain conditions) and the performance at this point is tested. If the performance is not sufficiently reduced, a cubic interpolation is obtained and the minimum of the cubic interpolation becomes the new tentative optimum point. This process is continued until a sufficient reduction in the performance is obtained.

The backtracking algorithm is described in Dennis and Schnabel. It is used as the default line search for the quasi-Newton algorithms, although it might not be the best technique for all problems.
**Algorithms**

`srchbac` locates the minimum of the performance function in the search direction `dX`, using the backtracking algorithm described on page 126 and 328 of Dennis and Schnabel’s book, noted below.

**References**


**See Also**

`srchcha` | `srchgol` | `srchhyb`

**Introduced before R2006a**
srchbre

1-D interval location using Brent’s method

Syntax

\[
[a,gX,perf,retcode,delta,tol] = \\
\text{srchbre}(\text{net},X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch\_perf)
\]

Description

\text{srchbre} is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called Brent’s technique.

\[
[a,gX,perf,retcode,delta,tol] = \\
\text{srchbre}(\text{net},X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch\_perf)
\]
takes these inputs,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Vector containing current values of weights and biases</td>
</tr>
<tr>
<td>Pd</td>
<td>Delayed input vectors</td>
</tr>
<tr>
<td>Tl</td>
<td>Layer target vectors</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial input delay conditions</td>
</tr>
<tr>
<td>Q</td>
<td>Batch size</td>
</tr>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
<tr>
<td>dX</td>
<td>Search direction vector</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient vector</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at current X</td>
</tr>
<tr>
<td>dperf</td>
<td>Slope of performance value at current X in direction of dX</td>
</tr>
<tr>
<td>delta</td>
<td>Initial step size</td>
</tr>
<tr>
<td>tol</td>
<td>Tolerance on search</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td>ch_perf</td>
<td>Change in performance on previous step</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>a</th>
<th>Step size that minimizes performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>gX</td>
<td>Gradient at new minimum point</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at new minimum point</td>
</tr>
<tr>
<td>retcode</td>
<td>Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.</td>
</tr>
<tr>
<td>0</td>
<td>Normal</td>
</tr>
<tr>
<td>1</td>
<td>Minimum step taken</td>
</tr>
<tr>
<td>2</td>
<td>Maximum step taken</td>
</tr>
<tr>
<td>3</td>
<td>Beta condition not met</td>
</tr>
<tr>
<td>delta</td>
<td>New initial step size, based on the current step size</td>
</tr>
<tr>
<td>tol</td>
<td>New tolerance on search</td>
</tr>
</tbody>
</table>

Parameters used for the Brent algorithm are

<table>
<thead>
<tr>
<th>alpha</th>
<th>Scale factor that determines sufficient reduction in perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td>bmax</td>
<td>Largest step size</td>
</tr>
<tr>
<td>scale_tol</td>
<td>Parameter that relates the tolerance tol to the initial step size delta, usually set to 20</td>
</tr>
</tbody>
</table>

The defaults for these parameters are set in the training function that calls them. See traincfgf, traincgb, traincgp, trainbfg, and trainoss.

Dimensions for these variables are
<table>
<thead>
<tr>
<th>Pd</th>
<th>No-by-Ni-by-TS cell array</th>
<th>Each element $P_{i,j,ts}$ is a $D_{ij}$-by-$Q$ matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tl</td>
<td>Nl-by-TS cell array</td>
<td>Each element $P_{i,ts}$ is a $V_i$-by-$Q$ matrix.</td>
</tr>
<tr>
<td>Ai</td>
<td>Nl-by-LD cell array</td>
<td>Each element $A_{i,k}$ is an $S_i$-by-$Q$ matrix.</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th>Ni</th>
<th>=</th>
<th>net.numInputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nl</td>
<td>=</td>
<td>net.numLayers</td>
</tr>
<tr>
<td>LD</td>
<td>=</td>
<td>net.numLayerDelays</td>
</tr>
<tr>
<td>Ri</td>
<td>=</td>
<td>net.inputs{i}.size</td>
</tr>
<tr>
<td>Si</td>
<td>=</td>
<td>net.layers{i}.size</td>
</tr>
<tr>
<td>Vi</td>
<td>=</td>
<td>net.targets{i}.size</td>
</tr>
<tr>
<td>Dij</td>
<td>=</td>
<td>$R_i \times \text{length(net.inputWeights{i,j}.delays)}$</td>
</tr>
</tbody>
</table>

**More About**

**Brent’s Search**

Brent’s search is a linear search that is a hybrid of the golden section search and a quadratic interpolation. Function comparison methods, like the golden section search, have a first-order rate of convergence, while polynomial interpolation methods have an asymptotic rate that is faster than superlinear. On the other hand, the rate of convergence for the golden section search starts when the algorithm is initialized, whereas the asymptotic behavior for the polynomial interpolation methods can take many iterations to become apparent. Brent’s search attempts to combine the best features of both approaches.

For Brent’s search, you begin with the same interval of uncertainty used with the golden section search, but some additional points are computed. A quadratic function is then fitted to these points and the minimum of the quadratic function is computed. If this minimum is within the appropriate interval of uncertainty, it is used in the next stage of the search and a new quadratic approximation is performed. If the minimum falls outside the known interval of uncertainty, then a step of the golden section search is performed.
See [Bren73] for a complete description of this algorithm. This algorithm has the advantage that it does not require computation of the derivative. The derivative computation requires a backpropagation through the network, which involves more computation than a forward pass. However, the algorithm can require more performance evaluations than algorithms that use derivative information.

**Algorithms**

`srchbre` brackets the minimum of the performance function in the search direction $dX$, using Brent’s algorithm, described on page 46 of Scales (see reference below). It is a hybrid algorithm based on the golden section search and the quadratic approximation.

**References**


**See Also**

`srchbac` | `srchcha` | `srchgol` | `srchhyb`

*Introduced before R2006a*
srchcha

1-D minimization using Charalambous' method

Syntax

\[ [a,gX,perf,retcode,delta,tol] = srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf) \]

Description

srchcha is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique based on Charalambous’ method.

(srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)) takes these inputs,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Vector containing current values of weights and biases</td>
</tr>
<tr>
<td>Pd</td>
<td>Delayed input vectors</td>
</tr>
<tr>
<td>Tl</td>
<td>Layer target vectors</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial input delay conditions</td>
</tr>
<tr>
<td>Q</td>
<td>Batch size</td>
</tr>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
<tr>
<td>dX</td>
<td>Search direction vector</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient vector</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at current X</td>
</tr>
<tr>
<td>dperf</td>
<td>Slope of performance value at current X in direction of dX</td>
</tr>
<tr>
<td>delta</td>
<td>Initial step size</td>
</tr>
</tbody>
</table>
Tolerance on search

Change in performance on previous step

and returns

Step size that minimizes performance

Gradient at new minimum point

Performance value at new minimum point

Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

0 Normal

1 Minimum step taken

2 Maximum step taken

3 Beta condition not met

New initial step size, based on the current step size

New tolerance on search

Parameters used for the Charalambous algorithm are

Scale factor that determines sufficient reduction in perf

Scale factor that determines sufficiently large step size

Parameter to avoid small reductions in performance, usually set to 0.1

Parameter that relates the tolerance tol to the initial step size delta, usually set to 20

The defaults for these parameters are set in the training function that calls them. See \texttt{traincfg}, \texttt{traincgb}, \texttt{traincgp}, \texttt{trainbfg}, and \texttt{trainoss}.

Dimensions for these variables are
<table>
<thead>
<tr>
<th>Pd</th>
<th>No-by-Ni-by-TS cell array</th>
<th>Each element $P_{i,j,ts}$ is a $D_{ij}$-by-$Q$ matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tl</td>
<td>Nl-by-TS cell array</td>
<td>Each element $P_{i,ts}$ is a $V_i$-by-$Q$ matrix.</td>
</tr>
<tr>
<td>Ai</td>
<td>Nl-by-LD cell array</td>
<td>Each element $A_{i,k}$ is an $S_i$-by-$Q$ matrix.</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th>Ni</th>
<th>=</th>
<th>net.numInputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nl</td>
<td>=</td>
<td>net.numLayers</td>
</tr>
<tr>
<td>LD</td>
<td>=</td>
<td>net.numLayerDelays</td>
</tr>
<tr>
<td>Ri</td>
<td>=</td>
<td>net.inputs{i}.size</td>
</tr>
<tr>
<td>Si</td>
<td>=</td>
<td>net.layers{i}.size</td>
</tr>
<tr>
<td>Vi</td>
<td>=</td>
<td>net.targets{i}.size</td>
</tr>
<tr>
<td>Dij</td>
<td>=</td>
<td>$R_i \times $ length(net.inputWeights{i,j}.delays)</td>
</tr>
</tbody>
</table>

**More About**

**Charalambous’ Search**

The method of Charalambous, `srchcha`, was designed to be used in combination with a conjugate gradient algorithm for neural network training. Like `srchbre` and `srchhyb`, it is a hybrid search. It uses a cubic interpolation together with a type of sectioning.

See [Char92] for a description of Charalambous' search. This routine is used as the default search for most of the conjugate gradient algorithms because it appears to produce excellent results for many different problems. It does require the computation of the derivatives (backpropagation) in addition to the computation of performance, but it overcomes this limitation by locating the minimum with fewer steps. This is not true for all problems, and you might want to experiment with other line searches.
Algorithms

srchcha locates the minimum of the performance function in the search direction dX, using an algorithm based on the method described in Charalambous (see reference below).

References


See Also

srchbac | srchbre | srchgol | srchhyb

Introduced before R2006a
srchgol

1-D minimization using golden section search

Syntax

\[ [a,gX,perf,retcode,delta,tol] = srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf) \]

Description

srchgol is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique called the golden section search.

\[ [a,gX,perf,retcode,delta,tol] = srchgol(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf) \] takes these inputs,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Vector containing current values of weights and biases</td>
</tr>
<tr>
<td>Pd</td>
<td>Delayed input vectors</td>
</tr>
<tr>
<td>Tl</td>
<td>Layer target vectors</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial input delay conditions</td>
</tr>
<tr>
<td>Q</td>
<td>Batch size</td>
</tr>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
<tr>
<td>dX</td>
<td>Search direction vector</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient vector</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at current X</td>
</tr>
<tr>
<td>dperf</td>
<td>Slope of performance value at current X in direction of dX</td>
</tr>
<tr>
<td>delta</td>
<td>Initial step size</td>
</tr>
</tbody>
</table>
Tolerance on search
Change in performance on previous step

and returns

<table>
<thead>
<tr>
<th>a</th>
<th>Step size that minimizes performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>gX</td>
<td>Gradient at new minimum point</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at new minimum point</td>
</tr>
<tr>
<td>retcode</td>
<td>Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.</td>
</tr>
<tr>
<td>0</td>
<td>Normal</td>
</tr>
<tr>
<td>1</td>
<td>Minimum step taken</td>
</tr>
<tr>
<td>2</td>
<td>Maximum step taken</td>
</tr>
<tr>
<td>3</td>
<td>Beta condition not met</td>
</tr>
<tr>
<td>delta</td>
<td>New initial step size, based on the current step size</td>
</tr>
<tr>
<td>tol</td>
<td>New tolerance on search</td>
</tr>
</tbody>
</table>

Parameters used for the golden section algorithm are

<table>
<thead>
<tr>
<th>alpha</th>
<th>Scale factor that determines sufficient reduction in perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmax</td>
<td>Largest step size</td>
</tr>
<tr>
<td>scale_tol</td>
<td>Parameter that relates the tolerance tol to the initial step size delta, usually set to 20</td>
</tr>
</tbody>
</table>

The defaults for these parameters are set in the training function that calls them. See traincfgf, traincgb, traincgp, trainbfg, and trainoss.

Dimensions for these variables are

<p>| Pd          | No-by-Ni-by-TS cell array | Each element P{i,j,ts} is a Dij-by-Q matrix. |</p>
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ni</td>
<td>net.numInputs</td>
<td>net.numInputs</td>
</tr>
<tr>
<td>Nl</td>
<td>net.numLayers</td>
<td>net.numLayers</td>
</tr>
<tr>
<td>LD</td>
<td>net.numLayerDelays</td>
<td>net.numLayerDelays</td>
</tr>
<tr>
<td>Ri</td>
<td>net.inputs{i}.size</td>
<td>net.inputs{i}.size</td>
</tr>
<tr>
<td>Si</td>
<td>net.layers{i}.size</td>
<td>net.layers{i}.size</td>
</tr>
<tr>
<td>Vi</td>
<td>net.targets{i}.size</td>
<td>net.targets{i}.size</td>
</tr>
<tr>
<td>Dij</td>
<td>Ri * length(net.inputWeights{i,j}.delays)</td>
<td>Ri * length(net.inputWeights{i,j}.delays)</td>
</tr>
</tbody>
</table>

### More About

#### Golden Section Search

The golden section search `srchgol` is a linear search that does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance function occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of `delta` and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determine a section of the interval that can be discarded, and a new interior point is placed within the new interval. This procedure is continued until the interval of uncertainty is reduced to a width of `tol`, which is equal to `delta/scale_tol`.

See [HDB96], starting on page 12-16, for a complete description of the golden section search. Try the Neural Network Design demonstration `nnd12sd1` [HDB96] for an illustration of the performance of the golden section search in combination with a conjugate gradient algorithm.
Algorithms

srchgol locates the minimum of the performance function in the search direction $dX$, using the golden section search. It is based on the algorithm as described on page 33 of Scales (see reference below).

References


See Also

srchbac | srchbre | srchcha | srchhyb

Introduced before R2006a
srchhyb

1-D minimization using a hybrid bisection-cubic search

Syntax

[a,gX,perf,retcode,delta,tol] = srchhyb(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)

Description

srchhyb is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique that is a combination of a bisection and a cubic interpolation.

[a,gX,perf,retcode,delta,tol] = srchhyb(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf) takes these inputs,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Vector containing current values of weights and biases</td>
</tr>
<tr>
<td>Pd</td>
<td>Delayed input vectors</td>
</tr>
<tr>
<td>Tl</td>
<td>Layer target vectors</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial input delay conditions</td>
</tr>
<tr>
<td>Q</td>
<td>Batch size</td>
</tr>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
<tr>
<td>dX</td>
<td>Search direction vector</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient vector</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at current X</td>
</tr>
<tr>
<td>dperf</td>
<td>Slope of performance value at current X in direction of dX</td>
</tr>
<tr>
<td>delta</td>
<td>Initial step size</td>
</tr>
</tbody>
</table>
Tolerance on search

Change in performance on previous step

and returns

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Step size that minimizes performance</td>
</tr>
<tr>
<td>gX</td>
<td>Gradient at new minimum point</td>
</tr>
<tr>
<td>perf</td>
<td>Performance value at new minimum point</td>
</tr>
<tr>
<td>retcode</td>
<td>Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.</td>
</tr>
<tr>
<td>delta</td>
<td>New initial step size, based on the current step size</td>
</tr>
<tr>
<td>tol</td>
<td>New tolerance on search</td>
</tr>
</tbody>
</table>

Parameters used for the hybrid bisection-cubic algorithm are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>Scale factor that determines sufficient reduction in perf</td>
</tr>
<tr>
<td>beta</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td>bmax</td>
<td>Largest step size</td>
</tr>
<tr>
<td>scale_tol</td>
<td>Parameter that relates the tolerance tol to the initial step size delta, usually set to 20</td>
</tr>
</tbody>
</table>

The defaults for these parameters are set in the training function that calls them. See traincgf, traincgb, traincgp, trainbfg, and trainoss.

Dimensions for these variables are

1-459
<table>
<thead>
<tr>
<th>Pd</th>
<th>No-by-Ni-by-TS cell array</th>
<th>Each element $P_{i,j,ts}$ is a $D_{ij}$-by-$Q$ matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tl</td>
<td>Nl-by-TS cell array</td>
<td>Each element $P_{i,ts}$ is a $V_i$-by-$Q$ matrix.</td>
</tr>
<tr>
<td>Ai</td>
<td>Nl-by-LD cell array</td>
<td>Each element $A_{i,k}$ is an $S_i$-by-$Q$ matrix.</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th>Ni</th>
<th>=</th>
<th>net.numInputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nl</td>
<td>=</td>
<td>net.numLayers</td>
</tr>
<tr>
<td>LD</td>
<td>=</td>
<td>net.numLayerDelays</td>
</tr>
<tr>
<td>Ri</td>
<td>=</td>
<td>net.inputs{i}.size</td>
</tr>
<tr>
<td>Si</td>
<td>=</td>
<td>net.layers{i}.size</td>
</tr>
<tr>
<td>Vi</td>
<td>=</td>
<td>net.targets{i}.size</td>
</tr>
<tr>
<td>$D_{ij}$</td>
<td>=</td>
<td>$R_i \ast \text{length}(\text{net.inputWeights}{i,j}.\text{delays})$</td>
</tr>
</tbody>
</table>

**More About**

**Hybrid Bisection Cubic Search**

Like Brent’s search, `srchhyb` is a hybrid algorithm. It is a combination of bisection and cubic interpolation. For the bisection algorithm, one point is located in the interval of uncertainty, and the performance and its derivative are computed. Based on this information, half of the interval of uncertainty is discarded. In the hybrid algorithm, a cubic interpolation of the function is obtained by using the value of the performance and its derivative at the two endpoints. If the minimum of the cubic interpolation falls within the known interval of uncertainty, then it is used to reduce the interval of uncertainty. Otherwise, a step of the bisection algorithm is used.

See [Scal85] for a complete description of the hybrid bisection-cubic search. This algorithm does require derivative information, so it performs more computations at each step of the algorithm than the golden section search or Brent’s algorithm.
Algorithms

srchhyb locates the minimum of the performance function in the search direction dX, using the hybrid bisection-cubic interpolation algorithm described on page 50 of Scales (see reference below).

References

Scales, L.E., Introduction to Non-Linear Optimization, New York Springer-Verlag, 1985

See Also

srchbac | srchbre | srchcha | srchgol

Introduced before R2006a
sse

Sum squared error performance function

Syntax

perf = sse(net,t,y,ew)
[...] = sse(...,'regularization',regularization)
[...] = sse(...,'normalization',normalization)
[...] = sse(...,'squaredWeighting',squaredWeighting)
[...] = sse(...,FP)

Description

sse is a network performance function. It measures performance according to the sum of
squared errors.

perf = sse(net,t,y,ew) takes these input arguments and optional function
parameters,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Matrix or cell array of target vectors</td>
</tr>
<tr>
<td>y</td>
<td>Matrix or cell array of output vectors</td>
</tr>
<tr>
<td>ew</td>
<td>Error weights (default = {1})</td>
</tr>
</tbody>
</table>

and returns the sum squared error.

This function has three optional function parameters which can be defined with
parameter name/pair arguments, or as a structure FP argument with fields having the
parameter name and assigned the parameter values.

[...] = sse(...,'regularization',regularization)
[...] = sse(...,'normalization',normalization)
[...] = sse(...,'squaredWeighting',squaredWeighting)
[...] = sse(...,FP)

- **regularization** — can be set to any value between the default of 0 and 1. The greater the regularization value, the more squared weights and biases are taken into account in the performance calculation.

- **normalization** — can be set to the default 'absolute', or 'normalized' (which normalizes errors to the [+2 -2] range consistent with normalized output and target ranges of [-1 1]) or 'percent' (which normalizes errors to the range [-1 +1]).

- **squaredWeighting** — can be set to the default true, for applying error weights to squared errors; or false for applying error weights to the absolute errors before squaring.

**Examples**

Here a network is trained to fit a simple data set and its performance calculated

```matlab
[x,t] = simplefit_dataset;
net = fitnet(10);
net.performFcn = 'sse';
net = train(net,x,t)
y = net(x)
e = t-y
perf = sse(net,t,y)
```

**Network Use**

To prepare a custom network to be trained with sse, set net.performFcn to 'sse'. This automatically sets net.performParam to the default function parameters.

Then calling train, adapt or perform will result in sse being used to calculate performance.

*Introduced before R2006a*
**staticderiv**

Static derivative function

**Syntax**

```plaintext
staticderiv('dperf_dwb', net, X, T, Xi, Ai, EW)
staticderiv('de_dwb', net, X, T, Xi, Ai, EW)
```

**Description**

This function calculates derivatives using the chain rule from the networks performance or outputs back to its inputs. For time series data and dynamic networks this function ignores the delay connections resulting in a approximation (which may be good or not) of the actual derivative. This function is used by Elman networks (elmannet) which is a dynamic network trained by the static derivative approximation when full derivative calculations are not available. As full derivatives are calculated by all the other derivative functions, this function is not recommended for dynamic networks except for research into training algorithms.

`staticderiv('dperf_dwb', net, X, T, Xi, Ai, EW)` takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>Neural network</td>
</tr>
<tr>
<td>X</td>
<td>Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)</td>
</tr>
<tr>
<td>T</td>
<td>Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)</td>
</tr>
<tr>
<td>Xi</td>
<td>Initial input delay states (optional)</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay states (optional)</td>
</tr>
<tr>
<td>EW</td>
<td>Error weights (optional)</td>
</tr>
</tbody>
</table>

and returns the gradient of performance with respect to the network’s weights and biases, where R and S are the number of input and output elements and Q is the number of samples (and N and M are the number of input and output signals, Ri and Si are the number of each input and outputs elements, and TS is the number of timesteps).
staticderiv('de_dwb',net,X,T,Xi,Ai,EW) returns the Jacobian of errors with respect to the network’s weights and biases.

Examples

Here a feedforward network is trained and both the gradient and Jacobian are calculated.

[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y);
gwb = staticderiv('dperf_dwb',net,x,t)
jwb = staticderiv('de_dwb',net,x,t)

See Also

bttderiv | defaultderiv | fpderiv | num2deriv

Introduced in R2010b
sumabs

Sum of absolute elements of matrix or matrices

Syntax

[s,n] = sumabs(x)

Description

[s,n] = sumabs(x) takes a matrix or cell array of matrices and returns,

<table>
<thead>
<tr>
<th>s</th>
<th>Sum of all absolute finite values</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of finite values</td>
</tr>
</tbody>
</table>

If x contains no finite values, the sum returned is 0.

Examples

m = sumabs([1 2; 3 4])
[m,n] = sumabs({[1 2; NaN 4], [4 5; 2 3]})

See Also

meanabs | meansqr | sumsqr

Introduced in R2010b
**sumsqr**

Sum of squared elements of matrix or matrices

**Syntax**

\[ [s,n] = \text{sumsqr}(x) \]

**Description**

\[ [s,n] = \text{sumsqr}(x) \] takes a matrix or cell array of matrices and returns,

<table>
<thead>
<tr>
<th>s</th>
<th>Sum of all squared finite values</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of finite values</td>
</tr>
</tbody>
</table>

If \( x \) contains no finite values, the sum returned is 0.

**Examples**

\[
m = \text{sumsqr}([1 \ 2; 3 \ 4])
\]
\[
[m,n] = \text{sumsqr}({[1 \ 2; \text{NaN} \ 4], [4 \ 5; 2 \ 3]})
\]

**See Also**

meanabs | meansqr | sumabs

**Introduced before R2006a**
**tansig**

Hyperbolic tangent sigmoid transfer function

**Graph and Symbol**

![Graph of tansig function]

\[ a = \tansig(n) \]

Tan-Sigmoid Transfer Function

**Syntax**

\[ A = \tansig(N,FP) \]

**Description**

tansig is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \tansig(N,FP) \] takes \( N \) and optional function parameters,

<table>
<thead>
<tr>
<th>N</th>
<th>( S )-by-( Q ) matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \( A \), the \( S \)-by-\( Q \) matrix of \( N \)'s elements squashed into \([-1 \ 1]\).
Examples

Here is the code to create a plot of the tansig transfer function.

```matlab
n = -5:0.1:5;
a = tansig(n);
plot(n,a)
```

Assign this transfer function to layer i of a network.

```matlab
net.layers{i}.transferFcn = 'tansig';
```

Algorithms

```matlab
a = tansig(n) = 2/(1+exp(-2*n))-1
```

This is mathematically equivalent to \( \tanh(N) \). It differs in that it runs faster than the MATLAB implementation of \( \tanh \), but the results can have very small numerical differences. This function is a good tradeoff for neural networks, where speed is important and the exact shape of the transfer function is not.

References


See Also

logsig | sim

Introduced before R2006a
tapdelay

Shift neural network time series data for tap delay

Syntax

tapdelay(x,i,ts,delays)

Description

tapdelay(x,i,ts,delays) takes these arguments,

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Neural network time series data</td>
</tr>
<tr>
<td>i</td>
<td>Signal index</td>
</tr>
<tr>
<td>ts</td>
<td>Timestep index</td>
</tr>
<tr>
<td>delays</td>
<td>Row vector of increasing zero or positive delays</td>
</tr>
</tbody>
</table>

and returns the tap delay values of signal i at timestep ts given the specified tap delays.

Examples

Here a random signal x consisting of eight timesteps is defined, and a tap delay with delays of [0 1 4] is simulated at timestep 6.

```matlab
x = num2cell(rand(1,8));
y = tapdelay(x,1,6,[0 1 4])
```

See Also

extendts | nndata | preparets

Introduced in R2010b
timedelaynet

Time delay neural network

Syntax

timedelaynet(inputDelays,hiddenSizes,trainFcn)

Description

Time delay networks are similar to feedforward networks, except that the input weight has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the distributed delay neural network (distdelaynet), which has delays on the layer weights in addition to the input weight.

timedelaynet(inputDelays,hiddenSizes,trainFcn) takes these arguments,

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputDelays</td>
<td>Row vector of increasing 0 or positive delays (default = 1:2)</td>
</tr>
<tr>
<td>hiddenSizes</td>
<td>Row vector of one or more hidden layer sizes (default = 10)</td>
</tr>
<tr>
<td>trainFcn</td>
<td>Training function (default = 'trainlm')</td>
</tr>
</tbody>
</table>

and returns a time delay neural network.

Examples

Train Time Delay Network and Predict on New Data

Partition the training set. Use Xnew to do prediction in closed loop mode later.

[X,T] = simpleseries_dataset;
Xnew = X(81:100);
X = X(1:80);
T = T(1:80);
Train a time delay network, and simulate it on the first 80 observations.

```matlab
net = timedelaynet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```

Calculate the network performance.

```matlab
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y);
```

Run the prediction for 20 timesteps ahead in closed loop mode.

```matlab
[netc,Xic,Aic] = closeloop(net,Xf,Af);
view(netc)
```
y2 = netc(Xnew, Xic, Aic);

**See Also**
distdelaynet | narnet | narxnet | prepares | removedelay

*Introduced in R2010b*
tonndata

Convert data to standard neural network cell array form

Syntax

\[ [y, \text{wasMatrix}] = \text{tonndata}(x, \text{columnSamples}, \text{cellTime}) \]

Description

\[ [y, \text{wasMatrix}] = \text{tonndata}(x, \text{columnSamples}, \text{cellTime}) \]

takes these arguments,

+ \( x \) — Matrix or cell array of matrices
+ \( \text{columnSamples} \) — True if original samples are oriented as columns, false if rows
+ \( \text{cellTime} \) — True if original samples are columns of a cell array, false if they are stored in a matrix

and returns

+ \( y \) — Original data transformed into standard neural network cell array form
+ \( \text{wasMatrix} \) — True if original data was a matrix (as opposed to cell array)

If \( \text{columnSamples} \) is false, then matrix \( x \) or matrices in cell array \( x \) will be transposed, so row samples will now be stored as column vectors.

If \( \text{cellTime} \) is false, then matrix samples will be separated into columns of a cell array so time originally represented as vectors in a matrix will now be represented as columns of a cell array.

The returned value \( \text{wasMatrix} \) can be used by fromnndata to reverse the transformation.
Examples

Here data consisting of six timesteps of 5-element vectors, originally represented as a matrix with six columns, is converted to standard neural network representation and back.

```matlab
x = rands(5,6)
columnSamples = true; % samples are by columns.
cellTime = false; % time-steps in matrix, not cell array.
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
x2 = fromnndata(y,wasMatrix,columnSamples,cellTime)
```

See Also
fromnndata | nndata | nndata2sim | sim2nndata

Introduced in R2010b
train

Train shallow neural network

Syntax

trainedNet = train(net,X,T,Xi,Ai,EW)
[trainedNet,tr] = train(net,X,T,Xi,Ai,EW)
[trainedNet,tr] = train(net,X,T,Xi,Ai,EW,Name,Value)

Description

This function trains a shallow neural network. For deep learning with convolutional or LSTM neural networks, see trainNetwork instead.

trainedNet = train(net,X,T,Xi,Ai,EW) trains a network net according to net.trainFcn and net.trainParam.

[trainedNet,tr] = train(net,X,T,Xi,Ai,EW) also returns a training record.

[trainedNet,tr] = train(net,X,T,Xi,Ai,EW,Name,Value) trains a network with additional options specified by one or more name-value pair arguments.

Examples

Train and Plot Networks

Here input x and targets t define a simple function that you can plot:

```matlab
x = [0 1 2 3 4 5 6 7 8];
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28 0.66 0.99];
plot(x,t,'o')
```

Here feedforwardnet creates a two-layer feed-forward network. The network has one hidden layer with ten neurons.
net = feedforwardnet(10);
net = configure(net,x,t);
y1 = net(x)
plot(x,t,'o',x,y1,'x')

The network is trained and then resimulated.

net = train(net,x,t);
y2 = net(x)
plot(x,t,'o',x,y1,'x',x,y2,'*')

**Train NARX Time Series Network**

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current x and the magnet’s vertical position response t, then simulates the network. The function preparets prepares the data before training and simulation. It creates the open-loop network’s combined inputs xo, which contains both the external input x and previous values of position t. It also prepares the delay states xi.

[x,t] = maglev_dataset;
net = narxnet(10);
[xo,xi,~,to] = preparets(net,x,{},t);
net = train(net,xo,to,xi);
y = net(xo,xi)

This same system can also be simulated in closed-loop form.

netc = closeloop(net);
view(netc)
[xc,xi,ai,tc] = preparets(netc,x,{},t);
yc = netc(xc,xi,ai);

**Train a Network in Parallel on a Parallel Pool**

Parallel Computing Toolbox allows Deep Learning Toolbox to simulate and train networks faster and on larger datasets than can fit on one PC. Parallel training is currently supported for backpropagation training only, not for self-organizing maps.

Here training and simulation happens across parallel MATLAB workers.
parpool
[X,T] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,X,T,'useParallel','yes','showResources','yes');
Y = net(X);

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded as it is distributed then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

[X,T] = vinyl_dataset;
Q = size(X,2);
Xc = Composite;
Tc = Composite;
umWorkers = numel(Xc);
ind = [0 ceil((1:numWorkers)*(Q/numWorkers))];
for i=1:numWorkers
   indi = (ind(i)+1):ind(i+1);
   Xc{i} = X(:,indi);
   Tc{i} = T(:,indi);
end
net = feedforwardnet;
net = configure(net,X,T);
net = train(net,Xc,Tc);
Yc = net(Xc);

Note in the example above the function configure was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when train is called, but when providing composite data this step must be done manually with non-Composite data.

**Train a Network on GPUs**

Networks can be trained using the current GPU device, if it is supported by Parallel Computing Toolbox. GPU training is currently supported for backpropagation training only, not for self-organizing maps.

[X,T] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,X,T,'useGPU','yes');
y = net(X);
To put the data on a GPU manually:

```matlab
[X,T] = vinyl_dataset;
Xgpu = gpuArray(X);
Tgpu = gpuArray(T);
net = configure(net,X,T);
net = train(net,Xgpu,Tgpu);
Ygpu = net(Xgpu);
Y = gather(Ygpu);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when `train` is called, but when providing `gpuArray` data this step must be done manually with non-`gpuArray` data.

To run in parallel, with workers each assigned to a different unique GPU, with extra workers running on CPU:

```matlab
net = train(net,X,T,'useParallel','yes','useGPU','yes');
y = net(X);
```

Using only workers with unique GPUs might result in higher speed, as CPU workers might not keep up.

```matlab
net = train(net,X,T,'useParallel','yes','useGPU','only');
Y = net(X);
```

**Train Network Using Checkpoint Saves**

Here a network is trained with checkpoints saved at a rate no greater than once every two minutes.

```matlab
[x,t] = vinyl_dataset;
net = fitnet([60 30]);
net = train(net,x,t,'CheckpointFile','MyCheckpoint','CheckpointDelay',120);
```

After a computer failure, the latest network can be recovered and used to continue training from the point of failure. The checkpoint file includes a structure variable `checkpoint`, which includes the network, training record, filename, time, and number.

```matlab
[x,t] = vinyl_dataset;
load MyCheckpoint
net = checkpoint.net;
net = train(net,x,t,'CheckpointFile','MyCheckpoint');
```
Another use for the checkpoint feature is when you stop a parallel training session (started with the 'UseParallel' parameter) even though the Neural Network Training Tool is not available during parallel training. In this case, set a 'CheckpointFile', use Ctrl+C to stop training any time, then load your checkpoint file to get the network and training record.

**Input Arguments**

**net — Input network**

*network object*

Input network, specified as a *network object*. To create a *network object*, use for example, *feedforwardnet* or *narxnet*.

**X — Network inputs**

*matrix | cell array | composite data | gpuArray*

Network inputs, specified as an R-by-Q matrix or an Ni-by-TS cell array, where

- R is the input size
- Q is the batch size
- Ni = `net.numInputs`
- TS is the number of time steps

*train* arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

- The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can be used with networks that have more. When the network has multiple inputs, the matrix size is (sum of Ri)-by-Q.
- The cell array format is more general, and more convenient for networks with multiple inputs and outputs, allowing sequences of inputs to be presented. Each element \( X\{i,ts\} \) is an Ri-by-Q matrix, where \( Ri = net.inputs\{i\}.size \).

If Composite data is used, then 'useParallel' is automatically set to 'yes'. The function takes Composite data and returns Composite results.
If gpuArray data is used, then 'useGPU' is automatically set to 'yes'. The function takes gpuArray data and returns gpuArray results.

**Note** If a column of X contains at least one NaN, train does not use that column for training, testing, or validation. If a target value in T is a NaN, then train ignores that row, and uses the other rows for training, testing, or validation.

### T — Network targets

*zeros (default) | matrix | cell array | composite data | gpuArray*

Network targets, specified as a U-by-Q matrix or an No-by-TS cell array, where

- U is the output size
- Q is the batch size
- No = net.numOutputs
- TS is the number of time steps

train arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

- The matrix format can be used if only one time step is to be simulated (TS = 1). It is convenient for networks with only one input and output, but can be used with networks that have more. When the network has multiple inputs, the matrix size is (sum of U)-by-Q.
- The cell array format is more general, and more convenient for networks with multiple inputs and outputs, allowing sequences of inputs to be presented. Each element T{i,ts} is a Ui-by-Q matrix, where Ui = net.outputs{i}.size.

If Composite data is used, then 'useParallel' is automatically set to 'yes'. The function takes Composite data and returns Composite results.

If gpuArray data is used, then 'useGPU' is automatically set to 'yes'. The function takes gpuArray data and returns gpuArray results.

Note that T is optional and need only be used for networks that require targets.
**Note** Any NaN values in the inputs $X$ or the targets $T$, are treated as missing data. If a column of $X$ or $T$ contains at least one NaN, that column is not used for training, testing, or validation.

**Xi — Initial input delay conditions**
zeros (default) | cell array | matrix

Initial input delay conditions, specified as an $Ni$-by-$ID$ cell array or an $R$-by-$(ID*Q)$ matrix, where

- $ID = net.numInputDelays$
- $Ni = net.numInputs$
- $R$ is the input size
- $Q$ is the batch size

For cell array input, the columns of $Xi$ are ordered from the oldest delay condition to the most recent: $Xi[i,k]$ is the input $i$ at time $ts = k - ID$.

$Xi$ is also optional and need only be used for networks that have input or layer delays.

**Ai — Initial layer delay conditions**
zeros (default) | cell array | matrix

Initial layer delay conditions, specified as a $Nl$-by-$LD$ cell array or a (sum of $Si$)-by-$(LD*Q)$ matrix, where

- $Nl = net.numLayers$
- $LD = net.numLayerDelays$
- $Si = net.layers{i}.size$
- $Q$ is the batch size

For cell array input, the columns of $Ai$ are ordered from the oldest delay condition to the most recent: $Ai[i,k]$ is the layer output $i$ at time $ts = k - LD$.

**EW — Error weights**
cell array

Error weights, specified as a $No$-by-$TS$ cell array or a (sum of $Ui$)-by-$Q$ matrix, where
• No = net.numOutputs
• TS is the number of time steps
• Ui = net.outputs{i}.size
• Q is the batch size

For cell array input, each element EW{i,ts} is a Ui-by-Q matrix, where
• Ui = net.outputs{i}.size
• Q is the batch size

The error weights EW can also have a size of 1 in place of all or any of No, TS, Ui or Q. In that case, EW is automatically dimension extended to match the targets T. This allows for conveniently weighting the importance in any dimension (such as per sample) while having equal importance across another (such as time, with TS=1). If all dimensions are 1, for instance if EW = {1}, then all target values are treated with the same importance. That is the default value of EW.

As noted above, the error weights EW can be of the same dimensions as the targets T, or have some dimensions set to 1. For instance if EW is 1-by-Q, then target samples will have different importances, but each element in a sample will have the same importance. If EW is (sum of Ui)-by-Q, then each output element has a different importance, with all samples treated with the same importance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'useParallel','yes'

useParallel — Option to specify parallel calculations
'no' (default) | 'yes'

Option to specify parallel calculations, specified as 'yes' or 'no'.

• 'no' - Calculations occur on normal MATLAB thread. This is the default 'useParallel' setting.
• 'yes' - Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.

**useGPU — Option to specify GPU calculations**

'no' (default) | 'yes' | 'only'

Option to specify GPU calculations, specified as 'yes', 'no', or 'only'.

- 'no' - Calculations occur on the CPU. This is the default 'useGPU' setting.
- 'yes' - Calculations occur on the current gpuDevice if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current gpuDevice is not supported, calculations remain on the CPU. If 'useParallel' is also 'yes' and a parallel pool is open, then each worker with a unique GPU uses that GPU, other workers run calculations on their respective CPU cores.
- 'only' - If no parallel pool is open, then this setting is the same as 'yes'. If a parallel pool is open then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.

**showResources — Option to show resources**

'no' (default) | 'yes'

Option to show resources, specified as 'yes' or 'no'.

- 'no' - Do not display computing resources used at the command line. This is the default setting.
- 'yes' - Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker’s computation mode is described, including workers in the pool that are not used.

**reduction — Memory reduction**

1 (default) | positive integer

Memory reduction, specified as a positive integer.

For most neural networks, the default CPU training computation mode is a compiled MEX algorithm. However, for large networks the calculations might occur with a MATLAB calculation mode. This can be confirmed using 'showResources'. If MATLAB is being used and memory is an issue, setting the reduction option to a value N greater than 1,
reduces much of the temporary storage required to train by a factor of N, in exchange for longer training times.

**CheckpointFile — Checkpoint file**

' ' (default) | character vector

Checkpoint file, specified as a character vector.

The value for 'CheckpointFile' can be set to a filename to save in the current working folder, to a file path in another folder, or to an empty string to disable checkpoint saves (the default value).

**CheckpointDelay — Checkpoint delay**

60 (default) | nonnegative integer

Checkpoint delay, specified as a nonnegative integer.

The optional parameter 'CheckpointDelay' limits how often saves happen. Limiting the frequency of checkpoints can improve efficiency by keeping the amount of time saving checkpoints low compared to the time spent in calculations. It has a default value of 60, which means that checkpoint saves do not happen more than once per minute. Set the value of 'CheckpointDelay' to 0 if you want checkpoint saves to occur only once every epoch.

**Output Arguments**

**trainedNet — Trained network**

network object

Trained network, returned as a network object.

**tr — Training record**

structure

Training record (epoch and perf), returned as a structure whose fields depend on the network training function (net.NET.trainFcn). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets
• Data division masks for training validation and test sets
• Number of epochs (num_epochs) and the best epoch (best_epoch).
• A list of training state names (states).
• Fields for each state name recording its value throughout training
• Performances of the best network (best_perf, best_vperf, best_tperf)

**Algorithms**

train calls the function indicated by net.trainFcn, using the training parameter values indicated by net.trainParam.

Typically one epoch of training is defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of the function net.trainFcn occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly for each epoch from concurrent input vectors (or sequences). competlayer returns networks that use trainru, a training function that does this.

**See Also**

adapt | init | revert | sim

**Introduced before R2006a**
**trainb**

Batch training with weight and bias learning rules

**Syntax**

```matlab
net.trainFcn = 'trainb'
[net,tr] = train(net,...)
```

**Description**

`trainb` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'trainb'`, thus:

```matlab
net.trainFcn = 'trainb' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainb.
```

`trainb` trains a network with weight and bias learning rules with batch updates. The weights and biases are updated at the end of an entire pass through the input data.

Training occurs according to `trainb`'s training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-6</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses trainb by calling linearlayer.

To prepare a custom network to be trained with trainb,

1  Set net.trainFcn to 'trainb'. This sets net.trainParam to trainb's default parameters.

2  Set each net.inputWeights{i,j}.learnFcn to a learning function. Set each net.layerWeights{i,j}.learnFcn to a learning function. Set each net.biases{i}.learnFcn to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1  Set net.trainParam properties to desired values.

2  Set weight and bias learning parameters to desired values.

3  Call train.

Algorithms

Each weight and bias is updated according to its learning function after each epoch (one pass through the entire set of input vectors).

Training stops when any of these conditions is met:

• The maximum number of epochs (repetitions) is reached.
• Performance is minimized to the goal.
• The maximum amount of time is exceeded.
• Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

See Also

linearlayer | train
Introduced before R2006a
**trainbfg**

BFGS quasi-Newton backpropagation

**Syntax**

```matlab
net.trainFcn = 'trainbfg'
[net,tr] = train(net,...)
```

**Description**

`trainbfg` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method.

`net.trainFcn = 'trainbfg'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainbfg`.

Training occurs according to `trainbfg` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training window</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-6</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.searchFcn</td>
<td>'srchbac'</td>
<td>Name of line search routine to use</td>
</tr>
</tbody>
</table>

Parameters related to line search methods (not all used for all methods):

1-490
### Network Use

You can create a standard network that uses `trainbfg` with `feedfowardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainbfg`:

1. Set `NET.trainFcn` to `'trainbfg'`. This sets `NET.trainParam` to `trainbfg`'s default parameters.
2. Set `NET.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainbfg`.

### Examples

<table>
<thead>
<tr>
<th>net.trainParam.scal_tol</th>
<th>20</th>
<th>Divide into <code>delta</code> to determine tolerance for linear search.</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.alpha</td>
<td>0.001</td>
<td>Scale factor that determines sufficient reduction in perf</td>
</tr>
<tr>
<td>net.trainParam.beta</td>
<td>0.1</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td>net.trainParam.delta</td>
<td>0.01</td>
<td>Initial step size in interval location step</td>
</tr>
<tr>
<td>net.trainParam.gama</td>
<td>0.1</td>
<td>Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code>)</td>
</tr>
<tr>
<td>net.trainParam.low_lim</td>
<td>0.1</td>
<td>Lower limit on change in step size</td>
</tr>
<tr>
<td>net.trainParam.up_lim</td>
<td>0.5</td>
<td>Upper limit on change in step size</td>
</tr>
<tr>
<td>net.trainParam.maxstep</td>
<td>100</td>
<td>Maximum step length</td>
</tr>
<tr>
<td>net.trainParam.minstep</td>
<td>1.0e-6</td>
<td>Minimum step length</td>
</tr>
<tr>
<td>net.trainParam.bmax</td>
<td>26</td>
<td>Maximum step size</td>
</tr>
<tr>
<td>net.trainParam.batch_frag</td>
<td>0</td>
<td>In case of multiple batches, they are considered independent. Any nonzero value implies a fragmented batch, so the final layer's conditions of a previous trained epoch are used as initial conditions for the next epoch.</td>
</tr>
</tbody>
</table>
Train Neural Network Using \texttt{trainbfg} Train Function

This example shows how to train a neural network using the \texttt{trainbfg} train function.

Here a neural network is trained to predict body fat percentages.

\begin{verbatim}
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainbfg');
net = train(net, x, t);
y = net(x);
\end{verbatim}

More About

BFGS Quasi-Newton Backpropagation

Newton’s method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton’s method is

\[ x_{k+1} = x_k - A_k^{-1}g_k \]

where \( A_k^{-1} \) is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton’s method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that is based on Newton’s method, but which does not require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm is implemented in the \texttt{trainbfg} routine.

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is \( n \times n \), where \( n \) is equal to the number of weights and biases in the network. For very large networks it might be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, \texttt{trainbfg} can be an efficient training function.
**Algorithms**

`trainbfg` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

\[ X = X + a \cdot dX; \]

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

\[ dX = -H \cdot gX; \]

where `gX` is the gradient and `H` is a approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (*Practical Optimization*, 1981) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

**References**


**See Also**

cascadeforwardnet | feedforwardnet | traincgb | traincfgf | traincgp | traingda | traingdm | traingdx | trainlm | trainoss | trainrp | trainscg
Introduced before R2006a
trainbfgc

BFGS quasi-Newton backpropagation for use with NN model reference adaptive controller

**Syntax**

\[
[\text{net}, \text{TR}, \text{Y}, \text{E}, \text{Pf}, \text{Af}, \text{flag\_stop}] = \text{trainbfgc}(\text{net}, \text{P}, \text{T}, \text{Pi}, \text{Ai}, \text{epochs}, \text{TS}, \text{Q})
\]

\[
\text{info} = \text{trainbfgc}(\text{code})
\]

**Description**

\text{trainbfgc} is a network training function that updates weight and bias values according to the BFGS quasi-Newton method. This function is called from \text{nnmodref}, a GUI for the model reference adaptive control Simulink block.

\[
[\text{net}, \text{TR}, \text{Y}, \text{E}, \text{Pf}, \text{Af}, \text{flag\_stop}] = \text{trainbfgc}(\text{net}, \text{P}, \text{T}, \text{Pi}, \text{Ai}, \text{epochs}, \text{TS}, \text{Q})
\]

takes these inputs,

<table>
<thead>
<tr>
<th>net</th>
<th>Neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Delayed input vectors</td>
</tr>
<tr>
<td>T</td>
<td>Layer target vectors</td>
</tr>
<tr>
<td>Pi</td>
<td>Initial input delay conditions</td>
</tr>
<tr>
<td>Ai</td>
<td>Initial layer delay conditions</td>
</tr>
<tr>
<td>epochs</td>
<td>Number of iterations for training</td>
</tr>
<tr>
<td>TS</td>
<td>Time steps</td>
</tr>
<tr>
<td>Q</td>
<td>Batch size</td>
</tr>
</tbody>
</table>

and returns

<table>
<thead>
<tr>
<th>net</th>
<th>Trained network</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR</td>
<td>Training record of various values over each epoch:</td>
</tr>
</tbody>
</table>

1-495
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR.epoch</td>
<td>Epoch number</td>
</tr>
<tr>
<td>TR.perf</td>
<td>Training performance</td>
</tr>
<tr>
<td>TR.vperf</td>
<td>Validation performance</td>
</tr>
<tr>
<td>TR.tperf</td>
<td>Test performance</td>
</tr>
<tr>
<td>Y</td>
<td>Network output for last epoch</td>
</tr>
<tr>
<td>E</td>
<td>Layer errors for last epoch</td>
</tr>
<tr>
<td>Pf</td>
<td>Final input delay conditions</td>
</tr>
<tr>
<td>Af</td>
<td>Collective layer outputs for last epoch</td>
</tr>
<tr>
<td>flag_stop</td>
<td>Indicates if the user stopped the training</td>
</tr>
</tbody>
</table>

Training occurs according to `trainbfgc`'s training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>100</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-6</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>5</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.searchFcn</td>
<td>'srchbac x'</td>
<td>Name of line search routine to use</td>
</tr>
</tbody>
</table>

Parameters related to line search methods (not all used for all methods):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.scal_tol</td>
<td>20</td>
<td>Divide into delta to determine tolerance for linear search.</td>
</tr>
<tr>
<td>net.trainParam.alpha</td>
<td>0.001</td>
<td>Scale factor that determines sufficient reduction in perf</td>
</tr>
<tr>
<td>net.trainParam.beta</td>
<td>0.1</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td>net.trainParam.delta</td>
<td>0.01</td>
<td>Initial step size in interval location step</td>
</tr>
<tr>
<td>net.trainParam.gama</td>
<td>0.1</td>
<td>Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>net.trainParam.low_lim</td>
<td>0.1</td>
<td>Lower limit on change in step size</td>
</tr>
<tr>
<td>net.trainParam.up_lim</td>
<td>0.5</td>
<td>Upper limit on change in step size</td>
</tr>
<tr>
<td>net.trainParam.maxstep</td>
<td>100</td>
<td>Maximum step length</td>
</tr>
<tr>
<td>net.trainParam.minstep</td>
<td>1.0e-6</td>
<td>Minimum step length</td>
</tr>
<tr>
<td>net.trainParam.bmax</td>
<td>26</td>
<td>Maximum step size</td>
</tr>
</tbody>
</table>

\[ \text{info} = \text{trainbfgc}(\text{code}) \] returns useful information for each \text{code} character vector:

<table>
<thead>
<tr>
<th>'pnames'</th>
<th>Names of training parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdefaults'</td>
<td>Default training parameters</td>
</tr>
</tbody>
</table>

### Algoritms

\text{trainbfgc} can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance \( \text{perf} \) with respect to the weight and bias variables \( X \). Each variable is adjusted according to the following:

\[
X = X + a \cdot dX;
\]

where \( dX \) is the search direction. The parameter \( a \) is selected to minimize the performance along the search direction. The line search function \text{searchFcn} is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

\[
dX = -H \cdot gX;
\]

where \( gX \) is the gradient and \( H \) is an approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (\textit{Practical Optimization}, 1981) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
• The performance gradient falls below \texttt{min_grad}.
• Precision problems have occurred in the matrix inversion.

References


\textbf{Introduced in R2006a}
trainbr

Bayesian regularization backpropagation

**Syntax**

```matlab
net.trainFcn = 'trainbr'
[net,tr] = train(net,...)
```

**Description**

`trainbr` is a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes well. The process is called Bayesian regularization.

`net.trainFcn = 'trainbr'` sets the network trainFcn property.

`[net,tr] = train(net,...)` trains the network with `trainbr`.

Training occurs according to `trainbr` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.mu</td>
<td>0.005</td>
<td>Marquardt adjustment parameter</td>
</tr>
<tr>
<td>net.trainParam.mu_dec</td>
<td>0.1</td>
<td>Decrease factor for <code>mu</code></td>
</tr>
<tr>
<td>net.trainParam.mu_inc</td>
<td>10</td>
<td>Increase factor for <code>mu</code></td>
</tr>
<tr>
<td>net.trainParam.mu_max</td>
<td>1e10</td>
<td>Maximum value for <code>mu</code></td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>inf</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-7</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
</tbody>
</table>
Validation stops are disabled by default (max_fail = inf) so that training can continue until an optimal combination of errors and weights is found. However, some weight/bias minimization can still be achieved with shorter training times if validation is enabled by setting max_fail to 6 or some other strictly positive value.

Network Use

You can create a standard network that uses trainbr with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with trainbr,

1. Set NET.trainFcn to 'trainbr'. This sets NET.trainParam to trainbr’s default parameters.
2. Set NET.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with trainbr. See feedforwardnet and cascadeforwardnet for examples.

Examples

Here is a problem consisting of inputs p and targets t to be solved with a network. It involves fitting a noisy sine wave.

\[
p = [-1:.05:1];
t = \sin(2\pi p) + 0.1 \cdot \text{randn(size(p))};
\]

A feed-forward network is created with a hidden layer of 2 neurons.

net = feedforwardnet(2,'trainbr');

Here the network is trained and tested.

net = train(net,p,t);
a = net(p)
Limitations

This function uses the Jacobian for calculations, which assumes that performance is a mean or sum of squared errors. Therefore networks trained with this function must use either the `mse` or `sse` performance function.

Algorithms

`trainbr` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Bayesian regularization minimizes a linear combination of squared errors and weights. It also modifies the linear combination so that at the end of training the resulting network has good generalization qualities. See MacKay \( (Neural \ Computation, \ Vol. \ 4, \ No. \ 3, \ 1992, \ pp. \ 415 \ to \ 447) \) and Foresee and Hagan \( (Proceedings \ of \ the \ International \ Joint \ Conference \ on \ Neural \ Networks, \ June, \ 1997) \) for more detailed discussions of Bayesian regularization.

This Bayesian regularization takes place within the Levenberg-Marquardt algorithm. Backpropagation is used to calculate the Jacobian \( jX \) of performance \( perf \) with respect to the weight and bias variables \( X \). Each variable is adjusted according to Levenberg-Marquardt,

\[
jj = jX * jX \\
je = jX * E \\
dX = -(jj+I*mu) \backslash je
\]

where \( E \) is all errors and \( I \) is the identity matrix.

The adaptive value \( mu \) is increased by \( mu\_inc \) until the change shown above results in a reduced performance value. The change is then made to the network, and \( mu \) is decreased by \( mu\_dec \).

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below \( \text{min\_grad} \).
• mu exceeds mu_max.

References


See Also
cascadeforwardnet | feedforwardnet | trainbfg | traincgb | traincgf | traincgp | traingda | traingdm | traingdx | trainlm | trainrp | trainscg

Introduced before R2006a
trainbu

Batch unsupervised weight/bias training

Syntax

net.trainFcn = 'trainbu'
[net,tr] = train(net,...)

Description

trainbu trains a network with weight and bias learning rules with batch updates. Weights and biases updates occur at the end of an entire pass through the input data.

trainbu is not called directly. Instead the train function calls it for networks whose NET.trainFcn property is set to 'trainbu', thus:

net.trainFcn = 'trainbu' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainbu.

Training occurs according to trainbu training parameters, shown here with the following default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>

Validation and test vectors have no impact on training for this function, but act as independent measures of network generalization.
Network Use

You can create a standard network that uses trainbu by calling selforgmap. To prepare a custom network to be trained with trainbu:

1. Set NET.trainFcn to 'trainbu'. (This option sets NET.trainParam to trainbu default parameters.)
2. Set each NET.inputWeights{i,j}.learnFcn to a learning function.
3. Set each NET.layerWeights{i,j}.learnFcn to a learning function.
4. Set each NET.biases{i}.learnFcn to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network:

1. Set NET.trainParam properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call train.

See selforgmap for training examples.

Algorithms

Each weight and bias updates according to its learning function after each epoch (one pass through the entire set of input vectors).

Training stops when any of these conditions is met:

- The maximum number of epochs (repetitions) is reached.
- Performance is minimized to the goal.
- The maximum amount of time is exceeded.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

See Also

train | trainb
Introduced in R2010b
**trainc**

Cyclical order weight/bias training

**Syntax**

net.trainFcn = 'trainc'

[net,tr] = train(net,...)

**Description**

trainc is not called directly. Instead it is called by train for networks whose net.trainFcn property is set to 'trainc', thus:

net.trainFcn = 'trainc' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainc.

trainc trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in cyclic order.

Training occurs according to trainc training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value (default)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses `trainc` by calling `competlayer`. To prepare a custom network to be trained with `trainc`,

1. Set `net.trainFcn` to `'trainc'`. This sets `net.trainParam` to `trainc`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function. Set each `net.layerWeights{i,j}.learnFcn` to a learning function. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See `perceptron` for training examples.

Algorithms

For each epoch, each vector (or sequence) is presented in order to the network, with the weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of epochs (repetitions) is reached.
- Performance is minimized to the goal.
- The maximum amount of time is exceeded.

See Also

`competlayer` | `train`

Introduced before R2006a
traincgb

Conjugate gradient backpropagation with Powell-Beale restarts

Syntax

net.trainFcn = 'traincgb'
[net,tr] = train(net,...)

Description

traincgb is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Powell-Beale restarts.

net.trainFcn = 'traincgb' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with traincgb.

Training occurs according to traincgb training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-10</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.searchFcn</td>
<td>'srchc'</td>
<td>Name of line search routine to use</td>
</tr>
</tbody>
</table>

Parameters related to line search methods (not all used for all methods):
### Network Use

You can create a standard network that uses traincgb with feedforwardnet or cascadeforwardnet.

To prepare a custom network to be trained with traincgb,

1. Set `net.trainFcn` to `'traincgb'`. This sets `net.trainParam` to `traincgb`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincgb`.

### Examples

**Train Neural Network Using traincgb Train Function**

This example shows how to train a neural network using the `traincgb` train function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.scal_tol</code></td>
<td>20</td>
<td>Divide into <code>delta</code> to determine tolerance for linear search.</td>
</tr>
<tr>
<td><code>net.trainParam.alpha</code></td>
<td>0.001</td>
<td>Scale factor that determines sufficient reduction in <code>perf</code></td>
</tr>
<tr>
<td><code>net.trainParam.beta</code></td>
<td>0.1</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td><code>net.trainParam.delta</code></td>
<td>0.01</td>
<td>Initial step size in interval location step</td>
</tr>
<tr>
<td><code>net.trainParam.gama</code></td>
<td>0.1</td>
<td>Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code>)</td>
</tr>
<tr>
<td><code>net.trainParam.low_lim</code></td>
<td>0.1</td>
<td>Lower limit on change in step size</td>
</tr>
<tr>
<td><code>net.trainParam.up_lim</code></td>
<td>0.5</td>
<td>Upper limit on change in step size</td>
</tr>
<tr>
<td><code>net.trainParam.maxstep</code></td>
<td>100</td>
<td>Maximum step length</td>
</tr>
<tr>
<td><code>net.trainParam.minstep</code></td>
<td>1.0e-6</td>
<td>Minimum step length</td>
</tr>
<tr>
<td><code>net.trainParam.bmax</code></td>
<td>26</td>
<td>Maximum step size</td>
</tr>
</tbody>
</table>
Here a neural network is trained to predict body fat percentages.

```matlab
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgb');
net = train(net, x, t);
y = net(x);
```

## More About

### Powell-Beale Algorithm

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. This technique restarts if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality:

\[
|g_k^T - g_{k-1}| \geq 0.2\|g_k\|^2
\]

If this condition is satisfied, the search direction is reset to the negative of the gradient.

The `traincgb` routine has somewhat better performance than `traincgp` for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

## Algorithms

`traincgb` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

\[
X = X + a*dX;
\]
where $dX$ is the search direction. The parameter $a$ is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

$$dX = -gX + dX_{old}Z;$$

where $gX$ is the gradient. The parameter $Z$ can be computed in several different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. See Powell, *Mathematical Programming*, Vol. 12, 1977, pp. 241 to 254, for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below $\text{min}_\text{grad}$.
- Validation performance has increased more than $\text{max}_\text{fail}$ times since the last time it decreased (when using validation).

**References**


**See Also**

`trainbfg` | `traincgf` | `traincgp` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainscg`
traincgf

Conjugate gradient backpropagation with Fletcher-Reeves updates

Syntax

net.trainFcn = 'traincgf'
[net,tr] = train(net,...)

Description

traincgf is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Fletcher-Reeves updates.

net.trainFcn = 'traincgf' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with traincgf.

Training occurs according to traincgf training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-10</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.searchFcn</td>
<td>'srchch a'</td>
<td>Name of line search routine to use</td>
</tr>
</tbody>
</table>

Parameters related to line search methods (not all used for all methods):
### Network Use

You can create a standard network that uses `traincgf` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be trained with `traincgf`,

1. Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincgf`.

### Examples

**Train Neural Network Using `traincgf` Train Function**

This example shows how to train a neural network using the `traincgf` train function.
Here a neural network is trained to predict body fat percentages.

```matlab
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincfg');
net = train(net, x, t);
y = net(x);
```

### More About

#### Conjugate Gradient Algorithms

All the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

\[ p_0 = -g_0 \]

A line search is then performed to determine the optimal distance to move along the current search direction:

\[ x_{k+1} = x_k + \alpha_k p_k \]

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

\[ p_k = -g_k + \beta_k p_{k-1} \]

The various versions of the conjugate gradient algorithm are distinguished by the manner in which the constant \( \beta_k \) is computed. For the Fletcher-Reeves update the procedure is

\[ \beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}} \]

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.
The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than trainrp, although the results vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms. Therefore, these algorithms are good for networks with a large number of weights.

Try the Neural Network Design demonstration nnd12cg [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

**Algorithms**

traincfg can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

\[ X = X + a \cdot dX; \]

where \( dX \) is the search direction. The parameter \( a \) is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction, according to the formula

\[ dX = -gX + dX_{old} \cdot Z; \]

where \( gX \) is the gradient. The parameter \( Z \) can be computed in several different ways. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

\[ Z = \frac{\text{normnew}_sqr}{\text{norm}_sqr}; \]

where \( \text{norm}_sqr \) is the norm square of the previous gradient and \( \text{normnew}_sqr \) is the norm square of the current gradient. See page 78 of Scales (Introduction to Non-Linear Optimization) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
• Performance is minimized to the goal.
• The performance gradient falls below $\min_{\text{grad}}$.
• Validation performance has increased more than $\max_{\text{fail}}$ times since the last time it decreased (when using validation).

References


See Also

trainbfg | traincgb | traincgf | traingda | traingdm | traingdx | trainlm | trainoss | trainscg

Introduced before R2006a
**traincgp**

Conjugate gradient backpropagation with Polak-Ribiére updates

**Syntax**

```matlab
net.trainFcn = 'traincgp'
[net,tr] = train(net,...)
```

**Description**

`traincgp` is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Polak-Ribiére updates.

`net.trainFcn = 'traincgp'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traincgp`.

Training occurs according to `traincgp` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.epochs</code></td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td><code>net.trainParam.show</code></td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td><code>net.trainParam.showCommandLine</code></td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td><code>net.trainParam.showWindow</code></td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td><code>net.trainParam.goal</code></td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td><code>net.trainParam.time</code></td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td><code>net.trainParam.min_grad</code></td>
<td>1e-10</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td><code>net.trainParam.max_fail</code></td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td><code>net.trainParam.searchFcn</code></td>
<td>'srchcha'</td>
<td>Name of line search routine to use</td>
</tr>
</tbody>
</table>

Parameters related to line search methods (not all used for all methods):
Network Use

You can create a standard network that uses traincgp with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traincgp,

1. Set net.trainFcn to 'traincgp'. This sets net.trainParam to traincgp’s default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with traincgp.

Examples

Train Neural Network Using traincgp Train Function

This example shows how to train a neural network using the traincgp train function.

Here a neural network is trained to predict body fat percentages.

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.scal_tol</td>
<td>20</td>
<td>Divide into delta to determine tolerance for linear search.</td>
</tr>
<tr>
<td>net.trainParam.alpha</td>
<td>0.001</td>
<td>Scale factor that determines sufficient reduction in perf</td>
</tr>
<tr>
<td>net.trainParam.beta</td>
<td>0.1</td>
<td>Scale factor that determines sufficiently large step size</td>
</tr>
<tr>
<td>net.trainParam.delta</td>
<td>0.01</td>
<td>Initial step size in interval location step</td>
</tr>
<tr>
<td>net.trainParam.gama</td>
<td>0.1</td>
<td>Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)</td>
</tr>
<tr>
<td>net.trainParam.low_lim</td>
<td>0.1</td>
<td>Lower limit on change in step size</td>
</tr>
<tr>
<td>net.trainParam.up_lim</td>
<td>0.5</td>
<td>Upper limit on change in step size</td>
</tr>
<tr>
<td>net.trainParam.maxstep</td>
<td>100</td>
<td>Maximum step length</td>
</tr>
<tr>
<td>net.trainParam.minstep</td>
<td>1.0e-6</td>
<td>Minimum step length</td>
</tr>
<tr>
<td>net.trainParam.bmax</td>
<td>26</td>
<td>Maximum step size</td>
</tr>
</tbody>
</table>
More About

Conjugate Gradient Backpropagation with Polak-Ribiére Updates

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, \texttt{traincgf}, the search direction at each iteration is determined by

\[
p_k = -g_k + \beta_k p_{k-1}
\]

For the Polak-Ribiére update, the constant $\beta_k$ is computed by

\[
\beta_k = \frac{\Delta g_k^T g_k}{g_k^{T-1} g_k - 1}
\]

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

The \texttt{traincgp} routine has performance similar to \texttt{traincgf}. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

Algorithms

\texttt{traincgp} can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables $X$. Each variable is adjusted according to the following:
\[ X = X + a \cdot dX; \]

where \( dX \) is the search direction. The parameter \( a \) is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

\[ dX = -gX + dX_{\text{old}} \cdot Z; \]

where \( gX \) is the gradient. The parameter \( Z \) can be computed in several different ways. For the Polak-Ribière variation of conjugate gradient, it is computed according to

\[ Z = ((gX - gX_{\text{old}})' \cdot gX) / \text{norm\_sqr}; \]

where \( \text{norm\_sqr} \) is the norm square of the previous gradient, and \( gX_{\text{old}} \) is the gradient on the previous iteration. See page 78 of Scales (*Introduction to Non-Linear Optimization*, 1985) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below \( \text{min\_grad} \).
- Validation performance has increased more than \( \text{max\_fail} \) times since the last time it decreased (when using validation).

**References**


**See Also**

`trainbfg` | `traincgb` | `traincgf` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainrp` | `trainscg`

**Introduced before R2006a**
**traingd**

Gradient descent backpropagation

**Syntax**

```matlab
net.trainFcn = 'traingd'
[net,tr] = train(net,...)
```

**Description**

`traingd` is a network training function that updates weight and bias values according to gradient descent.

`net.trainFcn = 'traingd'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traingd`.

Training occurs according to `traingd` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.lr</td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-5</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses traingd with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traingd,

1. Set `net.trainFcn` to `'traingd'`. This sets `net.trainParam` to `traingd`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingd`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

More About

Gradient Descent Backpropagation

The batch steepest descent training function is `traingd`. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network `trainFcn` to `traingd`, and then call the function `train`. There is only one training function associated with a given network.

There are seven training parameters associated with `traingd`:

- `epochs`
- `show`
- `goal`
- `time`
- `min_grad`
- `max_fail`
- `lr`

The learning rate `lr` is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set
too small, the algorithm takes a long time to converge. See page 12-8 of [HDB96] for a discussion of the choice of learning rate.

The training status is displayed for every show iterations of the algorithm. (If show is set to NaN, then the training status is never displayed.) The other parameters determine when the training stops. The training stops if the number of iterations exceeds epochs, if the performance function drops below goal, if the magnitude of the gradient is less than mingrad, or if the training time is longer than time seconds. max_fail, which is associated with the early stopping technique, is discussed in Improving Generalization.

The following code creates a training set of inputs p and targets t. For batch training, all the input vectors are placed in one matrix.

\[
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
\]

Create the feedforward network.

\[
net = feedforwardnet(3,'traingd');
\]

In this simple example, turn off a feature that is introduced later.

\[
net.divideFcn = ' ';
\]

At this point, you might want to modify some of the default training parameters.

\[
net.trainParam.show = 50;
et.net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
\]

If you want to use the default training parameters, the preceding commands are not necessary.

Now you are ready to train the network.

\[
[net,tr] = train(net,p,t);
\]

The training record tr contains information about the progress of training.

Now you can simulate the trained network to obtain its response to the inputs in the training set.
a = net(p)
a =
   -1.0026  -0.9962   1.0010   0.9960

Try the *Neural Network Design* demonstration nnd12sd1 [HDB96] for an illustration of the performance of the batch gradient descent algorithm.

**Algorithms**

`traingd` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent:

```
dX = lr * dperf/dX
```

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

**See Also**

`traingda` | `traingdm` | `traingdx` | `trainlm`

*Introduced before R2006a*
traingda

Gradient descent with adaptive learning rate backpropagation

Syntax

net.trainFcn = 'traingda'
[net,tr] = train(net,...)

Description

traingda is a network training function that updates weight and bias values according to gradient descent with adaptive learning rate.

net.trainFcn = 'traingda' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with traingda.

Training occurs according to traingda training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.lr</td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td>net.trainParam.lr_inc</td>
<td>1.05</td>
<td>Ratio to increase learning rate</td>
</tr>
<tr>
<td>net.trainParam.lr_dec</td>
<td>0.7</td>
<td>Ratio to decrease learning rate</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.max_perf_inc</td>
<td>1.04</td>
<td>Maximum performance increase</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-5</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses traingda with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traingda,

1 Set net.trainFcn to 'traingda'. This sets net.trainParam to traingda’s default parameters.
2 Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with traingda.

See help feedforwardnet and help cascadeforwardnet for examples.

More About

Gradient Descent with Adaptive Learning Rate
Backpropagation

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

You can improve the performance of the steepest descent algorithm if you allow the learning rate to change during the training process. An adaptive learning rate attempts to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by traingd. First, the initial network output and error are calculated. At each epoch new
weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio, max_perf_inc (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by \( lr_{\text{dec}} = 0.7 \)). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by \( lr_{\text{inc}} = 1.05 \)).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it is decreased until stable learning resumes.

Try the *Neural Network Design* demonstration nnd12vl [HDB96] for an illustration of the performance of the variable learning rate algorithm.

Backpropagation training with an adaptive learning rate is implemented with the function traingda, which is called just like traingd, except for the additional training parameters max_perf_inc, lr_dec, and lr_inc. Here is how it is called to train the previous two-layer network:

```MATLAB
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traingda');
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net = train(net,p,t);
y = net(p)
```

## Algorithms

*traingda* can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance dperf with respect to the weight and bias variables \( X \). Each variable is adjusted according to gradient descent:

\[
dX = lr \cdot \frac{d\text{perf}}{dX}
\]
At each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor \( lr\_inc \). If performance increases by more than the factor \( max\_perf\_inc \), the learning rate is adjusted by the factor \( lr\_dec \) and the change that increased the performance is not made.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below \( min\_grad \).
- Validation performance has increased more than \( max\_fail \) times since the last time it decreased (when using validation).

**See Also**

traingd | traingdm | traingdx | trainlm

*Introduced before R2006a*
traingdm

Gradient descent with momentum backpropagation

Syntax

```matlab
net.trainFcn = 'traingdm'
[net, tr] = train(net, ...)
```

Description

traingdm is a network training function that updates weight and bias values according to gradient descent with momentum.

net.trainFcn = 'traingdm' sets the network trainFcn property.

[net, tr] = train(net, ...) trains the network with traingdm.

Training occurs according to traingdm training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.lr</td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.mc</td>
<td>0.9</td>
<td>Momentum constant</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-5</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between showing progress</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses traindgm with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traindgm,

1. Set net.trainFcn to 'traindgm'. This sets net.trainParam to traindgm's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with traindgm.

See help feedforwardnet and help cascadeforwardnet for examples.

More About

Gradient Descent with Momentum

In addition to traind, there are three other variations of gradient descent.

Gradient descent with momentum, implemented by traindgm, allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12-9 of [HDB96] for a discussion of momentum.

Gradient descent with momentum depends on two training parameters. The parameter lr indicates the learning rate, similar to the simple gradient descent. The parameter mc is the momentum constant that defines the amount of momentum. mc is set between 0 (no momentum) and values close to 1 (lots of momentum). A momentum constant of 1 results in a network that is completely insensitive to the local gradient and, therefore, does not learn properly.

```matlab
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traindgm');
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
```
net = train(net,p,t);
y = net(p)

Try the *Neural Network Design* demonstration nnd12mo [HDB96] for an illustration of the performance of the batch momentum algorithm.

**Algorithms**

*traingdm* can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance *perf* with respect to the weight and bias variables *X*. Each variable is adjusted according to gradient descent with momentum,

\[ dX = mc \cdot dX_{prev} + lr \cdot (1 - mc) \cdot dperf/dX \]

where *dX_{prev}* is the previous change to the weight or bias.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

**See Also**

*traingd* | *traingda* | *traingdx* | *trainlm*

*Introduced before R2006a*
**traingdx**

Gradient descent with momentum and adaptive learning rate backpropagation

**Syntax**

```matlab
net.trainFcn = 'traingdx'
[net,tr] = train(net,...)
```

**Description**

*traingdx* is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

`net.trainFcn = 'traingdx'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with *traingdx*.

Training occurs according to *traingdx* training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.epochs</code></td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td><code>net.trainParam.goal</code></td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td><code>net.trainParam.lr</code></td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td><code>net.trainParam.lr_inc</code></td>
<td>1.05</td>
<td>Ratio to increase learning rate</td>
</tr>
<tr>
<td><code>net.trainParam.lr_dec</code></td>
<td>0.7</td>
<td>Ratio to decrease learning rate</td>
</tr>
<tr>
<td><code>net.trainParam.max_fail</code></td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td><code>net.trainParam.max_perf_inc</code></td>
<td>1.04</td>
<td>Maximum performance increase</td>
</tr>
<tr>
<td><code>net.trainParam.mc</code></td>
<td>0.9</td>
<td>Momentum constant</td>
</tr>
<tr>
<td><code>net.trainParam.min_grad</code></td>
<td>1e-5</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td><code>net.trainParam.show</code></td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td><code>net.trainParam.showCommandLine</code></td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses traingdx with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traingdx,

1. Set `net.trainFcn` to 'traingdx'. This sets `net.trainParam` to traingdx's default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with traingdx.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

Algorithms

The function traingdx combines adaptive learning rate with momentum training. It is invoked in the same way as traingda, except that it has the momentum coefficient `mc` as an additional training parameter.

traingdx can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to gradient descent with momentum,

\[ dX = mc*dXprev + lr*mc*dperf/dX \]

where `dXprev` is the previous change to the weight or bias.

For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor `lr_inc`. If performance increases by more than the factor `max_perf_inc`, the learning rate is adjusted by the factor `lr_dec` and the change that increased the performance is not made.

---

**Table:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.showWindow</code></td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td><code>net.trainParam.time</code></td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

See Also
traingd | traingda | traingdm | trainlm

Introduced before R2006a
**trainlm**

Levenberg-Marquardt backpropagation

**Syntax**

```matlab
net.trainFcn = 'trainlm'
[net,tr] = train(net,...)
```

**Description**

*trainlm* is a network training function that updates weight and bias values according to Levenberg-Marquardt optimization.

*trainlm* is often the fastest backpropagation algorithm in the toolbox, and is highly recommended as a first-choice supervised algorithm, although it does require more memory than other algorithms.

`net.trainFcn = 'trainlm'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with *trainlm*.

Training occurs according to *trainlm* training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.epochs</code></td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td><code>net.trainParam.goal</code></td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td><code>net.trainParam.max_fail</code></td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td><code>net.trainParam.min_grad</code></td>
<td>1e-7</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td><code>net.trainParam.mu</code></td>
<td>0.001</td>
<td>Initial mu</td>
</tr>
<tr>
<td><code>net.trainParam.mu_dec</code></td>
<td>0.1</td>
<td>Mu decrease factor</td>
</tr>
<tr>
<td><code>net.trainParam.mu_inc</code></td>
<td>10</td>
<td>Mu increase factor</td>
</tr>
<tr>
<td><code>net.trainParam.mu_max</code></td>
<td>1e10</td>
<td>Maximum mu</td>
</tr>
</tbody>
</table>
Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for \( \text{max\_fail} \) epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

**Network Use**

You can create a standard network that uses `trainlm` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be trained with `trainlm`,

1. Set `net.trainFcn` to `'trainlm'`. This sets `net.trainParam` to `trainlm`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainlm`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

**Examples**

**Train Neural Network Using `trainlm` Train Function**

This example shows how to train a neural network using the `trainlm` train function.

Here a neural network is trained to predict body fat percentages.

```matlab
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainlm');
```
net = train(net, x, t);
y = net(x);

Limitations

This function uses the Jacobian for calculations, which assumes that performance is a mean or sum of squared errors. Therefore, networks trained with this function must use either the `mse` or `sse` performance function.

More About

Levenberg-Marquardt Algorithm

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

\[ H = J^T J \]

and the gradient can be computed as

\[ g = J^T e \]

where \( J \) is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and \( e \) is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

\[ x_{k+1} = x_k - (J^T J + \mu I)^{-1} J^T e \]

When the scalar \( \mu \) is zero, this is just Newton’s method, using the approximate Hessian matrix. When \( \mu \) is large, this becomes gradient descent with a small step size. Newton’s method is faster and more accurate near an error minimum, so the aim is to shift toward
Newton's method as quickly as possible. Thus, $\mu$ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has an efficient implementation in MATLAB® software, because the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB environment.

Try the Neural Network Design demonstration nnd12m [HDB96] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

## Algorithms

`trainlm` supports training with validation and test vectors if the network’s `NET.divideFcn` property is set to a data division function. Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`trainlm` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate the Jacobian $jX$ of performance $\text{perf}$ with respect to the weight and bias variables $X$. Each variable is adjusted according to Levenberg-Marquardt,

\[
\begin{align*}
jj &= jX \ast jX \\
je &= jX \ast E \\
dX &= -(jj+I*mu) \ast je
\end{align*}
\]

where $E$ is all errors and $I$ is the identity matrix.

The adaptive value $\mu$ is increased by $\mu_{\text{inc}}$ until the change above results in a reduced performance value. The change is then made to the network and $\mu$ is decreased by $\mu_{\text{dec}}$. 
Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below $\text{min}_\text{grad}$.
- $\mu$ exceeds $\mu_{\text{max}}$.
- Validation performance has increased more than $\text{max}_\text{fail}$ times since the last time it decreased (when using validation).

**Introduced before R2006a**
**trainoss**

One-step secant backpropagation

**Syntax**

```matlab
net.trainFcn = 'trainoss'
[net,tr] = train(net,...)
```

**Description**

*trainoss* is a network training function that updates weight and bias values according to the one-step secant method.

*net.trainFcn = 'trainoss'* sets the network *trainFcn* property.

*[net,tr] = train(net,...)* trains the network with *trainoss*.

Training occurs according to *trainoss* training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-10</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.searchFcn</td>
<td>'srchbac'</td>
<td>Name of line search routine to use</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>

Parameters related to line search methods (not all used for all methods):
Network Use

You can create a standard network that uses trainoss with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with trainoss:

1. Set net.trainFcn to 'trainoss'. This sets net.trainParam to trainoss’s default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with trainoss.

Examples

Train Neural Network Using trainoss Train Function

This example shows how to train a neural network using the trainoss train function.

Here a neural network is trained to predict body fat percentages.
More About

One Step Secant Method

Because the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration, the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

The one step secant method is described in [Batt92]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

Algorithms

**trainoss** can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance **perf** with respect to the weight and bias variables $X$. Each variable is adjusted according to the following:

$$X = X + a \cdot dX;$$

where $dX$ is the search direction. The parameter $a$ is selected to minimize the performance along the search direction. The line search function **searchFcn** is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous steps and gradients, according to the following formula:
\[ dX = -gX + Ac*X_{\text{step}} + Bc*dgX; \]

where \( gX \) is the gradient, \( X_{\text{step}} \) is the change in the weights on the previous iteration, and \( dgX \) is the change in the gradient from the last iteration. See Battiti (\textit{Neural Computation}, Vol. 4, 1992, pp. 141-166) for a more detailed discussion of the one-step secant algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below \texttt{min_grad}.
- Validation performance has increased more than \texttt{max_fail} times since the last time it decreased (when using validation).

\section*{References}


\section*{See Also}

\texttt{trainbfg \mid traincgb \mid traincgf \mid traincgp \mid trainda \mid traindm \mid traindx \mid trainlm \mid trainrp \mid trainscg}

\texttt{Introduced before R2006a}
**trainr**

Random order incremental training with learning functions

**Syntax**

```matlab
net.trainFcn = 'trainr'
[net,tr] = train(net,...)
```

**Description**

`trainr` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to 'trainr', thus:

`net.trainFcn = 'trainr'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainr`.

`trainr` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

Training occurs according to `trainr` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.epochs</code></td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td><code>net.trainParam.goal</code></td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td><code>net.trainParam.max_fail</code></td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td><code>net.trainParam.show</code></td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td><code>net.trainParam.showCommandLine</code></td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td><code>net.trainParam.showWindow</code></td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td><code>net.trainParam.time</code></td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses `trainr` by calling `competlayer` or `selforgmap`. To prepare a custom network to be trained with `trainr`,

1. Set `net.trainFcn` to `'trainr'`. This sets `net.trainParam` to `trainr`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to a learning function.
3. Set each `net.layerWeights{i,j}.learnFcn` to a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

See help `competlayer` and help `selforgmap` for training examples.

Algorithms

For each epoch, all training vectors (or sequences) are each presented once in a different random order, with the network and weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of epochs (repetitions) is reached.
- Performance is minimized to the goal.
- The maximum amount of time is exceeded.

See Also

`train`
Introduced before R2006a
**trainrp**

Resilient backpropagation

**Syntax**

```matlab
net.trainFcn = 'trainrp'
[net,tr] = train(net,...)
```

**Description**

`trainrp` is a network training function that updates weight and bias values according to the resilient backpropagation algorithm (Rprop).

`net.trainFcn = 'trainrp'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainrp`.

Training occurs according to `trainrp` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.epochs</code></td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td><code>net.trainParam.show</code></td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td><code>net.trainParam.showCommandLine</code></td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td><code>net.trainParam.showWindow</code></td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td><code>net.trainParam.goal</code></td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td><code>net.trainParam.time</code></td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td><code>net.trainParam.min_grad</code></td>
<td>1e-5</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td><code>net.trainParam.max_fail</code></td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td><code>net.trainParam.lr</code></td>
<td>0.01</td>
<td>Learning rate</td>
</tr>
<tr>
<td><code>net.trainParam.delt_inc</code></td>
<td>1.2</td>
<td>Increment to weight change</td>
</tr>
<tr>
<td><code>net.trainParam.delt_dec</code></td>
<td>0.5</td>
<td>Decrement to weight change</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses trainrp with feedforwardnet or cascadeforwardnet.

To prepare a custom network to be trained with trainrp,

1. Set net.trainFcn to 'trainrp'. This sets net.trainParam to trainrp's default parameters.
2. Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with trainrp.

Examples

Here is a problem consisting of inputs p and targets t to be solved with a network.

\[
p = [0 \ 1 \ 2 \ 3 \ 4 \ 5];
\]

\[
t = [0 \ 0 \ 0 \ 1 \ 1 \ 1];
\]

A two-layer feed-forward network with two hidden neurons and this training function is created.

Create and test a network.

\[
et = feedforwardnet(2,'trainrp');
\]

Here the network is trained and retested.

\[
et.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = net(p)
\]

See help feedforwardnet and help cascadeforwardnet for other examples.
More About

Resilient Backpropagation

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called “squashing” functions, because they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slopes must approach zero as the input gets large. This causes a problem when you use steepest descent to train a multilayer network with sigmoid functions, because the gradient can have a very small magnitude and, therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative can determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, the update value remains the same. Whenever the weights are oscillating, the weight change is reduced. If the weight continues to change in the same direction for several iterations, the magnitude of the weight change increases. A complete description of the Rprop algorithm is given in [RiBr93].

The following code recreates the previous network and trains it using the Rprop algorithm. The training parameters for trainrp are epochs, show, goal, time, min_grad, max_fail, delt_inc, delt_dec, delta0, and deltamax. The first eight parameters have been previously discussed. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below, the training parameters are left at the default values:

```matlab
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'trainrp');
net = train(net,p,t);
y = net(p)
```
rprop is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. You do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

## Algorithms

trainrp can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

\[
\text{d}X = \text{deltax}.*\text{sign(gX)};
\]

where the elements of `deltax` are all initialized to `deltat0`, and `gX` is the gradient. At each iteration the elements of `deltax` are modified. If an element of `gX` changes sign from one iteration to the next, then the corresponding element of `deltax` is decreased by `deltadec`. If an element of `gX` maintains the same sign from one iteration to the next, then the corresponding element of `deltax` is increased by `deltainc`. See Riedmiller, M., and H. Braun, “A direct adaptive method for faster backpropagation learning: The RPROP algorithm,” *Proceedings of the IEEE International Conference on Neural Networks*, 1993, pp. 586–591.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## References

See Also
trainbfg | traincgb | traincfg | traincgf | traincgp | trainda | traindm | traindx | 
trainlm | trainoss | trainscg

Introduced before R2006a
**trainru**

Unsupervised random order weight/bias training

**Syntax**

```matlab
net.trainFcn = 'trainru'
[net,tr] = train(net,...)
```

**Description**

`trainru` is not called directly. Instead it is called by `train` for networks whose
`net.trainFcn` property is set to 'trainru', thus:

`net.trainFcn = 'trainru'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainru`.

`trainru` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

Training occurs according to `trainru` training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.trainParam.epochs</code></td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td><code>net.trainParam.show</code></td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td><code>net.trainParam.showCommandLine</code></td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td><code>net.trainParam.showWindow</code></td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td><code>net.trainParam.time</code></td>
<td>Inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>

**Network Use**

To prepare a custom network to be trained with `trainru`,
Set `net.trainFcn` to 'trainru'. This sets `net.trainParam` to `trainru`'s default parameters.

Set each `net.inputWeights{i,j}.learnFcn` to a learning function.

Set each `net.layerWeights{i,j}.learnFcn` to a learning function.

Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

**Algorithms**

For each epoch, all training vectors (or sequences) are each presented once in a different random order, with the network and weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.

**See Also**

`train` | `trainr`  

**Introduced in R2010b**
trains

Sequential order incremental training with learning functions

Syntax

net.trainFcn = 'trains'
[net, tr] = train(net, ...)

Description

trains is not called directly. Instead it is called by train for networks whose net.trainFcn property is set to 'trains', thus:

net.trainFcn = 'trains' sets the network trainFcn property.

[net, tr] = train(net, ...) trains the network with trains.

trains trains a network with weight and bias learning rules with sequential updates. The sequence of inputs is presented to the network with updates occurring after each time step.

This incremental training algorithm is commonly used for adaptive applications.

Training occurs according to trains training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>Inf</td>
<td>Maximum time to train in seconds</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses trains for adapting by calling perceptron or linearlayer.

To prepare a custom network to adapt with trains,

1. Set net.adaptFcn to 'trains'. This sets net.adaptParam to trains’s default parameters.
2. Set each net.inputWeights{i,j}.learnFcn to a learning function. Set each net.layerWeights{i,j}.learnFcn to a learning function. Set each net.biases{i}.learnFcn to a learning function. (Weight and bias learning parameters are automatically set to default values for the given learning function.)

To allow the network to adapt,

1. Set weight and bias learning parameters to desired values.
2. Call adapt.

See help perceptron and help linearlayer for adaption examples.

Algorithms

Each weight and bias is updated according to its learning function after each time step in the input sequence.

See Also

train | trainb | trainc | trainr

Introduced before R2006a
trainscg

Scaled conjugate gradient backpropagation

Syntax

net.trainFcn = 'trainscg'
[net,tr] = train(net,...)

Description

trainscg is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

net.trainFcn = 'trainscg' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainscg.

Training occurs according to trainscg training parameters, shown here with their default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>1000</td>
<td>Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25</td>
<td>Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.showCommandLine</td>
<td>false</td>
<td>Generate command-line output</td>
</tr>
<tr>
<td>net.trainParam.showWindow</td>
<td>true</td>
<td>Show training GUI</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0</td>
<td>Performance goal</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf</td>
<td>Maximum time to train in seconds</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-6</td>
<td>Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>6</td>
<td>Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.sigma</td>
<td>5.0e-5</td>
<td>Determine change in weight for second derivative approximation</td>
</tr>
</tbody>
</table>
Network Use

You can create a standard network that uses `trainscg` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `trainscg`,

1. Set `net.trainFcn` to `'trainscg'`. This sets `net.trainParam` to `trainscg`'s default parameters.

2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainscg`.

Examples

Here is a problem consisting of inputs `p` and targets `t` to be solved with a network.

```matlab
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

A two-layer feed-forward network with two hidden neurons and this training function is created.

```matlab
net = feedforwardnet(2,'trainscg');
```

Here the network is trained and retested.

```matlab
net = train(net,p,t);
a = net(p)
```

See `help feedforwardnet` and `help cascadeforwardnet` for other examples.
Algorithms

trainscg can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X.

The scaled conjugate gradient algorithm is based on conjugate directions, as in traincgp, traincgf, and traincgb, but this algorithm does not perform a line search at each iteration. See Moller (Neural Networks, Vol. 6, 1993, pp. 525–533) for a more detailed discussion of the scaled conjugate gradient algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

References

Moller, Neural Networks, Vol. 6, 1993, pp. 525-533

See Also

trainbfg | traincgb | traincgf | traincgp | traingda | traingdm | traingdx | trainlm | trainoss | trainrp

Introduced before R2006a
tribas

Triangular basis transfer function

Graph and Symbol

![Graph of tribas](image)

\[ a = \text{tribas}(n) \]

Triangular Basis Function

Syntax

\[ A = \text{tribas}(N,FP) \]

Description

\text{tribas} is a neural transfer function. Transfer functions calculate a layer’s output from its net input.

\[ A = \text{tribas}(N,FP) \] takes \( N \) and optional function parameters,

<table>
<thead>
<tr>
<th>( N )</th>
<th>S-by-Q matrix of net input (column) vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( FP )</td>
<td>Struct of function parameters (ignored)</td>
</tr>
</tbody>
</table>

and returns \( A \), an S-by-Q matrix of the triangular basis function applied to each element of \( N \).

\[ \text{info} = \text{tribas}('\text{code}') \] can take the following forms to return specific information:

\text{tribas}('name') returns the name of this function.
tribas('output',FP) returns the [min max] output range.

tribas('active',FP) returns the [min max] active input range.

tribas('fullderiv') returns 1 or 0, depending on whether \( \frac{dA}{dN} \) is \( S \times S \times Q \) or \( S \times Q \).

tribas('fpnames') returns the names of the function parameters.

tribas('fpdefaults') returns the default function parameters.

**Examples**

Here you create a plot of the tribas transfer function.

```matlab
n = -5:0.1:5;
a = tribas(n);
plot(n,a)
```

Assign this transfer function to layer \( i \) of a network.

```matlab
net.layers{i}.transferFcn = 'tribas';
```

**Algorithms**

\[
a = \text{tribas}(n) = \begin{cases} 
1 - \text{abs}(n), & \text{if } -1 \leq n \leq 1 \\
0, & \text{otherwise}
\end{cases}
\]

**See Also**

radbas | sim

*Introduced before R2006a*
**tritop**

Triangle layer topology function

**Syntax**

```matlab
pos = tritop(dimensions)
```

**Description**

`tritop` calculates neuron positions for layers whose neurons are arranged in an N-dimensional triangular grid.

`pos = tritop(dimensions)` takes one argument:

<table>
<thead>
<tr>
<th>dimensions</th>
<th>Row vector of dimension sizes</th>
</tr>
</thead>
</table>

and returns an N-by-S matrix of N coordinate vectors, where N is the number of dimensions and S is the product of `dimensions`.

**Examples**

**Display Layer with Triangular Pattern**

This example shows how to display a two-dimensional layer with 40 neurons arranged in an 8-by-5 triangular grid.

```matlab
pos = tritop([8 5]);
plotsom(pos)
```
See Also
gridtop | hextop | randtop

Introduced in R2010b
unconfigure

Unconfigure network inputs and outputs

Syntax

```matlab
unconfigure(net)
unconfigure(net, 'inputs', i)
unconfigure(net, 'outputs', i)
```

Description

`unconfigure(net)` returns a network with its input and output sizes set to 0, its input and output processing settings and related weight initialization settings set to values consistent with zero-sized signals. The new network will be ready to be reconfigured for data of the same or different dimensions than it was previously configured for.

`unconfigure(net, 'inputs', i)` unconfigures the inputs indicated by the indices `i`. If no indices are specified, all inputs are unconfigured.

`unconfigure(net, 'outputs', i)` unconfigures the outputs indicated by the indices `i`. If no indices are specified, all outputs are unconfigured.

Examples

Here a network is configured for a simple fitting problem, and then unconfigured.

```matlab
[x,t] = simplefit_dataset;
net = fitnet(10);
view(net)
net = configure(net,x,t);
view(net)
net = unconfigure(net)
view(net)
```
See Also
configure | isconfigured

Introduced in R2010b
vec2ind

Convert vectors to indices

Syntax

[ind,n] = vec2ind

Description

ind2vec and vec2ind(vec) allow indices to be represented either by themselves or as vectors containing a 1 in the row of the index they represent.

[ind,n] = vec2ind takes one argument,

<table>
<thead>
<tr>
<th>vec</th>
<th>Matrix of vectors, each containing a single 1</th>
</tr>
</thead>
</table>

and returns

<table>
<thead>
<tr>
<th>ind</th>
<th>The indices of the 1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number of rows in vec</td>
</tr>
</tbody>
</table>

Examples

Here three vectors are converted to indices and back, while preserving the number of rows.

vec = [0 0 1 0; 1 0 0 0; 0 1 0 0]'

vec =
     0     1     0
     0     0     1
     1     0     0
     0     0     0

[ind,n] = vec2ind(vec)
ind = 3 1 2

n = 4

vec2 = full(ind2vec(ind,n))

vec2 =
0 1 0
0 0 1
1 0 0
0 0 0

See Also
ind2sub | ind2vec | sub2ind

Introduced before R2006a
**view**

View neural network

**Syntax**

`view(net)`

**Description**

`view(net)` opens a window that shows your neural network (specified in `net`) as a graphical diagram.

**Example**

**View Neural Network**

This example shows how to view the diagram of a pattern recognition network.

```matlab
[x,t] = iris_dataset;
net = patternnet;
net = configure(net,x,t);
view(net)
```
Introduced in R2008a
Neural Net Fitting

Fit data by training a two-layer feed-forward network

Description
The Neural Net Fitting app leads you through solving a data-fitting problem using a two-layer feed-forward network. It helps you select data, divide it into training, validation, and testing sets, define the network architecture, and train the network. You can select your own data from the MATLAB workspace or use one of the example datasets. After training the network, evaluate its performance using mean squared error and regression analysis. Further analyze the results using visualization tools such as a regression fit or histogram of the errors. You can then evaluate the performance of the network on a test set. If you are not satisfied with the results, you can retrain the network with modified settings or on a larger data set.

You can generate MATLAB scripts to reproduce results or customize the training process. You can also save the trained network to test on new data or use for solving similar fitting problems. The app also provides the option to generate various deployable versions of your trained network. For example, you can deploy the trained network using MATLAB Compiler, MATLAB Coder, or Simulink Coder tools.

Required Products
- MATLAB
- Deep Learning Toolbox

Open the Neural Net Fitting App
- MATLAB Toolstrip: On the Apps tab, under Machine Learning, click the app icon.
- MATLAB command prompt: Enter nftool.
Examples

• “Fit Data with a Shallow Neural Network”

See Also

Apps
Neural Net Time Series | Neural Net Clustering | Neural Net Pattern Recognition

Functions
feedforwardnet | fitnet | trainbr | trainlm | trainscg

Topics
“Fit Data with a Shallow Neural Network”
Neural Net Clustering

Cluster data by training a self-organizing maps network

Description
The Neural Net Clustering app leads you through solving a clustering problem using a self-organizing map (SOM). It helps you select data, define the network architecture, and train the network. You can select your own data from the MATLAB workspace or use one of the example datasets. After training the network, analyze the results using various visualization tools. You can then evaluate the performance of the network on a test set. If you are not satisfied with the results, you can retrain the network with modified settings or on a larger data set.

You can generate MATLAB scripts to reproduce results or customize the training process. You can also save the trained network to test on new data or use for solving similar clustering problems. The app also provides the option to generate various deployable versions of your trained network. For example, you can deploy the trained network using MATLAB Compiler, MATLAB Coder, or Simulink Coder tools.

Required Products
- MATLAB
- Deep Learning Toolbox

Open the Neural Net Clustering App
- MATLAB Toolstrip: On the Apps tab, under Machine Learning, click the app icon.
- MATLAB command prompt: Enter nctool.

Examples
- “Cluster Data with a Self-Organizing Map”
See Also

Apps
Neural Net Fitting | Neural Net Pattern Recognition | Neural Net Time Series

Functions
learnsomb | selforgmap | trainbu

Topics
“Cluster Data with a Self-Organizing Map”
Neural Net Pattern Recognition

Classify data by training a two-layer feed-forward network

Description

The Neural Net Pattern Recognition app leads you through solving a data classification problem using a two-layer feed-forward network. It helps you select data, divide it into training, validation, and testing sets, define the network architecture, and train the network. You can select your own data from the MATLAB workspace or use one of the example datasets. After training the network, evaluate its performance using cross-entropy and percent misclassification error. Further analyze the results using visualization tools such as confusion matrices and receiver operating characteristic curves. You can then evaluate the performance of the network on a test set. If you are not satisfied with the results, you can retrain the network with modified settings or on a larger data set.

You can generate MATLAB scripts to reproduce results or customize the training process. You can also save the trained network to test on new data or use for solving similar classification problems. The app also provides the option to generate various deployable versions of your trained network. For example, you can deploy the trained network using MATLAB Compiler, MATLAB Coder, or Simulink Coder tools.

Required Products

- MATLAB
- Deep Learning Toolbox

Open the Neural Net Pattern Recognition App

- MATLAB Toolstrip: On the Apps tab, under Machine Learning, click the app icon.
- MATLAB command prompt: Enter nprtool.
Examples

• “Classify Patterns with a Shallow Neural Network”

See Also

Apps
Neural Net Fitting | Neural Net Clustering | Neural Net Time Series

Functions
patternnet | trainlm

Topics
“Classify Patterns with a Shallow Neural Network”
Neural Net Time Series

Solve a nonlinear time series problem by training a dynamic neural network

Description

The Neural Net Time Series app leads you through solving three different kinds of nonlinear time series problems using a dynamic network. It helps you select data, divide it into training, validation, and testing sets, define the network architecture, and train the network. You can select your own data from the MATLAB workspace or use one of the example datasets. After training the network, evaluate its performance using mean squared error and regression analysis. Further analyze the results using visualization tools such as an error autocorrelation plot or histogram of the errors. You can then evaluate the performance of the network on a test set. If you are not satisfied with the results, retrain the network with modified settings or on a larger data set.

You can generate MATLAB scripts to reproduce results or customize the training process. You can also save the trained network to test on new data or use for solving similar classification problems. The app also provides the option to generate various deployable versions of your trained network. For example, you can deploy the trained network using MATLAB Compiler, MATLAB Coder, or Simulink Coder tools.

Required Products

- MATLAB
- Deep Learning Toolbox

Open the Neural Net Time Series App

- MATLAB Toolstrip: On the Apps tab, under Machine Learning, click the app icon.
- MATLAB command prompt: Enter ntstool.
Examples

- “Shallow Neural Network Time-Series Prediction and Modeling”

See Also

Apps
Neural Net Fitting | Neural Net Clustering | Neural Net Pattern Recognition

Functions
narnet | narxnet

Topics
“Shallow Neural Network Time-Series Prediction and Modeling”
matlab.io.datastore.MiniBatchable class

Package: matlab.io.datastore

Add mini-batch support to datastore

Description

matlab.io.datastore.MiniBatchable is an abstract mixin class that adds support for mini-batches to your custom datastore for use with Deep Learning Toolbox. A mini-batch datastore contains training and test data sets for use in Deep Learning Toolbox training, prediction, and classification.

To use this mixin class, you must inherit from the matlab.io.datastore.MiniBatchable class in addition to inheriting from the matlab.io.Datastore base class. Type the following syntax as the first line of your class definition file:

```matlab
classdef MyDatastore < matlab.io.Datastore & ... 
    matlab.io.datastore.MiniBatchable

end
```

To add support for mini-batches to your datastore:

- Inherit from an additional class matlab.io.datastore.MiniBatchable
- Define two additional properties: MiniBatchSize and NumObservations.

For more details and steps to create your custom mini-batch datastore to optimize performance during training, prediction, and classification, see “Develop Custom Mini-Batch Datastore”.

Properties

MiniBatchSize — Number of observations in each batch
positive integer
Number of observations that are returned in each batch, or call of the read function. For training, prediction, and classification, the MiniBatchSize property is set to the minibatch size defined in trainingOptions.

**Attributes:**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>true</td>
</tr>
<tr>
<td>Access</td>
<td>Public</td>
</tr>
</tbody>
</table>

**NumObservations** — Total number of observations in the datastore  
positive integer

Total number of observations contained within the datastore. This number of observations is the length of one training epoch.

**Attributes:**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>true</td>
</tr>
<tr>
<td>SetAccess</td>
<td>Protected</td>
</tr>
<tr>
<td>ReadAccess</td>
<td>Public</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>true</td>
</tr>
<tr>
<td>Sealed</td>
<td>false</td>
</tr>
</tbody>
</table>

For information on class attributes, see “Class Attributes” (MATLAB).

**Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

**Examples**
Train Network Using Out-of-Memory Sequence Data

This example shows how to train a deep learning network on out-of-memory sequence data by transforming and combining datastores.

A transformed datastore transforms or processes data read from an underlying datastore. You can use a transformed datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use transformed datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data. When you have separate datastores containing predictors and labels, you can combine them so you can input the data into a deep learning network.

When training the network, the software creates mini-batches of sequences of the same length by padding, truncating, or splitting the input data. For in-memory data, the trainingOptions function provides options to pad and truncate input sequences, however, for out-of-memory data, you must pad and truncate the sequences manually.

Load Training Data

Load the Japanese Vowels data set as described in [1] and [2]. The zip file japaneseVowels.zip contains sequences of varying length. The sequences are divided into two folders, Train and Test, which contain training sequences and test sequences, respectively. In each of these folders, the sequences are divided into subfolders, which are numbered from 1 to 9. The names of these subfolders are the label names. A MAT file represents each sequence. Each sequence is a matrix with 12 rows, with one row for each feature, and a varying number of columns, with one column for each time step. The number of rows is the sequence dimension and the number of columns is the sequence length.

Unzip the sequence data.

```matlab
filename = "japaneseVowels.zip";
outputFolder = fullfile(tempdir,"japaneseVowels");
unzip(filename,outputFolder);
```

For the training predictors, create a file datastore and specify the read function to be the load function. The load function, loads the data from the MAT-file into a structure array. To read files from the subfolders in the training folder, set the 'IncludeSubfolders' option to true.

```matlab
folderTrain = fullfile(outputFolder,"Train");
fdsPredictorTrain = fileDatastore(folderTrain, ...
Preview the datastore. The returned struct contains a single sequence from the first file.

```matlab
tdsTrain = transform(fdsPredictorTrain,@(data) padSequence(data,sequenceLength));
```

For the labels, create a file datastore and specify the read function to be the `readLabel` function, defined at the end of the example. The `readLabel` function extracts the label from the subfolder name.

```matlab
classNames = string(1:9);
fdsLabelTrain = fileDatastore(folderTrain, ...  
    'ReadFcn',@(filename) readLabel(filename,classNames), ...  
    'IncludeSubfolders',true);
```

Preview the datastore. The output corresponds to the label of the first file.

```matlab
ans = categorical
    1
```

**Transform and Combine Datastores**

To input the sequence data from the datastore of predictors to a deep learning network, the mini-batches of the sequences must have the same length. Transform the datastore using the `padSequence` function, defined at the end of the datastore, that pads or truncates the sequences to have length 20.

```matlab
sequenceLength = 20;
tdsTrain = transform(fdsPredictorTrain,@(data) padSequence(data,sequenceLength));
```

Preview the transformed datastore. The output corresponds to the padded sequence from the first file.

```matlab
X = preview(tdsTrain)
X = 1x1 cell array
    {12x20 double}
To input both the predictors and labels from both datastores into a deep learning network, combine them using the `combine` function.

```matlab
cdsTrain = combine(tdsTrain,fdsLabelTrain);
```

Preview the combined datastore. The datastore returns a 1-by-2 cell array. The first element corresponds to the predictors. The second element corresponds to the label.

```matlab
preview(cdsTrain)
ans = 1x2 cell array
    {12x20 double}    {[1]}
```

**Define LSTM Network Architecture**

Define the LSTM network architecture. Specify the number of features of the input data as the input size. Specify an LSTM layer with 100 hidden units and to output the last element of the sequence. Finally, specify a fully connected layer with output size equal to the number of classes, followed by a softmax layer and a classification layer.

```matlab
numFeatures = 12;
numClasses = numel(classNames);
numHiddenUnits = 100;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Set the solver to 'adam' and 'GradientThreshold' to 2. Set the mini-batch size to 27 and set the maximum number of epochs to 75. The datastores do not support shuffling, so set 'Shuffle' to 'never'.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set 'ExecutionEnvironment' to 'cpu'. To train on a GPU, if available, set 'ExecutionEnvironment' to 'auto' (the default value).

```matlab
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
Train the LSTM network with the specified training options.

```matlab
net = trainNetwork(cdsTrain, layers, options);
```

Test the Network

Create a transformed datastore containing the held-out test data using the same steps as for the training data.

```matlab
folderTest = fullfile(outputFolder,"Test");
```
fdsPredictorTest = fileDatastore(folderTest, ...  
    'ReadFcn',@load, ...  
    'IncludeSubfolders',true);

tdsTest = transform(fdsPredictorTest,@(data) padSequence(data,sequenceLength));

Make predictions on the test data using the trained network.

YPred = classify(net,tdsTest,'MiniBatchSize',miniBatchSize);

Calculate the classification accuracy on the test data. To get the labels of the test set, create a file datastore with the read function `readLabel` and specify to include subfolders. Specify that the outputs are vertically concatenateable by setting the 'UniformRead' option to true.

fdsLabelTest = fileDatastore(folderTest, ...  
    'ReadFcn',@(filename) readLabel(filename,classNames), ...  
    'IncludeSubfolders',true, ...  
    'UniformRead',true);

YTest = readall(fdsLabelTest);

accuracy = mean(YPred == YTest)
accuracy = 0.9351

**Functions**

The `readLabel` function extracts the label from the specified filename over the categories in `classNames`.

```matlab
function label = readLabel(filename,classNames)
    filepath = fileparts(filename);
    [~,label] = fileparts(filepath);
    label = categorical(string(label),classNames);
end
```

The `padSequence` function pads or truncates the sequence in `data.X` to have the specified sequence length and returns the result in a 1-by-1 cell.

```matlab
function sequence = padSequence(data,sequenceLength)
    sequence = data.X;
    [C,S] = size(sequence);
```
if S < sequenceLength
    padding = zeros(C,sequenceLength-S);
    sequence = [sequence padding];
else
    sequence = sequence(:,1:sequenceLength);
end

sequence = {sequence};
end

 Compatibility Considerations

matlab.io.datastore.MiniBatchable is not recommended for custom image preprocessing
Not recommended starting in R2019a

Starting in R2019a, matlab.io.datastore.MiniBatchable is not recommended for custom image processing. Use the transform and combine functions with built-in datastores instead. For more information, see “Preprocess Images for Deep Learning”.

References


See Also

Topics
“Deep Learning in MATLAB”
“Develop Custom Mini-Batch Datastore”

Introduced in R2018a
read

Class: matlab.io.datastore.MiniBatchable
Package: matlab.io.datastore

Read data from mini-batch datastore

Note The read method of matlab.io.datastore.MiniBatchable is not recommended. For more information, see Compatibility Considerations.

Syntax

data = read(ds)
[data,info] = read(ds)

Description

data = read(ds) returns data from a mini-batch datastore. Subsequent calls to the read function continue reading from the endpoint of the previous call.

[data,info] = read(ds) also returns information about the extracted data in info, including metadata.

Input Arguments

mbds — Mini-batch datastore
dataset | custom MiniBatchable datastore | ...

Mini-batch datastore, specified as a built-in datastore or custom mini-batch datastore. For more information, see “Datastores for Deep Learning”.
Output Arguments

data — Output data
table

Output data, returned as a table with MiniBatchSize number of rows. For the last mini-
batch of data in the datastore, if NumObservations is not evenly divisible by
MiniBatchSize, then data should contain the remaining observations in the datastore
(a partial batch smaller than MiniBatchSize).

The table should have two columns, with predictors in the first column and responses in
the second column.

info — Information about read data
structure array

Information about read data, returned as a structure array.

Attributes

Hidden true

To learn about attributes of methods, see Method Attributes (MATLAB).

Compatibility Considerations

read is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning
networks, you had to specify a custom read function using the readFcn property of
imageDatastore. However, reading files using a custom read function was slow because
imageDatastore did not prefetch files.

In R2018a, four classes including matlab.io.datastore.MiniBatchable were
introduced as a solution to perform custom image preprocessing with support for
prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore
using matlab.io.datastore.MiniBatchable has several challenges and limitations.
• In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
• You must specify a value for the NumObservations property, but this value may be ill-defined or difficult to define in real-world applications.
• Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder™.

Starting in R2019a, built-in datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable`.

• The functions enable data preprocessing and concatenation for all types of datastores, including `ImageDatastore`.
• The `transform` function only requires you to define the data processing pipeline.
• When used on a deterministic datastore, the functions support `tall` data types and `MapReduce`.
• The functions support deployed workflows.

**Note** The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove the `read` method of `matlab.io.datastore.MiniBatchable` at this time.

**See Also**
`combine` | `matlab.io.Datastore` | `matlab.io.datastore.MiniBatchable` | `read (Datastore)` | `transform`

**Topics**
“Datastores for Deep Learning”
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a
matlab.io.datastore.BackgroundDispatchable class

Package: matlab.io.datastore

(Not recommended) Add prefetch reading support to datastore

Note matlab.io.datastore.BackgroundDispatchable is not recommended. For more information, see Compatibility Considerations.

Description

matlab.io.datastore.BackgroundDispatchable is an abstract mixin class that adds support for prefetch reading to your custom datastore for use with Deep Learning Toolbox.

To use this mixin class, you must inherit from the matlab.io.datastore.BackgroundDispatchable class in addition to inheriting from the matlab.io.Datastore base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ... 
    matlab.io.datastore.BackgroundDispatchable 
    ... 
end
```

To add support for parallel processing to your custom datastore, you must:

- Inherit from an additional class matlab.io.datastore.BackgroundDispatchable
- Define the additional method: readByIndex

For more details and steps to create your custom datastore to optimize performance during training, prediction, and classification, see “Develop Custom Mini-Batch Datastore”.

1-590
Properties

**DispatchInBackground** — Dispatch observations in background

```
true (default) | false
```

Dispatch observations in the background during training, prediction, or classification, specified as `true` or `false`. To use background dispatching, you must have Parallel Computing Toolbox.

Attributes:

- **Public**

Methods

- **readByIndex** (Not recommended) Return observations from a datastore specified by index

Attributes

- **Abstract**
- **Sealed**

For information on class attributes, see “Class Attributes” (MATLAB).

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Compatibility Considerations

**matlab.io.datastore.BackgroundDispatchable is not recommended**

_Not recommended starting in R2019a_
Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the readFcn property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the NumObservations property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support tall data types and MapReduce.
- The functions support deployed workflows.

**Note** The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.
There are no plans to remove `matlab.io.datastore.BackgroundDispatchable` at this time.

**See Also**


**Topics**

“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

**Introduced in R2018a**
**readByIndex**

Class: `matlab.io.datastore.BackgroundDispatchable`
Package: `matlab.io.datastore`

(Not recommended) Return observations from a datastore specified by index

**Note** readByIndex is not recommended. For more information, see Compatibility Considerations.

**Syntax**

```
[data,info] = readByIndex(ds,ind)
```

**Description**

```
[data,info] = readByIndex(ds,ind) returns a subset of observations in a datastore, ds. The desired observations are specified by indices, ind.
```

**Input Arguments**

- **ds** — Input datastore
  
  Datastore object

  Input datastore, specified as a `Datastore` object.

- **ind** — Indices
  
  vector of positive integers

  Indices of observations, specified as a vector of positive integers.
Output Arguments

data — Observations from datastore
    table

Observations from the datastore, returned as a table or an array according to the read method of the datastore. For example, when ds is a custom mini-batch datastore, then data is a table with the same format as returned by the read (MiniBatchable) method.

info — Information about read data
    structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename</td>
<td>Filename is a fully resolved path containing the path string, name of the file, and file extension.</td>
</tr>
<tr>
<td>FileSize</td>
<td>Total file size, in bytes. For MAT-files, FileSize is the total number of key-value pairs in the file.</td>
</tr>
</tbody>
</table>

Attributes

Abstract    true
Access      Public

To learn about attributes of methods, see Method Attributes (MATLAB).

Tips

- You must implement the readByIndex method by deriving a subclass from the matlab.io.datastore.BackgroundDispatchable class.
Compatibility Considerations

**readByIndex is not recommended**

*Not recommended starting in R2019a*

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.BackgroundDispatchable`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
The functions support deployed workflows.

**Note** The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `matlab.io.datastore.BackgroundDispatchable` class or the `readByIndex` method at this time.

**See Also**
`combine`, `matlab.io.Datastore`, `read`, `readall`, `transform`

**Topics**
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

**Introduced in R2018a**
matlab.io.datastore.PartitionableByIndex class

Package: matlab.io.datastore

(Not recommended) Add parallelization support to datastore

Note matlab.io.datastore.PartitionableByIndex is not recommended. For more information, see Compatibility Considerations.

Description

matlab.io.datastore.PartitionableByIndex is an abstract mixin class that adds parallelization support to your custom datastore for use with Deep Learning Toolbox. This class requires Parallel Computing Toolbox.

To use this mixin class, you must inherit from the matlab.io.datastore.PartitionableByIndex class in addition to inheriting from the matlab.io.Datastore base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
    matlab.io.datastore.PartitionableByIndex
        ...
end
```

To add support for parallel processing to your custom datastore, you must:

- Inherit from an additional class matlab.io.datastore.PartitionableByIndex
- Define the additional method: partitionByIndex

For more details and steps to create your custom datastore with parallel processing support, see “Develop Custom Mini-Batch Datastore”.
Methods
partitionByIndex  (Not recommended) Partition a datastore according to indices

Attributes

Abstract       true
Sealed         false

For information on class attributes, see “Class Attributes” (MATLAB).

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Compatibility Considerations

matlab.io.datastore.PartitionableByIndex is not recommended
Not recommended starting in R2019a

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the readFcn property of imageDatastore. However, reading files using a custom read function was slow because imageDatastore did not prefetch files.

In R2018a, four classes including matlab.io.datastore.MiniBatchable and matlab.io.datastore.PartitionableByIndex were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using matlab.io.datastore.MiniBatchable has several challenges and limitations.

• In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

**Note** The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `matlab.io.datastore.PartitionableByIndex` at this time.

**See Also**

**Topics**
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”
Introduced in R2018a
partitionByIndex

(Not recommended) Partition a datastore according to indices

**Note** partitionByIndex is not recommended. For more information, see Compatibility Considerations.

**Syntax**

\[ ds2 = \text{partitionByIndex}(ds, \text{ind}) \]

**Description**

\[ ds2 = \text{partitionByIndex}(ds, \text{ind}) \] partitions a subset of observations in a datastore, \( ds \), into a new datastore, \( ds2 \). The desired observations are specified by indices, \( \text{ind} \).

**Input Arguments**

- **ds** — Input datastore  
  Datastore object

  Input datastore, specified as a `Datastore` object.

- **ind** — Indices  
  vector of positive integers

  Indices of observations, specified as a vector of positive integers.

**Output Arguments**

- **ds2** — Partitioned datastore  
  Datastore object

  Partitioned datastore, returned as a `Datastore` object.
Attributes

<table>
<thead>
<tr>
<th>Abstract</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Public</td>
</tr>
</tbody>
</table>

To learn about attributes of methods, see Method Attributes (MATLAB).

Tips

- You must implement the `partitionByIndex` method by deriving a subclass from the `matlab.io.datastore.Partitionable` class.

Compatibility Considerations

**`partitionByIndex` is not recommended**

_Not recommended starting in R2019a_

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, four classes including `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using `matlab.io.datastore.MiniBatchable` has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.
Starting in R2019a, datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom data preprocessing, or transformations. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over `matlab.io.datastore.MiniBatchable` and `matlab.io.datastore.PartitionableByIndex`.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function only requires you to define the data processing pipeline.
- When used on a deterministic datastore, the functions support tall data types and MapReduce.
- The functions support deployed workflows.

**Note** The recommended solution to transform data with basic image preprocessing operations, including resizing, rotation, and reflection, is `augmentedImageDatastore`. For more information, see “Preprocess Images for Deep Learning”.

There are no plans to remove `partitionByIndex` at this time.

**See Also**
`combine` | `matlab.io.Datastore` | `transform`

**Topics**
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

**Introduced in R2018a**
trainAutoencoder

Train an autoencoder

Syntax

autoenc = trainAutoencoder(X)
autoenc = trainAutoencoder(X,hiddenSize)
autoenc = trainAutoencoder(____,Name,Value)

Description

autoenc = trainAutoencoder(X) returns an autoencoder, autoenc, trained using
the training data in X.

autoenc = trainAutoencoder(X,hiddenSize) returns an autoencoder autoenc,
with the hidden representation size of hiddenSize.

autoenc = trainAutoencoder(____,Name,Value) returns an autoencoder
autoenc, for any of the above input arguments with additional options specified by one
or more Name,Value pair arguments.

For example, you can specify the sparsity proportion or the maximum number of training
iterations.

Examples

Train Sparse Autoencoder

Load the sample data.

X = abalone_dataset;

X is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex
(M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera
weight, shell weight. For more information on the dataset, type `help abalone_dataset` in the command line.

Train a sparse autoencoder with default settings.

```matlab
t = trainAutoencoder(X);
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```matlab
XReconstructed = predict(autoenc,X);
```

Compute the mean squared reconstruction error.

```matlab
mseError = mse(X-XReconstructed)
```

```matlab
mseError = 0.0167
```

---

**Train Autoencoder with Specified Options**

Load the sample data.

```matlab
X = abalone_dataset;
```

X is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex (M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera weight, shell weight. For more information on the dataset, type `help abalone_dataset` in the command line.

Train a sparse autoencoder with hidden size 4, 400 maximum epochs, and linear transfer function for the decoder.

```matlab
autoenc = trainAutoencoder(X,4,'MaxEpochs',400, ...
'DecoderTransferFunction','purelin');
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```matlab
XReconstructed = predict(autoenc,X);
```

Compute the mean squared reconstruction error.

```matlab
mseError = mse(X-XReconstructed)
```

```matlab
mseError = 0.0046
```
Reconstruct Observations Using Sparse Autoencoder

Generate the training data.

```matlab
rng(0,'twister');  % For reproducibility
n = 1000;
r = linspace(-10,10,n)';
x = 1 + r*5e-2 + sin(r)./r + 0.2*randn(n,1);
```

Train autoencoder using the training data.

```matlab
hiddenSize = 25;
autoenc = trainAutoencoder(x',hiddenSize,...
    'EncoderTransferFunction','satlin',...
    'DecoderTransferFunction','purelin',...
    'L2WeightRegularization',0.01,...
    'SparsityRegularization',4,...
    'SparsityProportion',0.10);
```

Generate the test data.

```matlab
n = 1000;
r = sort(-10 + 20*rand(n,1));
xtest = 1 + r*5e-2 + sin(r)./r + 0.4*randn(n,1);
```

Predict the test data using the trained autoencoder, `autoenc`.

```matlab
xReconstructed = predict(autoenc,xtest');
```

Plot the actual test data and the predictions.

```matlab
figure;
plot(xtest,'r.');
hold on
plot(xReconstructed,'go');
```
Reconstruct Handwritten Digit Images Using Sparse Autoencoder

Load the training data.

\[ X_{\text{Train}} = \text{digitTrainCellArrayData}; \]

The training data is a 1-by-5000 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.
hiddenSize = 25;
autoenc = trainAutoencoder(XTrain,hiddenSize,...
    'L2WeightRegularization',0.004,...
    'SparsityRegularization',4,...
    'SparsityProportion',0.15);

Load the test data.
XTest = digitTestCellArrayData;

The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, autoenc.
xReconstructed = predict(autoenc,XTest);

View the actual test data.
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(XTest{i});
end
View the reconstructed test data.

```matlab
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(xReconstructed{i});
end
```
Input Arguments

*X — Training data*
matrix | cell array of image data

Training data, specified as a matrix of training samples or a cell array of image data. If *X* is a matrix, then each column contains a single sample. If *X* is a cell array of image data, then the data in each cell must have the same number of dimensions. The image data can be pixel intensity data for gray images, in which case, each cell contains an *m*-by-*n* matrix. Alternatively, the image data can be RGB data, in which case, each cell contains an *m*-by-*n*-3 matrix.
Data Types: single | double | cell

**hiddenSize — Size of hidden representation of the autoencoder**

10 (default) | positive integer value

Size of hidden representation of the autoencoder, specified as a positive integer value. This number is the number of neurons in the hidden layer.

Data Types: single | double

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`

Example: `'EncoderTransferFunction','satlin','L2WeightRegularization',0.05` specifies the transfer function for the encoder as the positive saturating linear transfer function and the L2 weight regularization as 0.05.

**EncoderTransferFunction — Transfer function for the encoder**

'logsig' (default) | 'satlin'

Transfer function for the encoder, specified as the comma-separated pair consisting of `'EncoderTransferFunction'` and one of the following.

<table>
<thead>
<tr>
<th>Transfer Function Option</th>
<th>Definition</th>
</tr>
</thead>
</table>
| 'logsig'                 | Logistic sigmoid function  
  \[ f(z) = \frac{1}{1 + e^{-z}} \] |
| 'satlin'                 | Positive saturating linear transfer function  
  \[ f(z) = \begin{cases} 
  0, & \text{if } z \leq 0 \\
  z, & \text{if } 0 < z < 1 \\
  1, & \text{if } z \geq 1 
\end{cases} \] |

Example: 'EncoderTransferFunction','satlin'
**DecoderTransferFunction** — Transfer function for the decoder

'logsig' (default) | 'satlin' | 'purelin'

Transfer function for the decoder, specified as the comma-separated pair consisting of 'DecoderTransferFunction' and one of the following.

<table>
<thead>
<tr>
<th>Transfer Function Option</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>'logsig'</td>
<td>Logistic sigmoid function $f(z) = \frac{1}{1 + e^{-z}}$</td>
</tr>
<tr>
<td>'satlin'</td>
<td>Positive saturating linear transfer function $f(z) = \begin{cases} 0, &amp; \text{if } z \leq 0 \ z, &amp; \text{if } 0 &lt; z &lt; 1 \ 1, &amp; \text{if } z \geq 1 \end{cases}$</td>
</tr>
<tr>
<td>'purelin'</td>
<td>Linear transfer function $f(z) = z$</td>
</tr>
</tbody>
</table>

Example: 'DecoderTransferFunction', 'purelin'

**MaxEpochs** — Maximum number of training epochs

1000 (default) | positive integer value

Maximum number of training epochs or iterations, specified as the comma-separated pair consisting of 'MaxEpochs' and a positive integer value.

Example: 'MaxEpochs', 1200

**L2WeightRegularization** — The coefficient for the $L_2$ weight regularizer

0.001 (default) | a positive scalar value

The coefficient for the $L_2$ weight regularizer on page 1-617 in the cost function (LossFunction), specified as the comma-separated pair consisting of 'L2WeightRegularization' and a positive scalar value.

Example: 'L2WeightRegularization', 0.05

**LossFunction** — Loss function to use for training

'msesparse' (default)
Loss function to use for training, specified as the comma-separated pair consisting of 'LossFunction' and 'msesparse'. It corresponds to the mean squared error function adjusted for training a sparse autoencoder on page 1-616 as follows:

\[ E = \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} (x_{kn} - \hat{x}_{kn})^2 + \lambda \ast \Omega_{weights}^{L2} + \beta \ast \Omega_{sparsity}^{sparsity} \]

where \( \lambda \) is the coefficient for the \( L_2 \) regularization term on page 1-617 and \( \beta \) is the coefficient for the sparsity regularization term on page 1-617. You can specify the values of \( \lambda \) and \( \beta \) by using the L2WeightRegularization and SparsityRegularization name-value pair arguments, respectively, while training an autoencoder.

**ShowProgressWindow — Indicator to show the training window**
true (default) | false

Indicator to show the training window, specified as the comma-separated pair consisting of 'ShowProgressWindow' and either true or false.

Example: 'ShowProgressWindow',false

**SparsityProportion — Desired proportion of training examples a neuron reacts to**
0.05 (default) | positive scalar value in the range from 0 to 1

Desired proportion of training examples a neuron reacts to, specified as the comma-separated pair consisting of 'SparsityProportion' and a positive scalar value. Sparsity proportion is a parameter of the sparsity regularizer. It controls the sparsity of the output from the hidden layer. A low value for SparsityProportion usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples. Hence, a low sparsity proportion encourages higher degree of sparsity. See Sparse Autoencoders on page 1-616.

Example: 'SparsityProportion',0.01 is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples.

**SparsityRegularization — Coefficient that controls the impact of the sparsity regularizer**
1 (default) | a positive scalar value
Coefficient that controls the impact of the sparsity regularizer on page 1-617 in the cost function, specified as the comma-separated pair consisting of 'SparsityRegularization' and a positive scalar value.

Example: 'SparsityRegularization',1.6

**TrainingAlgorithm — The algorithm to use for training the autoencoder**
'trainscg' (default)

The algorithm to use for training the autoencoder, specified as the comma-separated pair consisting of 'TrainingAlgorithm' and 'trainscg'. It stands for scaled conjugate gradient descent [1].

**ScaleData — Indicator to rescale the input data**
true (default) | false

Indicator to rescale the input data, specified as the comma-separated pair consisting of 'ScaleData' and either true or false.

Autoencoders attempt to replicate their input at their output. For it to be possible, the range of the input data must match the range of the transfer function for the decoder. trainAutoencoder automatically scales the training data to this range when training an autoencoder. If the data was scaled while training an autoencoder, the predict, encode, and decode methods also scale the data.

Example: 'ScaleData',false

**UseGPU — Indicator to use GPU for training**
false (default) | true

Indicator to use GPU for training, specified as the comma-separated pair consisting of 'UseGPU' and either true or false.

Example: 'UseGPU',true

**Output Arguments**

**autoenc — Trained autoencoder**
Autoencoder object

Trained autoencoder, returned as an Autoencoder object. For information on the properties and methods of this object, see Autoencoder class page.
More About

Autoencoders

An autoencoder is a neural network which is trained to replicate its input at its output. Autoencoders can be used as tools to learn deep neural networks. Training an autoencoder is unsupervised in the sense that no labeled data is needed. The training process is still based on the optimization of a cost function. The cost function measures the error between the input $x$ and its reconstruction at the output $\hat{x}$.

An autoencoder is composed of an encoder and a decoder. The encoder and decoder can have multiple layers, but for simplicity consider that each of them has only one layer.

If the input to an autoencoder is a vector $x \in \mathbb{R}^{D_x}$, then the encoder maps the vector $x$ to another vector $z \in \mathbb{R}^{D^{(1)}}$ as follows:

$$z = h^{(1)}(W^{(1)}x + b^{(1)})$$

where the superscript (1) indicates the first layer. $h^{(1)}: \mathbb{R}^{D^{(1)}} \rightarrow \mathbb{R}^{D^{(1)}}$ is a transfer function for the encoder, $W^{(1)} \in \mathbb{R}^{D^{(1)} \times D_x}$ is a weight matrix, and $b^{(1)} \in \mathbb{R}^{D^{(1)}}$ is a bias vector.

Then, the decoder maps the encoded representation $z$ back into an estimate of the original input vector, $x$, as follows:

$$\hat{x} = h^{(2)}(W^{(2)}z + b^{(2)})$$

where the superscript (2) represents the second layer. $h^{(2)}: \mathbb{R}^{D_x} \rightarrow \mathbb{R}^{D_x}$ is the transfer function for the decoder, $W^{(1)} \in \mathbb{R}^{D_x \times D^{(1)}}$ is a weight matrix, and $b^{(2)} \in \mathbb{R}^{D_x}$ is a bias vector.

Sparse Autoencoders

Encouraging sparsity of an autoencoder is possible by adding a regularizer to the cost function [2]. This regularizer is a function of the average output activation value of a neuron. The average output activation measure of a neuron $i$ is defined as:

$$f_{avg}(i) = \frac{1}{D_x} \sum_{j=1}^{D_x} a_{ji}$$
\[ \hat{\rho}_i = \frac{1}{n} \sum_{j=1}^{n} z_i^{(1)}(x_j) = \frac{1}{n} \sum_{j=1}^{n} h(w_i^{(1)^T}x_j + b_i^{(1)}) \]

where \( n \) is the total number of training examples. \( x_j \) is the \( j \)th training example, \( w_i^{(1)^T} \) is the \( i \)th row of the weight matrix \( W^{(1)} \), and \( b_i^{(1)} \) is the \( i \)th entry of the bias vector, \( b^{(1)} \). A neuron is considered to be ‘firing’, if its output activation value is high. A low output activation value means that the neuron in the hidden layer fires in response to a small number of the training examples. Adding a term to the cost function that constrains the values of \( \hat{\rho}_i \) to be low encourages the autoencoder to learn a representation, where each neuron in the hidden layer fires to a small number of training examples. That is, each neuron specializes by responding to some feature that is only present in a small subset of the training examples.

**Sparsity Regularization**

Sparsity regularizer attempts to enforce a constraint on the sparsity of the output from the hidden layer. Sparsity can be encouraged by adding a regularization term that takes a large value when the average activation value, \( \hat{\rho}_i \), of a neuron \( i \) and its desired value, \( \rho \), are not close in value \[2\]. One such sparsity regularization term can be the Kullback-Leibler divergence.

\[
\Omega_{\text{Sparsity}} = \sum_{i=1}^{D^{(1)}} KL(\rho \parallel \hat{\rho}_i) = \sum_{i=1}^{D^{(1)}} \rho \log \left( \frac{\rho}{\hat{\rho}_i} \right) + (1-\rho) \log \left( \frac{1-\rho}{1-\hat{\rho}_i} \right)
\]

Kullback-Leibler divergence is a function for measuring how different two distributions are. In this case, it takes the value zero when \( \rho \) and \( \hat{\rho}_i \) are equal to each other, and becomes larger as they diverge from each other. Minimizing the cost function forces this term to be small, hence \( \rho \) and \( \hat{\rho}_i \) to be close to each other. You can define the desired value of the average activation value using the *SparsityProportion* name-value pair argument while training an autoencoder.

**L_2 Regularization**

When training a sparse autoencoder, it is possible to make the sparsity regulariser small by increasing the values of the weights \( w^{(i)} \) and decreasing the values of \( z^{(1)} \) \[2\]. Adding a regularization term on the weights to the cost function prevents it from happening. This term is called the L_2 regularization term and is defined by:
\[ \Omega_{\text{weights}} = \frac{1}{2} \sum_{l} \sum_{j} \sum_{i} \left( w_{ji}^{(l)} \right)^2 , \]

where \( L \) is the number of hidden layers, \( n \) is the number of observations (examples), and \( k \) is the number of variables in the training data.

**Cost Function**

The cost function for training a sparse autoencoder on page 1-616 is an adjusted mean squared error function as follows:

\[ E = \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} (x_{kn} - \hat{x}_{kn})^2 + \lambda * \Omega_{\text{weights}}^{L_2 \text{ regularization}} + \beta * \Omega_{\text{sparsity}}^{\text{sparsity regularization}} , \]

where \( \lambda \) is the coefficient for the \( L_2 \) regularization term on page 1-617 and \( \beta \) is the coefficient for the sparsity regularization term on page 1-617. You can specify the values of \( \lambda \) and \( \beta \) by using the \texttt{L2WeightRegularization} and \texttt{SparsityRegularization} name-value pair arguments, respectively, while training an autoencoder.

**References**


**See Also**

\texttt{Autoencoder} | \texttt{encode} | \texttt{stack} | \texttt{trainSoftmaxLayer}

**Topics**

“Train Stacked Autoencoders for Image Classification”

**Introduced in R2015b**
trainSoftmaxLayer

Train a softmax layer for classification

Syntax

net = trainSoftmaxLayer(X,T)
net = trainSoftmaxLayer(X,T,Name,Value)

Description

net = trainSoftmaxLayer(X,T) trains a softmax layer, net, on the input data X and the targets T.

net = trainSoftmaxLayer(X,T,Name,Value) trains a softmax layer, net, with additional options specified by one or more of the Name,Value pair arguments.

For example, you can specify the loss function.

Examples

Classify Using Softmax Layer

Load the sample data.

[X,T] = iris_dataset;

X is a 4x150 matrix of four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

T is a 3x150 matrix of associated class vectors defining which of the three classes each input is assigned to. Each row corresponds to a dummy variable representing one of the iris species (classes). In each column, a 1 in one of the three rows represents the class that particular sample (observation or example) belongs to. There is a zero in the rows for the other classes that the observation does not belong to.
Train a softmax layer using the sample data.

```matlab
net = trainSoftmaxLayer(X,T);
```

Classify the observations into one of the three classes using the trained softmax layer.

```matlab
Y = net(X);
```

Plot the confusion matrix using the targets and the classifications obtained from the softmax layer.

```matlab
plotconfusion(T,Y);
```
**Input Arguments**

**X — Training data**

* m-by-n matrix

Training data, specified as an m-by-n matrix, where *m* is the number of variables in training data, and *n* is the number of observations (examples). Hence, each column of *X* represents a sample.

Data Types: single | double

**T — Target data**

* k-by-n matrix

Target data, specified as a k-by-n matrix, where *k* is the number of classes, and *n* is the number of observations. Each row is a dummy variable representing a particular class. In other words, each column represents a sample, and all entries of a column are zero except for a single one in a row. This single entry indicates the class for that sample.

Data Types: single | double

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'MaxEpochs',400,'ShowProgressWindow',false specifies the maximum number of iterations as 400 and hides the training window.

**MaxEpochs — Maximum number of training iterations**

1000 (default) | positive integer value

Maximum number of training iterations, specified as the comma-separated pair consisting of 'MaxEpochs' and a positive integer value.

Example: 'MaxEpochs',500

Data Types: single | double

**LossFunction — Loss function for the softmax layer**

'crossentropy' (default) | 'mse'
Loss function for the softmax layer, specified as the comma-separated pair consisting of 'LossFunction' and either 'crossentropy' or 'mse'.

**mse** stands for mean squared error function, which is given by:

\[ E = \frac{1}{n} \sum_{j=1}^{n} \sum_{i=1}^{k} (t_{ij} - y_{ij})^2, \]

where \( n \) is the number of training examples, and \( k \) is the number of classes. \( t_{ij} \) is the \( ij \)th entry of the target matrix, \( T \), and \( y_{ij} \) is the \( i \)th output from the autoencoder when the input vector is \( x_j \).

The cross entropy function is given by:

\[ E = \frac{1}{n} \sum_{j=1}^{n} \sum_{i=1}^{k} t_{ij}\ln y_{ij} + (1 - t_{ij})\ln(1 - y_{ij}). \]

Example: 'LossFunction','mse'

**ShowProgressWindow** — Indicator to display the training window

true (default) | false

Indicator to display the training window during training, specified as the comma-separated pair consisting of 'ShowProgressWindow' and either true or false.

Example: 'ShowProgressWindow',false

Data Types: logical

**TrainingAlgorithm** — Training algorithm

'trainscg' (default)

Training algorithm used to train the softmax layer, specified as the comma-separated pair consisting of 'TrainingAlgorithm' and 'trainscg', which stands for scaled conjugate gradient.

Example: 'TrainingAlgorithm','trainscg'
Output Arguments

net — Softmax layer for classification
network object

Softmax layer for classification, returned as a network object. The softmax layer, net, is the same size as the target T.

See Also

stack | trainAutoencoder

Introduced in R2015b
Autoencoder class

Autoencoder class

Description

An Autoencoder object contains an autoencoder network, which consists of an encoder and a decoder. The encoder maps the input to a hidden representation. The decoder attempts to map this representation back to the original input.

Construction

autoenc = trainAutoencoder(X) returns an autoencoder trained using the training data in X.

autoenc = trainAutoencoder(X,hiddenSize) returns an autoencoder with the hidden representation size of hiddenSize.

autoenc = trainAutoencoder(____,Name,Value) returns an autoencoder for any of the above input arguments with additional options specified by one or more name-value pair arguments.

Input Arguments

X — Training data
matrix | cell array of image data

Training data, specified as a matrix of training samples or a cell array of image data. If X is a matrix, then each column contains a single sample. If X is a cell array of image data, then the data in each cell must have the same number of dimensions. The image data can be pixel intensity data for gray images, in which case, each cell contains an $m$-by-$n$ matrix. Alternatively, the image data can be RGB data, in which case, each cell contains an $m$-by-$n$-3 matrix.

Data Types: single | double | cell
**hiddenSize — Size of hidden representation of the autoencoder**
10 (default) | positive integer value

Size of hidden representation of the autoencoder, specified as a positive integer value. This number is the number of neurons in the hidden layer.

Data Types: single | double

**Properties**

**HiddenSize — Size of the hidden representation**
a positive integer value

Size of the hidden representation in the hidden layer of the autoencoder, stored as a positive integer value.

Data Types: double

**EncoderTransferFunction — Name of the transfer function for the encoder**
string

Name of the transfer function for the encoder, stored as a string.

Data Types: char

**EncoderWeights — Weights for the encoder**
matrix

Weights for the encoder, stored as a matrix.

Data Types: double

**EncoderBiases — Bias values for the encoder**
vector

Bias values for the encoder, stored as a vector.

Data Types: double

**DecoderTransferFunction — Name of the transfer function for the decoder**
string

Name of the transfer function for the decoder, stored as a string.
Data Types: char

**DecoderWeights — Weights for the decoder matrix**

Weights for the decoder, stored as a matrix.

Data Types: double

**DecoderBiases — Bias values for the decoder vector**

Bias values for the decoder, stored as a vector.

Data Types: double

**TrainingParameters — Parameters that trainAutoencoder uses for training the autoencoder structure**

Parameters that trainAutoencoder uses for training the autoencoder, stored as a structure.

Data Types: struct

**ScaleData — Indicator for data that is rescaled**

true or 1 (default) | false or 0

Indicator for data that is rescaled while passing to the autoencoder, stored as either true or false.

Autoencoders attempt to replicate their input at their output. For it to be possible, the range of the input data must match the range of the transfer function for the decoder. trainAutoencoder automatically scales the training data to this range when training an autoencoder. If the data was scaled while training an autoencoder, the predict, encode, and decode methods also scale the data.

Data Types: logical
Methods

- **decode**: Decode encoded data
- **encode**: Encode input data
- **generateFunction**: Generate a MATLAB function to run the autoencoder
- **generateSimulink**: Generate a Simulink model for the autoencoder
- **network**: Convert *Autoencoder* object into *network* object
- **plotWeights**: Plot a visualization of the weights for the encoder of an autoencoder
- **predict**: Reconstruct the inputs using trained autoencoder
- **stack**: Stack encoders from several autoencoders together
- **view**: View autoencoder

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

- trainAutoencoder

Topics

- Class Attributes (MATLAB)
- Property Attributes (MATLAB)

Introduced in R2015b
decode

Class: Autoencoder

Decode encoded data

Syntax

Y = decode(autoenc,Z)

Description

Y = decode(autoenc,Z) returns the decoded data on page 1-630 Y, using the autoencoder autoenc.

Input Arguments

autoenc — Trained autoencoder

Autoencoder object

Trained autoencoder, returned by the trainAutoencoder function as an object of the Autoencoder class.

Z — Data encoded by autoenc

matrix

Data encoded by autoenc, specified as a matrix. Each column of Z represents an encoded sample (observation).

Data Types: single | double

Output Arguments

Y — Decoded data

matrix | cell array of image data
Decoded data, returned as a matrix or a cell array of image data.

If the autoencoder `autoenc` was trained on a cell array of image data, then `Y` is also a cell array of images.

If the autoencoder `autoenc` was trained on a matrix, then `Y` is also a matrix, where each column of `Y` corresponds to one sample or observation.

**Examples**

**Decode Encoded Data For New Images**

Load the training data.

```matlab
X = digitTrainCellArrayData;
```

`X` is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder using the training data with a hidden size of 15.

```matlab
hiddenSize = 15;
autoenc = trainAutoencoder(X,hiddenSize);
```

Extract the encoded data for new images using the autoencoder.

```matlab
Xnew = digitTestCellArrayData;
features = encode(autoenc,Xnew);
```

Decode the encoded data from the autoencoder.

```matlab
Y = decode(autoenc,features);
```

`Y` is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

**Algorithms**

If the input to an autoencoder is a vector \( x \in \mathbb{R}^{D_x} \), then the encoder maps the vector \( x \) to another vector \( z \in \mathbb{R}^{D^{(1)}} \) as follows:
\[ z = h^{(1)}(W^{(1)}x + b^{(1)}) , \]

where the superscript (1) indicates the first layer. \( h^{(1)}: \mathbb{R}^{D^{(1)}} \to \mathbb{R}^{D^{(1)}} \) is a transfer function for the encoder, \( W^{(1)} \in \mathbb{R}^{D^{(1)} \times D_x} \) is a weight matrix, and \( b^{(1)} \in \mathbb{R}^{D^{(1)}} \) is a bias vector. Then, the decoder maps the encoded representation \( z \) back into an estimate of the original input vector, \( x \), as follows:

\[ \hat{x} = h^{(2)}(W^{(2)}z + b^{(2)}) , \]

where the superscript (2) represents the second layer. \( h^{(2)}: \mathbb{R}^{D_x} \to \mathbb{R}^{D_x} \) is the transfer function for the decoder, \( W^{(1)} \in \mathbb{R}^{D_x \times D^{(1)}} \) is a weight matrix, and \( b^{(2)} \in \mathbb{R}^{D_x} \) is a bias vector.

**See Also**

encode | trainAutoencoder

**Introduced in R2015b**
**encode**

**Class:** Autoencoder

Encode input data

**Syntax**

\[ Z = \text{encode}(\text{autoenc}, \text{Xnew}) \]

**Description**

\[ Z = \text{encode}(\text{autoenc}, \text{Xnew}) \] returns the encoded data on page 1-633, \( Z \), for the input data \( \text{Xnew} \), using the autoencoder, \( \text{autoenc} \).

**Input Arguments**

- **autoenc** — Trained autoencoder
  
  Autoencoder object

  Trained autoencoder, returned as an object of the Autoencoder class.

- **Xnew** — Input data
  
  matrix | cell array of image data | array of single image data

  Input data, specified as a matrix of samples, a cell array of image data, or an array of single image data.

  If the autoencoder \( \text{autoenc} \) was trained on a matrix, where each column represents a single sample, then \( \text{Xnew} \) must be a matrix, where each column represents a single sample.

  If the autoencoder \( \text{autoenc} \) was trained on a cell array of images, then \( \text{Xnew} \) must either be a cell array of image data or an array of single image data.

  **Data Types:** single | double | cell
Output Arguments

**Z — Data encoded by autoenc matrix**

Data encoded by autoenc, specified as a matrix. Each column of Z represents an encoded sample (observation).

Data Types: single | double

Examples

**Encode Decoded Data for New Images**

Load the sample data.

```
X = digitTrainCellArrayData;
```

X is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden size of 50 using the training data.

```
autoenc = trainAutoencoder(X,50);
```

Encode decoded data for new image data.

```
Xnew = digitTestCellArrayData;
Z = encode(autoenc,Xnew);
```

Xnew is a 1-by-5000 cell array. Z is a 50-by-5000 matrix, where each column represents the image data of one handwritten digit in the new data Xnew.

Algorithms

If the input to an autoencoder is a vector \( x \in \mathbb{R}^{D_x} \), then the encoder maps the vector \( x \) to another vector \( z \in \mathbb{R}^{D_1} \) as follows:
\[ z = h^{(1)}(W^{(1)}x + b^{(1)}), \]

where the superscript (1) indicates the first layer. \( h^{(1)}: \mathbb{R}^{D^{(1)}} \rightarrow \mathbb{R}^{D^{(1)}} \) is a transfer function for the encoder, \( W^{(1)} \in \mathbb{R}^{D^{(1)} \times D_x} \) is a weight matrix, and \( b^{(1)} \in \mathbb{R}^{D^{(1)}} \) is a bias vector.

**See Also**

decode | stack | trainAutoencoder

*Introduced in R2015b*
generateFunction

**Class:** Autoencoder

Generate a MATLAB function to run the autoencoder

**Syntax**

generateFunction(autoenc)
generateFunction(autoenc,pathname)
generateFunction(autoenc,pathname,Name,Value)

**Description**

generateFunction(autoenc) generates a complete stand-alone function in the current directory, to run the autoencoder autoenc on input data.

generateFunction(autoenc,pathname) generates a complete stand-alone function to run the autoencoder autoenc on input data in the location specified by pathname.

generateFunction(autoenc,pathname,Name,Value) generates a complete stand-alone function with additional options specified by the Name,Value pair argument.

**Input Arguments**

**autoenc — Trained autoencoder**  
Autoencoder object  

Trained autoencoder, returned as an object of the Autoencoder class.

**pathname — Location for generated function**  
string  

Location for generated function, specified as a string.

Example: 'C:\MyDocuments\Autoencoders'
Data Types: char

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**ShowLinks — Indicator to display the links to the generated code**
false (default) | true

Indicator to display the links to the generated code in the command window, specified as the comma-separated pair consisting of 'ShowLinks' and either true or false.

Example: 'ShowLinks',true

Data Types: logical

**Examples**

**Generate MATLAB Function for Running Autoencoder**

Load the sample data.

```matlab
X = iris_dataset;
```

Train an autoencoder with 4 neurons in the hidden layer.

```matlab
autoenc = trainAutoencoder(X,4);
```

Generate the code for running the autoencoder. Show the links to the MATLAB function.

```matlab
generateFunction(autoenc)
```

MATLAB function generated: neural_function.m
To view generated function code: edit neural_function
For examples of using function: help neural_function

Generate the code for the autoencoder in a specific path.

```matlab
generateFunction(autoenc,'H:\Documents\Autoencoder')
```
MATLAB function generated: H:\Documents\Autoencoder.m
To view generated function code: edit Autoencoder
For examples of using function: help Autoencoder

Tips

- If you do not specify the path and the file name, generateFunction, by default, creates the code in an m-file with the name neural_function.m. You can change the file name after generateFunction generates it. Or you can specify the path and file name using the pathname input argument in the call to generateFunction.

See Also
genFunction | generateSimulink

Introduced in R2015b
generateSimulink

Class: Autoencoder

Generate a Simulink model for the autoencoder

Syntax

generateSimulink(autoenc)

Description

generateSimulink(autoenc) creates a Simulink model for the autoencoder, autoenc.

Input Arguments

autoenc — Trained autoencoder
Autoencoder object

Trained autoencoder, returned as an object of the Autoencoder class.

Examples

Generate Simulink Model for Autoencoder

Load the training data.

X = digitsmall_dataset;

The training data is a 1-by-500 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.
hiddenSize = 25;
autoenc = trainAutoencoder(X,hiddenSize,...
    'L2WeightRegularization',0.004,...
    'SparsityRegularization',4,...
    'SparsityProportion',0.15);

Create a Simulink model for the autoencoder, autoenc.

generateSimulink(autoenc)
Introduced in R2015b
**network**

**Class:** Autoencoder

Convert Autoencoder object into network object

**Syntax**

```
net = network(autoenc)
```

**Description**

`net = network(autoenc)` returns a network object which is equivalent to the autoencoder `autoenc`.

**Input Arguments**

- `autoenc` — Trained autoencoder
  
  Autoencoder object

  Trained autoencoder, returned as an object of the `Autoencoder` class.

**Output Arguments**

- `net` — Neural network
  
  Network object

  Neural network, that is equivalent to the autoencoder `autoenc`, returned as an object of the `network` class.

**Examples**
Create Network from Autoencoder

Load the sample data.

\[
X = \text{bodyfat
dataset};
\]

\[
X = \text{bodyfat
dataset};
\]

\(X\) is a 13-by-252 matrix defining thirteen attributes of 252 different neighborhoods. For more information on the data, type \texttt{help house
dataset} in the command line.

Train an autoencoder on the attribute data.

\[
\text{autoenc} = \text{trainAutoencoder}(X);
\]

Create a network object from the autoencoder, \texttt{autoenc}.

\[
\text{net} = \text{network(autoenc)};
\]

Predict the attributes using the network, \texttt{net}.

\[
X\text{pred} = \text{net}(X);
\]

Fit a linear regression model between the actual and estimated attributes data. Compute the estimated Pearson correlation coefficient, the slope and the intercept (bias) of the regression model, using all attribute data as one data set.

\[
[C, S, B] = \text{regression}(X, X\text{pred}, 'one')
\]

\(C = 0.9996\)

\(S = 0.9982\)

\(B = 0.1195\)

The correlation coefficient is almost 1, which indicates that the attributes data and the estimations from the neural network are highly close to each other.

Plot the actual data and the fitted line.

\[
\text{plotregression}(X, X\text{pred});
\]
The data appears to be on the fitted line, which visually supports the conclusion that the predictions are very close to the actual data.

**See Also**

Autoencoder | trainAutoencoder
Introduced in R2015b
plotWeights

Class: Autoencoder

Plot a visualization of the weights for the encoder of an autoencoder

Syntax

plotWeights(autoenc)
h = plotWeights(autoenc)

Description

plotWeights(autoenc) visualizes the weights for the autoencoder, autoenc.

h = plotWeights(autoenc) returns a function handle h, for the visualization of the encoder weights for the autoencoder, autoenc.

Input Arguments

autoenc — Trained autoencoder
Autoencoder object

Trained autoencoder, returned as an object of the Autoencoder class.

Output Arguments

h — Image object
handle

Image object, returned as a handle.
Examples

Visualize Learned Features

Load the training data.

\[ X = \text{digitTrainCellArrayData}; \]

The training data is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer of 25 neurons.

\[
\text{hiddenSize} = 25; \\
\text{autoenc} = \text{trainAutoencoder}(X, \text{hiddenSize}, ...) \\
    \quad \text{''L2WeightRegularization'', 0.004, ...} \\
    \quad \text{''SparsityRegularization'', 4, ...} \\
    \quad \text{''SparsityProportion'', 0.2});
\]

Visualize the learned features.

\[ \text{plotWeights(autoenc);} \]
Tips

- `plotWeights` allows the visualization of the features that the autoencoder learns. Use it when the autoencoder is trained on image data. The visualization of the weights has the same dimensions as the images used for training.

See Also

`trainAutoencoder`
Introduced in R2015b
predict

Class: Autoencoder

Reconstruct the inputs using trained autoencoder

Syntax

\[ Y = \text{predict}(\text{autoenc}, X) \]

Description

\[ Y = \text{predict}(\text{autoenc}, X) \] returns the predictions \( Y \) for the input data \( X \), using the autoencoder \( \text{autoenc} \). The result \( Y \) is a reconstruction of \( X \).

Input Arguments

\texttt{autoenc} — Trained autoencoder

Autoencoder object

Trained autoencoder, returned as an object of the \texttt{Autoencoder} class.

\texttt{Xnew} — Input data

matrix | cell array of image data | array of single image data

Input data, specified as a matrix of samples, a cell array of image data, or an array of single image data.

If the autoencoder \texttt{autoenc} was trained on a matrix, where each column represents a single sample, then \texttt{Xnew} must be a matrix, where each column represents a single sample.

If the autoencoder \texttt{autoenc} was trained on a cell array of images, then \texttt{Xnew} must either be a cell array of image data or an array of single image data.

Data Types: \texttt{single} | \texttt{double} | \texttt{cell}
Output Arguments

Y — Predictions for the input data Xnew
matrix | cell array of image data | array of single image data

Predictions for the input data Xnew, returned as a matrix or a cell array of image data.

• If Xnew is a matrix, then Y is also a matrix, where each column corresponds to a single sample (observation or example).
• If Xnew is a cell array of image data, then Y is also a cell array of image data, where each cell contains the data for a single image.
• If Xnew is an array of a single image data, then Y is also an array of a single image data.

Examples

Predict Continuous Measurements Using Trained Autoencoder

Load the training data.

X = iris_dataset;

The training data contains measurements on four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

Train an autoencoder on the training data using the positive saturating linear transfer function in the encoder and linear transfer function in the decoder.

autoenc = trainAutoencoder(X,'EncoderTransferFunction','satlin','DecoderTransferFunction','purelin');

Reconstruct the measurements using the trained network, autoenc.

xReconstructed = predict(autoenc,X);

Plot the predicted measurement values along with the actual values in the training dataset.

for i = 1:4
    h(i) = subplot(1,4,i);

The red dots represent the training data and the green circles represent the reconstructed data.
Reconstruct Handwritten Digit Images Using Sparse Autoencoder

Load the training data.

\[ \text{XTrain} = \text{digitTrainCellArrayData}; \]

The training data is a 1-by-5000 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer containing 25 neurons.

\[ \text{hiddenSize} = 25; \]
\[ \text{autoenc} = \text{trainAutoencoder}(	ext{XTrain}, \text{hiddenSize}, ...) \]
\[ \quad \text{L2WeightRegularization}', 0.004, ... \]
\[ \quad \text{'SparsityRegularization}', 4, ... \]
\[ \quad \text{'SparsityProportion}', 0.15); \]

Load the test data.

\[ \text{XTest} = \text{digitTestCellArrayData}; \]

The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, autoenc.

\[ \text{xReconstructed} = \text{predict}(	ext{autoenc}, \text{XTest}); \]

View the actual test data.

\[ \text{figure}; \]
\[ \text{for i = 1:20} \]
\[ \quad \text{subplot}(4, 5, i); \]
\[ \quad \text{imshow}(	ext{XTest}{i}); \]
\[ \text{end} \]
View the reconstructed test data.

```matlab
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(xReconstructed{i});
end
```
See Also

trainAutoencoder

Introduced in R2015b
stack

Class: Autoencoder

Stack encoders from several autoencoders together

Syntax

stackednet = stack(autoenc1,autoenc2,...)
stackednet = stack(autoenc1,autoenc2,...,net1)

Description

stackednet = stack(autoenc1,autoenc2,...) returns a network object created by stacking the encoders of the autoencoders, autoenc1, autoenc2, and so on.

stackednet = stack(autoenc1,autoenc2,...,net1) returns a network object created by stacking the encoders of the autoencoders and the network object net1.

The autoencoders and the network object can be stacked only if their dimensions match.

Input Arguments

autoenc1 — Trained autoencoder
Autoencoder object

Trained autoencoder, specified as an Autoencoder object.

autoenc2 — Trained autoencoder
Autoencoder object

Trained autoencoder, specified as an Autoencoder object.

net1 — Trained neural network
network object
Trained neural network, specified as a network object. net1 can be a softmax layer, trained using the trainSoftmaxLayer function.

Output Arguments

**stackednet — Stacked neural network**

*network object*

Stacked neural network (deep network), returned as a network object.

Examples

Create a Stacked Network

Load the training data.

```
[X,T] = iris_dataset;
```

Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
hiddenSize = 5;
autoenc = trainAutoencoder(X, hiddenSize, ...
    'L2WeightRegularization', 0.001, ...
    'SparsityRegularization', 4, ...
    'SparsityProportion', 0.05, ...)

features = encode(autoenc,X);
```

Extract the features in the hidden layer.

```
features = encode(autoenc,X);
```

Train a softmax layer for classification using the features.

```
softnet = trainSoftmaxLayer(features,T);
```

Stack the encoder and the softmax layer to form a deep network.

```
stackednet = stack(autoenc,softnet);
```
View the stacked network.

view(stackednet);

**Tips**

- The size of the hidden representation of one autoencoder must match the input size of the next autoencoder or network in the stack.

  The first input argument of the stacked network is the input argument of the first autoencoder. The output argument from the encoder of the first autoencoder is the input of the second autoencoder in the stacked network. The output argument from the encoder of the second autoencoder is the input argument to the third autoencoder in the stacked network, and so on.

- The stacked network object `stacknet` inherits its training parameters from the final input argument `net1`.

**See Also**

Autoencoder | trainAutoencoder

**Topics**

“Train Stacked Autoencoders for Image Classification”

**Introduced in R2015b**
view

Class: Autoencoder

View autoencoder

Syntax

view(autoenc)

Description

view(autoenc) returns a diagram of the autoencoder, autoenc.

Input Arguments

autoenc — Trained autoencoder
Autoencoder object

Trained autoencoder, returned as an object of the Autoencoder class.

Examples

View Autoencoder

Load the training data.
X = iris_dataset;

Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

hiddenSize = 5;
autoenc = trainAutoencoder(X, hiddenSize, ...
'L2WeightRegularization',0.001, ...
'SparsityRegularization',4, ...
'SparsityProportion',0.05, ...
'DecoderTransferFunction','purelin');

View the autoencoder:

view(autoenc)

See Also
trainAutoencoder

Introduced in R2015b
classificationLayer

Classification output layer

Syntax

layer = classificationLayer
layer = classificationLayer(Name,Value)

Description

A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.

The layer infers the number of classes from the output size of the previous layer. For example, to specify the number of classes $K$ of the network, include a fully connected layer with output size $K$ and a softmax layer before the classification layer.

layer = classificationLayer creates a classification layer.

layer = classificationLayer(Name,Value) sets the optional Name and Classes properties using name-value pairs. For example, classificationLayer('Name','output') creates a classification layer with the name 'output'. Enclose each property name in single quotes.

Examples

Create Classification Layer

Create a classification layer with the name 'output'.

layer = classificationLayer('Name','output')

layer = ClassificationOutputLayer with properties:
Include a classification output layer in a Layer array.

```plaintext
layers = [ ... 
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

layers = 7x1 Layer array with layers:

1 '' Image Input 28x28x1 images with 'zerocenter' normalization
2 '' Convolution 20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3 '' ReLU ReLU
4 '' Max Pooling 2x2 max pooling with stride [2 2] and padding [0 0 0 0]
5 '' Fully Connected 10 fully connected layer
6 '' Softmax softmax
7 '' Classification Output crossentropyex

**Input Arguments**

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: `classificationLayer('Name','output')` creates a classification layer with the name 'output'
Name — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

Classes — Classes of the output layer

'auto' (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If Classes is 'auto', then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors str, then the software sets the classes of the output layer to categorical(str,str). The default value is 'auto'.

Data Types: char | categorical | string | cell

Output Arguments

layer — Classification layer

ClassificationOutputLayer object

Classification layer, returned as a ClassificationOutputLayer object.

For information on concatenating layers to construct convolutional neural network architecture, see Layer.

More About

Classification Layer

A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.

For typical classification networks, the classification layer must follow the softmax layer. In the classification layer, trainNetwork takes the values from the softmax function and
assigns each input to one of the $K$ mutually exclusive classes using the cross entropy function for a 1-of-$K$ coding scheme [1]:

$$\text{loss} = - \sum_{i=1}^{N} \sum_{j=1}^{K} t_{ij} \ln y_{ij},$$

where $N$ is the number of samples, $K$ is the number of classes, $t_{ij}$ is the indicator that the $i$th sample belongs to the $j$th class, and $y_{ij}$ is the output for sample $i$ for class $j$, which in this case, is the value from the softmax function. That is, it is the probability that the network associates the $i$th input with class $j$.

**References**


**Extended Capabilities**

**GPU Code Generation**
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

**See Also**
ClassificationOutputLayer | regressionLayer | softmaxLayer

**Topics**
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

**Introduced in R2016a**
fitnet

Function fitting neural network

Syntax

net = fitnet(hiddenSizes)
net = fitnet(hiddenSizes,trainFcn)

Description

net = fitnet(hiddenSizes) returns a function fitting neural network with a hidden layer size of hiddenSizes.

net = fitnet(hiddenSizes,trainFcn) returns a function fitting neural network with a hidden layer size of hiddenSizes and training function, specified by trainFcn.

Examples

Construct and Train a Function Fitting Network

Load the training data.

[x,t] = simplefit_dataset;

The 1-by-94 matrix x contains the input values and the 1-by-94 matrix t contains the associated target output values.

Construct a function fitting neural network with one hidden layer of size 10.

net = fitnet(10);

View the network.

view(net)
The sizes of the input and output are zero. The software adjusts the sizes of these during training according to the training data.

Train the network `net` using the training data.

```
net = train(net,x,t);
```

View the trained network.

```
view(net)
```

You can see that the sizes of the input and output are 1.

Estimate the targets using the trained network.

```
y = net(x);
```

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net,y,t)
```
perf =

1.4639e-04

The default training algorithm for a function fitting network is Levenberg-Marquardt (‘trainlm’). Use the Bayesian regularization training algorithm and compare the performance results.

```matlab
net = fitnet(10,'trainbr');
net = train(net,x,t);
y = net(x);
perf = perform(net,y,t)
```

perf =

3.3261e-10

The Bayesian regularization training algorithm improves the performance of the network in terms of estimating the target values.

### Input Arguments

**hiddenSizes — Size of the hidden layers**

10 (default) | row vector

Size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network.

Example: For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: `[10, 8, 5]`

The input and output sizes are set to zero. The software adjusts the sizes of these during training according to the training data.

Data Types: `single` | `double`

**trainFcn — Training function name**

`'trainlm'` (default) | `'trainbr'` | `'trainbfg'` | `'trainrp'` | `'trainscg'` | ...
Training function name, specified as one of the following.

<table>
<thead>
<tr>
<th>Training Function</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>'trainlm'</td>
<td>Levenberg-Marquardt</td>
</tr>
<tr>
<td>'trainbr'</td>
<td>Bayesian Regularization</td>
</tr>
<tr>
<td>'trainbfg'</td>
<td>BFGS Quasi-Newton</td>
</tr>
<tr>
<td>'trainrp'</td>
<td>Resilient Backpropagation</td>
</tr>
<tr>
<td>'trainscg'</td>
<td>Scaled Conjugate Gradient</td>
</tr>
<tr>
<td>'traincgb'</td>
<td>Conjugate Gradient with Powell/Beale Restarts</td>
</tr>
<tr>
<td>'traincfg'</td>
<td>Fletcher-Powell Conjugate Gradient</td>
</tr>
<tr>
<td>'traincgp'</td>
<td>Polak-Ribiére Conjugate Gradient</td>
</tr>
<tr>
<td>'trainoss'</td>
<td>One Step Secant</td>
</tr>
<tr>
<td>'traingdx'</td>
<td>Variable Learning Rate Gradient Descent</td>
</tr>
<tr>
<td>'traingdm'</td>
<td>Gradient Descent with Momentum</td>
</tr>
<tr>
<td>'traingd'</td>
<td>Gradient Descent</td>
</tr>
</tbody>
</table>

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: 'traingdx'

For more information on the training functions, see “Train and Apply Multilayer Shallow Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: char

Output Arguments

net — Function fitting network

network object

Function fitting network, returned as a network object.
Tips

- Function fitting is the process of training a neural network on a set of inputs in order to produce an associated set of target outputs. After you construct the network with the desired hidden layers and the training algorithm, you must train it using a set of training data. Once the neural network has fit the data, it forms a generalization of the input-output relationship. You can then use the trained network to generate outputs for inputs it was not trained on.

See Also

feedforwardnet | network | nftool | perform | train | trainlm

Topics

“Fit Data with a Shallow Neural Network”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b
trainingOptions

Options for training deep learning neural network

Syntax

options = trainingOptions(solverName)
options = trainingOptions(solverName,Name,Value)

Description

options = trainingOptions(solverName) returns training options for the optimizer specified by solverName. To train a network, use the training options as an input argument to the trainNetwork function.

options = trainingOptions(solverName,Name,Value) returns training options with additional options specified by one or more name-value pair arguments.

Examples

Specify Training Options

Create a set of options for training a network using stochastic gradient descent with momentum. Reduce the learning rate by a factor of 0.2 every 5 epochs. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Turn on the training progress plot.

options = trainingOptions('sgdm', ...
   'LearnRateSchedule','piecewise', ...,
   'LearnRateDropFactor',0.2, ...,
   'LearnRateDropPeriod',5, ...,
   'MaxEpochs',20, ...
   'MiniBatchSize',64, ...
   'Plots','training-progress')
Monitor Deep Learning Training Progress

When you train networks for deep learning, it is often useful to monitor the training progress. By plotting various metrics during training, you can learn how the training is progressing. For example, you can determine if and how quickly the network accuracy is improving, and whether the network is starting to overfit the training data.

When you specify 'training-progress' as the 'Plots' value in trainingOptions and start network training, trainNetwork creates a figure and displays training metrics at every iteration. Each iteration is an estimation of the gradient and an update of the network parameters. If you specify validation data in trainingOptions, then the figure
shows validation metrics each time `trainNetwork` validates the network. The figure plots the following:

- **Training accuracy** — Classification accuracy on each individual mini-batch.
- **Smoothed training accuracy** — Smoothed training accuracy, obtained by applying a smoothing algorithm to the training accuracy. It is less noisy than the unsmoothed accuracy, making it easier to spot trends.
- **Validation accuracy** — Classification accuracy on the entire validation set (specified using `trainingOptions`).
- **Training loss, smoothed training loss, and validation loss** — The loss on each mini-batch, its smoothed version, and the loss on the validation set, respectively. If the final layer of your network is a `classificationLayer`, then the loss function is the cross entropy loss. For more information about loss functions for classification and regression problems, see “Output Layers”.

For regression networks, the figure plots the root mean square error (RMSE) instead of the accuracy.

The figure marks each training **Epoch** using a shaded background. An epoch is a full pass through the entire data set.

During training, you can stop training and return the current state of the network by clicking the stop button in the top-right corner. For example, you might want to stop training when the accuracy of the network reaches a plateau and it is clear that the accuracy is no longer improving. After you click the stop button, it can take a while for the training to complete. Once training is complete, `trainNetwork` returns the trained network.

When training finishes, view the **Results** showing the final validation accuracy and the reason that training finished. The final validation metrics are labeled `Final` in the plots. If your network contains batch normalization layers, then the final validation metrics are often different from the validation metrics evaluated during training. This is because batch normalization layers in the final network perform different operations than during training.

On the right, view information about the training time and settings. To learn more about training options, see “Set Up Parameters and Train Convolutional Neural Network”.
Plot Training Progress During Training

Train a network and plot the training progress during training.

Load the training data, which contains 5000 images of digits. Set aside 1000 of the images for network validation.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,,:,idx);
XTrain(:,:,,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];

Construct a network to classify the digit image data.

```MATLAB
layers = [ 
    imageInputLayer([28 28 1])
];
```
Specify options for network training. To validate the network at regular intervals during training, specify validation data. Choose the 'ValidationFrequency' value so that the network is validated about once per epoch. To plot training progress during training, specify 'training-progress' as the 'Plots' value.

```matlab
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...  
    'ValidationData',{XValidation,YValidation}, ...  
    'ValidationFrequency',30, ...  
    'Verbose',false, ...  
    'Plots','training-progress');
```

Train the network.

```matlab
net = trainNetwork(XTrain,YTrain,layers,options);
```
Input Arguments

**solverName — Solver for training network**

'sgdm' | 'rmsprop' | 'adam'

Solver for training network, specified as one of the following:
• 'sgdm' — Use the stochastic gradient descent with momentum (SGDM) optimizer. You can specify the momentum value using the 'Momentum' name-value pair argument.

• 'rmsprop' — Use the RMSProp optimizer. You can specify the decay rate of the squared gradient moving average using the 'SquaredGradientDecayFactor' name-value pair argument.

• 'adam' — Use the Adam optimizer. You can specify the decay rates of the gradient and squared gradient moving averages using the 'GradientDecayFactor' and 'SquaredGradientDecayFactor' name-value pair arguments, respectively.

For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example:
'InitialLearnRate',0.03,'L2Regularization',0.0005,'LearnRateSchedule','piecewise' specifies the initial learning rate as 0.03 and the L_2 regularization factor as 0.0005, and instructs the software to drop the learning rate every given number of epochs by multiplying with a certain factor.

**Plots and Display**

**Plots — Plots to display during network training**

'none' (default) | 'training-progress'

Plots to display during network training, specified as the comma-separated pair consisting of 'Plots' and one of the following:

• 'none' — Do not display plots during training.

• 'training-progress' — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network. For more information on the training progress plot, see “Monitor Deep Learning Training Progress” on page 1-670.

Example: 'Plots','training-progress'
Verbose — Indicator to display training progress information
1 (true) (default) | 0 (false)

Indicator to display training progress information in the command window, specified as the comma-separated pair consisting of 'Verbose' and either 1 (true) or 0 (false).

The verbose output displays the following information:

**Classification Networks**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch</td>
<td>Epoch number. An epoch corresponds to a full pass of the data.</td>
</tr>
<tr>
<td>Iteration</td>
<td>Iteration number. An iteration corresponds to a mini-batch.</td>
</tr>
<tr>
<td>Time Elapsed</td>
<td>Time elapsed in hours, minutes, and seconds.</td>
</tr>
<tr>
<td>Mini-batch Accuracy</td>
<td>Classification accuracy on the mini-batch.</td>
</tr>
<tr>
<td>Validation Accuracy</td>
<td>Classification accuracy on the validation data. If you do not specify validation data, then the function does not display this field.</td>
</tr>
<tr>
<td>Mini-batch Loss</td>
<td>Loss on the mini-batch. If the output layer is a ClassificationOutputLayer object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes.</td>
</tr>
<tr>
<td>Validation Loss</td>
<td>Loss on the validation data. If the output layer is a ClassificationOutputLayer object, then the loss is the cross entropy loss for multi-class classification problems with mutually exclusive classes. If you do not specify validation data, then the function does not display this field.</td>
</tr>
<tr>
<td>Base Learning Rate</td>
<td>Base learning rate. The software multiplies the learn rate factors of the layers by this value.</td>
</tr>
</tbody>
</table>
## Regression Networks

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch</td>
<td>Epoch number. An epoch corresponds to a full pass of the data.</td>
</tr>
<tr>
<td>Iteration</td>
<td>Iteration number. An iteration corresponds to a mini-batch.</td>
</tr>
<tr>
<td>Time Elapsed</td>
<td>Time elapsed in hours, minutes, and seconds.</td>
</tr>
<tr>
<td>Mini-batch RMSE</td>
<td>Root-mean-squared-error (RMSE) on the mini-batch.</td>
</tr>
<tr>
<td>Validation RMSE</td>
<td>RMSE on the validation data. If you do not specify validation data, then the software does not display this field.</td>
</tr>
<tr>
<td>Mini-batch Loss</td>
<td>Loss on the mini-batch. If the output layer is a RegressionOutputLayer object, then the loss is the half-mean-squared-error.</td>
</tr>
<tr>
<td>Validation Loss</td>
<td>Loss on the validation data. If the output layer is a RegressionOutputLayer object, then the loss is the half-mean-squared-error. If you do not specify validation data, then the software does not display this field.</td>
</tr>
<tr>
<td>Base Learning Rate</td>
<td>Base learning rate. The software multiplies the learn rate factors of the layers by this value.</td>
</tr>
</tbody>
</table>

To specify validation data, use the 'ValidationData' name-value pair.

Example: 'Verbose',false

### VerboseFrequency — Frequency of verbose printing

50 (default) | positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as the comma-separated pair consisting of 'VerboseFrequency' and a positive integer. This option only has an effect when the 'Verbose' value equals true.
If you validate the network during training, then `trainNetwork` also prints to the command window every time validation occurs.

Example: `VerbosityFrequency',100

**Mini-Batch Options**

**MaxEpochs — Maximum number of epochs**

30 (default) | positive integer

Maximum number of epochs to use for training, specified as the comma-separated pair consisting of 'MaxEpochs' and a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

Example: `MaxEpochs',20

**MiniBatchSize — Size of mini-batch**

128 (default) | positive integer

Size of the mini-batch to use for each training iteration, specified as the comma-separated pair consisting of 'MiniBatchSize' and a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights. See “Stochastic Gradient Descent” on page 1-691.

Example: `MiniBatchSize',256

**Shuffle — Option for data shuffling**

'once' (default) | 'never' | 'every-epoch'

Option for data shuffling, specified as the comma-separated pair consisting of 'Shuffle' and one of the following:

- **'once'** — Shuffle the training and validation data once before training.
- **'never'** — Do not shuffle the data.
- **'every-epoch'** — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then `trainNetwork` discards the training data that does not fit into the final complete mini-batch of each epoch. To avoid discarding the same data every epoch, set the 'Shuffle' value to 'every-epoch'.
Example: 'Shuffle', 'every-epoch'

Validation

**ValidationData — Data to use for validation during training**
image datastore | datastore | table | cell array

Data to use for validation during training, specified as an image datastore, a datastore that returns data in a two-column table or two-column cell array, a table, or a cell array. The format of the validation data depends on the type of task and correspond to valid inputs to the `trainNetwork` function.

This option supports networks with a single input only.

**Image Data**

For image data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image datastore</td>
<td>ImageDatastore object with categorical labels.</td>
<td>imds argument of trainNetwork</td>
</tr>
<tr>
<td>Dataset</td>
<td>Datastore that returns data in a two-column table or two-column cell array, where the two columns specify the network inputs and expected responses, respectively.</td>
<td>ds argument of trainNetwork</td>
</tr>
<tr>
<td>Table</td>
<td>Table, where the first column contains either image paths or images, and the subsequent columns contain the responses.</td>
<td>tbl argument of trainNetwork</td>
</tr>
<tr>
<td>Cell array {X,Y}</td>
<td>X Numeric array of images.</td>
<td>X argument of trainNetwork</td>
</tr>
</tbody>
</table>
Sequence and Time Series Data

For sequence and time series data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array {sequences,Y}</td>
<td>Cell array of sequences or time series data.</td>
<td>sequences argument of trainNetwork</td>
</tr>
<tr>
<td>Y</td>
<td>Categorical vector of labels, cell array of categorical sequences, matrix of numeric responses, or cell array of numeric sequences.</td>
<td>Y argument of trainNetwork</td>
</tr>
<tr>
<td>Table</td>
<td>Table containing absolute or relative file paths to a MAT files containing sequence or time series data.</td>
<td>tbl argument of trainNetwork</td>
</tr>
</tbody>
</table>

During training, trainNetwork calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the 'ValidationFrequency' name-value pair argument. You can also use the validation data to stop training automatically when the validation loss stops decreasing. To turn on automatic validation stopping, use the 'ValidationPatience' name-value pair argument.

If your network has layers that behave differently during prediction than during training (for example, dropout layers), then the validation accuracy can be higher than the training (mini-batch) accuracy.
The validation data is shuffled according to the 'Shuffle' value. If the 'Shuffle' value equals 'every-epoch', then the validation data is shuffled before each network validation.

**ValidationFrequency — Frequency of network validation**

50 (default) | positive integer

Frequency of network validation in number of iterations, specified as the comma-separated pair consisting of 'ValidationFrequency' and a positive integer.

The 'ValidationFrequency' value is the number of iterations between evaluations of validation metrics. To specify validation data, use the 'ValidationData' name-value pair argument.

Example: 'ValidationFrequency', 20

**ValidationPatience — Patience of validation stopping**

Inf (default) | positive integer

Patience of validation stopping of network training, specified as the comma-separated pair consisting of 'ValidationPatience' and a positive integer or Inf.

The 'ValidationPatience' value is the number of times that the loss on the validation set can be larger than or equal to the previously smallest loss before network training stops. To turn on automatic validation stopping, specify a positive integer as the 'ValidationPatience' value. If you use the default value of Inf, then the training stops after the maximum number of epochs. To specify validation data, use the 'ValidationData' name-value pair argument.

Example: 'ValidationPatience', 5

**Solver Options**

**InitialLearnRate — Initial learning rate**

positive scalar

Initial learning rate used for training, specified as the comma-separated pair consisting of 'InitialLearnRate' and a positive scalar. The default value is 0.01 for the 'sgdm' solver and 0.001 for the 'rmsprop' and 'adam' solvers. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training might reach a suboptimal result or diverge.

Example: 'InitialLearnRate', 0.03
Data Types: single | double

**LearnRateSchedule — Option for dropping learning rate during training**

'none' (default) | 'piecewise'

Option for dropping the learning rate during training, specified as the comma-separated pair consisting of 'LearnRateSchedule' and one of the following:

- 'none' — The learning rate remains constant throughout training.
- 'piecewise' — The software updates the learning rate every certain number of epochs by multiplying with a certain factor. Use the LearnRateDropFactor name-value pair argument to specify the value of this factor. Use the LearnRateDropPeriod name-value pair argument to specify the number of epochs between multiplications.

Example: 'LearnRateSchedule','piecewise'

**LearnRateDropPeriod — Number of epochs for dropping the learning rate**

10 (default) | positive integer

Number of epochs for dropping the learning rate, specified as the comma-separated pair consisting of 'LearnRateDropPeriod' and a positive integer. This option is valid only when the value of LearnRateSchedule is 'piecewise'.

The software multiplies the global learning rate with the drop factor every time the specified number of epochs passes. Specify the drop factor using the LearnRateDropFactor name-value pair argument.

Example: 'LearnRateDropPeriod',3

**LearnRateDropFactor — Factor for dropping the learning rate**

0.1 (default) | scalar from 0 to 1

Factor for dropping the learning rate, specified as the comma-separated pair consisting of 'LearnRateDropFactor' and a scalar from 0 to 1. This option is valid only when the value of LearnRateSchedule is 'piecewise'.

LearnRateDropFactor is a multiplicative factor to apply to the learning rate every time a certain number of epochs passes. Specify the number of epochs using the LearnRateDropPeriod name-value pair argument.

Example: 'LearnRateDropFactor',0.1
Data Types: single | double

**L2Regularization — Factor for L\textsubscript{2} regularization**
0.0001 (default) | nonnegative scalar

Factor for L\textsubscript{2} regularization (weight decay), specified as the comma-separated pair consisting of 'L2Regularization' and a nonnegative scalar. For more information, see “L2 Regularization” on page 1-694.

You can specify a multiplier for the L\textsubscript{2} regularization for network layers with learnable parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

Example: 'L2Regularization',0.0005

Data Types: single | double

**Momentum — Contribution of previous step**
0.9 (default) | scalar from 0 to 1

Contribution of the parameter update step of the previous iteration to the current iteration of stochastic gradient descent with momentum, specified as the comma-separated pair consisting of 'Momentum' and a scalar from 0 to 1. A value of 0 means no contribution from the previous step, whereas a value of 1 means maximal contribution from the previous step.

To specify the 'Momentum' value, you must set solverName to be 'sgdm'. The default value works well for most problems. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.

Example: 'Momentum',0.95

Data Types: single | double

**GradientDecayFactor — Decay rate of gradient moving average**
0.9 (default) | scalar from 0 to 1

Decay rate of gradient moving average for the Adam solver, specified as the comma-separated pair consisting of 'GradientDecayFactor' and a scalar from 0 to 1. The gradient decay rate is denoted by $\beta_1$ in [4].

To specify the 'GradientDecayFactor' value, you must set solverName to be 'adam'. The default value works well for most problems. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.
Example: 'GradientDecayFactor', 0.95
Data Types: single | double

**SquaredGradientDecayFactor** — Decay rate of squared gradient moving average
nonnegative scalar less than 1

Decay rate of squared gradient moving average for the Adam and RMSProp solvers, specified as the comma-separated pair consisting of 'SquaredGradientDecayFactor' and a nonnegative scalar less than 1. The squared gradient decay rate is denoted by $\beta_2$ in [4].

To specify the 'SquaredGradientDecayFactor' value, you must set solverName to be 'adam' or 'rmsprop'. Typical values of the decay rate are 0.9, 0.99, and 0.999, corresponding to averaging lengths of 10, 100, and 1000 parameter updates, respectively. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.

If solverName is 'adam', then the default value of 'SquaredGradientDecayFactor' is 0.999. If solverName is 'rmsprop', then the default value of 'SquaredGradientDecayFactor' is 0.9.

Example: 'SquaredGradientDecayFactor', 0.99
Data Types: single | double

**Epsilon** — Denominator offset
$10^{-8}$ (default) | positive scalar

Denominator offset for Adam and RMSProp solvers, specified as the comma-separated pair consisting of 'Epsilon' and a positive scalar. The solver adds the offset to the denominator in the network parameter updates to avoid division by zero.

To specify the 'Epsilon' value, you must set solverName to be 'adam' or 'rmsprop'. The default value works well for most problems. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.

Example: 'Epsilon', 1e-6
Data Types: single | double

**ResetInputNormalization** — Option to reset input layer normalization
true (default) | false
Option to reset input layer normalization, specified as one of the following:

- **true** - Reset the input layer normalization statistics and recalculate them at training time.
- **false** - Calculate normalization statistics at training time when they are empty.

**Gradient Clipping**

**GradientThreshold** — Gradient threshold

Inf (default) | positive scalar

Gradient threshold, specified as the comma-separated pair consisting of 'GradientThreshold' and Inf or a positive scalar. If the gradient exceeds the value of GradientThreshold, then the gradient is clipped according to GradientThresholdMethod.

Example: 'GradientThreshold',6

**GradientThresholdMethod** — Gradient threshold method

'l2norm' (default) | 'global-l2norm' | 'absolute-value'

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as the comma-separated pair consisting of 'GradientThresholdMethod' and one of the following:

- **'l2norm'** — If the L₂ norm of the gradient of a learnable parameter is larger than GradientThreshold, then scale the gradient so that the L₂ norm equals GradientThreshold.
- **'global-l2norm'** — If the global L₂ norm, L, is larger than GradientThreshold, then scale all gradients by a factor of GradientThreshold/L. The global L₂ norm considers all learnable parameters.
- **'absolute-value'** — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than GradientThreshold, then scale the partial derivative to have magnitude equal to GradientThreshold and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-694.

Example: 'GradientThresholdMethod','global-l2norm'
Sequence Options

**SequenceLength — Option to pad, truncate, or split input sequences**

'longest' (default) | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: 'SequenceLength','shortest'

**SequencePaddingDirection — Direction of padding or truncation**

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer **OutputMode** property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the **OutputMode** property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the
predictions for the earlier time steps. To pad or truncate sequence data on the right, set the 'SequencePaddingDirection' option to 'right'.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingValue** — Value to pad input sequences

0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when SequenceLength is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

Example: 'SequencePaddingValue',-1

**Hardware Options**

**ExecutionEnvironment** — Hardware resource for training network

'auto' (default) | 'cpu' | 'gpu' | 'multi-gpu' | 'parallel'

Hardware resource for training network, specified as one of the following:

- 'auto' — Use a GPU if one is available. Otherwise, use the CPU.
- 'cpu' — Use the CPU.
- 'gpu' — Use the GPU.
- 'multi-gpu' — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- 'parallel' — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel and in the Cloud”.

GPU, multi-GPU, and parallel options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
To see an improvement in performance when training in parallel, try scaling up the MiniBatchSize and InitialLearnRate training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Datastores used for multi-GPU training or parallel training must be partitionable. For more information, see “Use Datastore for Parallel Training and Background Dispatching”.

If you use the 'multi-gpu' option with a partitionable input datastore and the 'DispatchInBackground' option, then the software starts a parallel pool with size equal to the default pool size. Workers with unique GPUs perform training computation. The remaining workers are used for background dispatch.

Example: 'ExecutionEnvironment','cpu'

WorkerLoad — Parallel worker load division
scalar from 0 to 1 | positive integer | numeric vector

Parallel worker load division between GPUs or CPUs, specified as the comma-separated pair consisting of 'WorkerLoad' and one of the following:

- Scalar from 0 to 1 — Fraction of workers on each machine to use for network training computation. If you train the network using data in a mini-batch datastore with background dispatch enabled, then the remaining workers fetch and preprocess data in the background.
- Positive integer — Number of workers on each machine to use for network training computation. If you train the network using data in a mini-batch datastore with background dispatch enabled, then the remaining workers fetch and preprocess data in the background.
- Numeric vector — Network training load for each worker in the parallel pool. For a vector W, worker i gets a fraction W(i)/\text{sum}(W) of the work (number of examples per mini-batch). If you train a network using data in a mini-batch datastore with background dispatch enabled, then you can assign a worker load of 0 to use that worker for fetching data in the background. The specified vector must contain one value per worker in the parallel pool.

If the parallel pool has access to GPUs, then workers without a unique GPU are never used for training computation. The default for pools with GPUs is to use all workers with a unique GPU for training computation, and the remaining workers for background dispatch. If the pool does not have access to GPUs and CPUs are used for training, then the default is to use one worker per machine for background data dispatch.
**DispatchInBackground — Use background dispatch**

false (default) | true

Use background dispatch (asynchronous prefetch queuing) to read training data from datastores, specified as false or true. Background dispatch requires Parallel Computing Toolbox.

DispatchInBackground is only supported for datastores that are partitionable. For more information, see “Use Datastore for Parallel Training and Background Dispatching”.

**Checkpoints**

**CheckpointPath — Path for saving checkpoint networks**

'' (default) | character vector

Path for saving the checkpoint networks, specified as the comma-separated pair consisting of 'CheckpointPath' and a character vector.

- If you do not specify a path (that is, you use the default ''), then the software does not save any checkpoint networks.
- If you specify a path, then trainNetwork saves checkpoint networks to this path after every epoch and assigns a unique name to each network. You can then load any checkpoint network and resume training from that network.

If the folder does not exist, then you must first create it before specifying the path for saving the checkpoint networks. If the path you specify does not exist, then trainingOptions returns an error.

For more information about saving network checkpoints, see “Save Checkpoint Networks and Resume Training”.

Example: 'CheckpointPath','C:\Temp\checkpoint'

Data Types: char

**OutputFcn — Output functions**

function handle | cell array of function handles

Output functions to call during training, specified as the comma-separated pair consisting of 'OutputFcn' and a function handle or cell array of function handles. trainNetwork calls the specified functions once before the start of training, after each iteration, and once after training has finished. trainNetwork passes a structure containing information in the following fields:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch</td>
<td>Current epoch number</td>
</tr>
<tr>
<td>Iteration</td>
<td>Current iteration number</td>
</tr>
<tr>
<td>TimeSinceStart</td>
<td>Time in seconds since the start of training</td>
</tr>
<tr>
<td>TrainingLoss</td>
<td>Current mini-batch loss</td>
</tr>
<tr>
<td>ValidationLoss</td>
<td>Loss on the validation data</td>
</tr>
<tr>
<td>BaseLearnRate</td>
<td>Current base learning rate</td>
</tr>
<tr>
<td>TrainingAccuracy</td>
<td>Accuracy on the current mini-batch (classification networks)</td>
</tr>
<tr>
<td>TrainingRMSE</td>
<td>RMSE on the current mini-batch (regression networks)</td>
</tr>
<tr>
<td>ValidationAccuracy</td>
<td>Accuracy on the validation data (classification networks)</td>
</tr>
<tr>
<td>ValidationRMSE</td>
<td>RMSE on the validation data (regression networks)</td>
</tr>
<tr>
<td>State</td>
<td>Current training state, with a possible value of &quot;start&quot;, &quot;iteration&quot;, or &quot;done&quot;</td>
</tr>
</tbody>
</table>

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

You can use output functions to display or plot progress information, or to stop training. To stop training early, make your output function return `true`. If any output function returns `true`, then training finishes and `trainNetwork` returns the latest network. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training”.

Data Types: `function_handle` | `cell`

### Output Arguments

**options — Training options**

`TrainingOptionsSGDM | TrainingOptionsRMSProp | TrainingOptionsADAM`
Training options, returned as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object. To train a neural network, use the training options as an input argument to the `trainNetwork` function.

If `solverName` equals 'sgdm', 'rmsprop', or 'adam', then the training options are returned as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object, respectively.

**Tips**

- For most deep learning tasks, you can use a pretrained network and adapt it to your own data. For an example showing how to use transfer learning to retrain a convolutional neural network to classify a new set of images, see “Train Deep Learning Network to Classify New Images”. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

  If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Custom Training Loops”.

**Algorithms**

**Initial Weights and Biases**

The default for the initial weights is a Gaussian distribution with a mean of 0 and a standard deviation of 0.01. The default for the initial bias value is 0. You can manually change the initialization for the weights and biases. See “Specify Initial Weights and Biases in Convolutional Layer” on page 1-875 and “Specify Initial Weights and Biases in Fully Connected Layer” on page 1-976.

**Stochastic Gradient Descent**

The standard gradient descent algorithm updates the network parameters (weights and biases) to minimize the loss function by taking small steps at each iteration in the direction of the negative gradient of the loss,

\[ \theta_{t+1} = \theta_t - \alpha \nabla E(\theta_t), \]
where \( \ell \) is the iteration number, \( \alpha > 0 \) is the learning rate, \( \theta \) is the parameter vector, and \( E(\theta) \) is the loss function. In the standard gradient descent algorithm, the gradient of the loss function, \( \nabla E(\theta) \), is evaluated using the entire training set, and the standard gradient descent algorithm uses the entire data set at once.

By contrast, at each iteration the \textit{stochastic} gradient descent algorithm evaluates the gradient and updates the parameters using a subset of the training data. A different subset, called a mini-batch, is used at each iteration. The full pass of the training algorithm over the entire training set using mini-batches is one \textit{epoch}. Stochastic gradient descent is stochastic because the parameter updates computed using a mini-batch is a noisy estimate of the parameter update that would result from using the full data set. You can specify the mini-batch size and the maximum number of epochs by using the 'MiniBatchSize' and 'MaxEpochs' name-value pair arguments, respectively.

**Stochastic Gradient Descent with Momentum**

The stochastic gradient descent algorithm can oscillate along the path of steepest descent towards the optimum. Adding a momentum term to the parameter update is one way to reduce this oscillation \cite{2}. The stochastic gradient descent with momentum (SGDM) update is

\[
\theta_{\ell + 1} = \theta_{\ell} - \alpha \nabla E(\theta_{\ell}) + \gamma (\theta_{\ell} - \theta_{\ell - 1}),
\]

where \( \gamma \) determines the contribution of the previous gradient step to the current iteration. You can specify this value using the 'Momentum' name-value pair argument. To train a neural network using the stochastic gradient descent with momentum algorithm, specify solverName as 'sgdm'. To specify the initial value of the learning rate \( \alpha \), use the 'InitialLearnRate' name-value pair argument. You can also specify different learning rates for different layers and parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

**RMSProp**

Stochastic gradient descent with momentum uses a single learning rate for all the parameters. Other optimization algorithms seek to improve network training by using learning rates that differ by parameter and can automatically adapt to the loss function being optimized. RMSProp (root mean square propagation) is one such algorithm. It keeps a moving average of the element-wise squares of the parameter gradients,

\[
v_{\ell} = \beta_2 v_{\ell - 1} + (1 - \beta_2) [\nabla E(\theta_{\ell})]^2
\]
\( \beta_2 \) is the decay rate of the moving average. Common values of the decay rate are 0.9, 0.99, and 0.999. The corresponding averaging lengths of the squared gradients equal \( 1/(1-\beta_2) \), that is, 10, 100, and 1000 parameter updates, respectively. You can specify \( \beta_2 \) by using the 'SquaredGradientDecayFactor' name-value pair argument. The RMSProp algorithm uses this moving average to normalize the updates of each parameter individually,

\[
\theta_{t+1} = \theta_t - \frac{\alpha \nabla E(\theta_t)}{\sqrt{v_t} + \epsilon}
\]

where the division is performed element-wise. Using RMSProp effectively decreases the learning rates of parameters with large gradients and increases the learning rates of parameters with small gradients. \( \epsilon \) is a small constant added to avoid division by zero. You can specify \( \epsilon \) by using the 'Epsilon' name-value pair argument, but the default value usually works well. To use RMSProp to train a neural network, specify solverName as 'rmsprop'.

**Adam**

Adam (derived from *adaptive moment estimation*) [4] uses a parameter update that is similar to RMSProp, but with an added momentum term. It keeps an element-wise moving average of both the parameter gradients and their squared values,

\[
m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla E(\theta_t)
\]

\[
v_t = \beta_2 v_{t-1} + (1 - \beta_2)[\nabla E(\theta_t)]^2
\]

You can specify the \( \beta_1 \) and \( \beta_2 \) decay rates using the 'GradientDecayFactor' and 'SquaredGradientDecayFactor' name-value pair arguments, respectively. Adam uses the moving averages to update the network parameters as

\[
\theta_{t+1} = \theta_t - \frac{\alpha m_t}{\sqrt{v_t} + \epsilon}
\]

If gradients over many iterations are similar, then using a moving average of the gradient enables the parameter updates to pick up momentum in a certain direction. If the gradients contain mostly noise, then the moving average of the gradient becomes smaller, and so the parameter updates become smaller too. You can specify \( \epsilon \) by using the 'Epsilon' name-value pair argument. The default value usually works well, but for certain problems a value as large as 1 works better. To use Adam to train a neural network, specify solverName as 'adam'. The full Adam update also includes a
mechanism to correct a bias the appears in the beginning of training. For more information, see [4].

Specify the learning rate $\alpha$ for all optimization algorithms using the 'InitialLearnRate' name-value pair argument. The effect of the learning rate is different for the different optimization algorithms, so the optimal learning rates are also different in general. You can also specify learning rates that differ by layers and by parameter. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

**Gradient Clipping**

If the gradients increase in magnitude exponentially, then the training is unstable and can diverge within a few iterations. This "gradient explosion" is indicated by a training loss that goes to NaN or Inf. Gradient clipping helps prevent gradient explosion by stabilizing the training at higher learning rates and in the presence of outliers [3]. Gradient clipping enables networks to be trained faster, and does not usually impact the accuracy of the learned task.

There are two types of gradient clipping.

- Norm-based gradient clipping rescales the gradient based on a threshold, and does not change the direction of the gradient. The 'l2norm' and 'global-l2norm' values of GradientThresholdMethod are norm-based gradient clipping methods.

- Value-based gradient clipping clips any partial derivative greater than the threshold, which can result in the gradient arbitrarily changing direction. Value-based gradient clipping can have unpredictable behavior, but sufficiently small changes do not cause the network to diverge. The 'absolute-value' value of GradientThresholdMethod is a value-based gradient clipping method.

For examples, see “Time Series Forecasting Using Deep Learning” and “Sequence-to-Sequence Classification Using Deep Learning”.

**L₂ Regularization**

Adding a regularization term for the weights to the loss function $E(\theta)$ is one way to reduce overfitting [1], [2]. The regularization term is also called weight decay. The loss function with the regularization term takes the form

$$E_R(\theta) = E(\theta) + \lambda \Omega(w),$$
where \( w \) is the weight vector, \( \lambda \) is the regularization factor (coefficient), and the regularization function \( \Omega(w) \) is

\[
\Omega(w) = \frac{1}{2} w^T w.
\]

Note that the biases are not regularized [2]. You can specify the regularization factor \( \lambda \) by using the 'L2Regularization' name-value pair argument. You can also specify different regularization factors for different layers and parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers”.

The loss function that the software uses for network training includes the regularization term. However, the loss value displayed in the command window and training progress plot during training is the loss on the data only and does not include the regularization term.

**Compatibility Considerations**

'ValidationPatience' training option default is Inf

*Behavior changed in R2018b*

Starting in R2018b, the default value of the 'ValidationPatience' training option is Inf, which means that automatic stopping via validation is turned off. This behavior prevents the training from stopping before sufficiently learning from the data.

In previous versions, the default value is 5. To reproduce this behavior, set the 'ValidationPatience' option to 5.

Different file name for checkpoint networks

*Behavior changed in R2018b*

Starting in R2018b, when saving checkpoint networks, the software assigns file names beginning with `net_checkpoint_`. In previous versions, the software assigns file names beginning with `convnet_checkpoint_`.

If you have code that saves and loads checkpoint networks, then update your code to load files with the new name.
References


See Also

analyzeNetwork | trainNetwork

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2016a
trainNetwork

Train neural network for deep learning

Syntax

net = trainNetwork(imds,layers,options)
net = trainNetwork(ds,layers,options)
net = trainNetwork(X,Y,layers,options)
net = trainNetwork(sequences,Y,layers,options)
net = trainNetwork(tbl,layers,options)
net = trainNetwork(tbl,responseName,layers,options)

[net,info] = trainNetwork( ___ )

Description

Use trainNetwork to train a convolutional neural network (ConvNet, CNN), a long short-term memory (LSTM) network, or a bidirectional LSTM (BiLSTM) network for deep learning classification and regression problems. You can train a network on either a CPU or a GPU. For image classification and image regression, you can train using multiple GPUs or in parallel. Using GPU, multi-GPU, and parallel options requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. Specify training options, including options for the execution environment, by using trainingOptions.

net = trainNetwork(imds,layers,options) trains a network for image classification problems. The image datastore imds stores the input image data, layers defines the network architecture, and options defines the training options.

net = trainNetwork(ds,layers,options) trains a network using the datastore ds. For networks with multiple inputs, use this syntax with a combined or transformed datastore.
net = trainNetwork(X,Y,layers,options) trains a network for image classification and regression problems. The numeric array X contains the predictor variables and Y contains the categorical labels or numeric responses.

net = trainNetwork(sequences,Y,layers,options) trains an network for sequence classification and regression problems (for example, an LSTM or BiLSTM network), where sequences contains sequence or time series predictors and Y contains the responses. For classification problems, Y is a categorical vector or a cell array of categorical sequences. For regression problems, Y is a matrix of targets or a cell array of numeric sequences.

net = trainNetwork(tbl,layers,options) trains a network for classification and regression problems. The table tbl contains numeric data or file paths to the data. The predictors must be in the first column of tbl. For information on the targets or response variables, see “tbl” on page 1-0.

net = trainNetwork(tbl,responseName,layers,options) trains a network for classification and regression problems. The predictors must be in the first column of tbl. The responseName argument specifies the response variables in tbl.

[net,info] = trainNetwork(____) also returns information on the training using any of the input arguments in the previous syntaxes.

Examples

Train Network for Image Classification

Load the data as an ImageDatastore object.

digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath,
    'IncludeSubfolders',true,
    'LabelSource','foldernames');

The datastore contains 10,000 synthetic images of digits from 0 to 9. The images are generated by applying random transformations to digit images created with different fonts. Each digit image is 28-by-28 pixels. The datastore contains an equal number of images per category.
Display some of the images in the datastore.

```matlab
figure
umImages = 10000;
perm = randperm(numImages,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end
```

Divide the datastore so that each category in the training set has 750 images and the testing set has the remaining images from each label.
numTrainingFiles = 750;
[imdsTrain,imdsTest] = splitEachLabel(imds,numTrainingFiles,'randomize');

splitEachLabel splits the image files in digitData into two new datastores, imdsTrain and imdsTest.

Define the convolutional neural network architecture.

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

Set the options to the default settings for the stochastic gradient descent with momentum. Set the maximum number of epochs at 20, and start the training with an initial learning rate of 0.0001.

options = trainingOptions('sgdm', ...
    'MaxEpochs',20,...
    'InitialLearnRate',1e-4,...
    'Verbose',false,...
    'Plots','training-progress');

Train the network.

net = trainNetwork(imdsTrain,layers,options);
Run the trained network on the test set, which was not used to train the network, and predict the image labels (digits).

\[
\text{YPred} = \text{classify}(\text{net}, \text{imdsTest}); \\
\text{YTest} = \text{imdsTest}.\text{Labels};
\]

Calculate the accuracy. The accuracy is the ratio of the number of true labels in the test data matching the classifications from \text{classify} to the number of images in the test data.
accuracy = sum(YPred == YTest)/numel(YTest)
accuracy = 0.9412

Train Network with Augmented Images

Train a convolutional neural network using augmented image data. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

Load the sample data, which consists of synthetic images of handwritten digits.

[XTrain,YTrain] = digitTrain4DArrayData;
digitTrain4DArrayData loads the digit training set as 4-D array data. XTrain is a 28-by-28-by-1-by-5000 array, where:
• 28 is the height and width of the images.
• 1 is the number of channels.
• 5000 is the number of synthetic images of handwritten digits.

YTrain is a categorical vector containing the labels for each observation.

Set aside 1000 of the images for network validation.

idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,,:,idx);
XTrain(:,:,,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];

Create an imageDataAugmenter object that specifies preprocessing options for image augmentation, such as resizing, rotation, translation, and reflection. Randomly translate the images up to three pixels horizontally and vertically, and rotate the images with an angle up to 20 degrees.

imageAugmenter = imageDataAugmenter( ...  
  'RandRotation',[-20,20], ...  
  'RandXTranslation',[-3 3], ...  
  'RandYTranslation',[-3 3])
Create an `augmentedImageDatastore` object to use for network training and specify the image output size. During training, the datastore performs image augmentation and resizes the images. The datastore augments the images without saving any images to memory. The `trainNetwork` updates the network parameters and then discards the augmented images.

```matlab
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter);
```

Specify the convolutional neural network architecture.

```matlab
layers = [
    imageInputLayer(imageSize)
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
];
```
reluLayer

fullyConnectedLayer(10)
softmaxLayer
classificationLayer;

Specify training options for stochastic gradient descent with momentum.

opts = trainingOptions('sgdm', ...
    'MaxEpochs',15, ...    'Shuffle','every-epoch', ...    'Plots','training-progress', ...    'Verbose',false, ...    'ValidationData',{XValidation,YValidation});

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

net = trainNetwork(augimds,layers,opts);
Train Network for Image Regression

Load the sample data, which consists of synthetic images of handwritten digits. The third output contains the corresponding angles in degrees by which each image has been rotated.
Load the training images as 4-D arrays using `digitTrain4DArrayData`. The output `XTrain` is a 28-by-28-by-1-by-5000 array, where:

- 28 is the height and width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` contains the rotation angles in degrees.

```matlab
[XTrain,~,YTrain] = digitTrain4DArrayData;
```

Display 20 random training images using `imshow`.

```matlab
figure
numTrainImages = numel(YTrain);
idx = randperm(numTrainImages,20);
for i = 1:numel(idx)
    subplot(4,5,i)
    imshow(XTrain(:,:,:,idx(i)))
end
```
Specify the convolutional neural network architecture. For regression problems, include a regression layer at the end of the network.

```matlab
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer];
```

Specify the network training options. Set the initial learn rate to 0.001.

```matlab
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.001, ...
Train the network.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```

Test the performance of the network by evaluating the prediction accuracy of the test data. Use `predict` to predict the angles of rotation of the validation images.
[XTest,~,YTest] = digitTest4DArrayData;
YPred = predict(net,XTest);

Evaluate the performance of the model by calculating the root-mean-square error (RMSE) of the predicted and actual angles of rotation.

\[ \text{rmse} = \sqrt{\text{mean}((\text{YTest} - \text{YPred})^2)} \]

\[ \text{rmse} = \text{single}6.0354 \]

**Train Network for Sequence Classification**

Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. \texttt{XTrain} is a cell array containing 270 sequences of varying length with a feature dimension of 12. \texttt{Y} is a categorical vector of labels 1,2,...,9. The entries in \texttt{XTrain} are matrices with 12 rows (one row for each feature) and a varying number of columns (one column for each time step).

\[ \texttt{[XTrain,YTrain]} = \text{japaneseVowelsTrainData}; \]

Visualize the first time series in a plot. Each line corresponds to a feature.

\[ \text{figure} \]
\[ \text{plot}(\text{XTrain}{1}') \]
\[ \text{title}(\text{"Training Observation 1"}) \]
\[ \text{numFeatures} = \text{size}(\text{XTrain}{1},1); \]
\[ \text{legend}([\text{"Feature "} + \text{string}(1:\text{numFeatures}),'\text{Location}','\text{northeastoutside}') \]
Define the LSTM network architecture. Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer to have 100 hidden units and to output the last element of the sequence. Finally, specify nine classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```matlab
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ... 
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
];
```
softmaxLayer
classificationLayer]
layers =
5x1 Layer array with layers:
  1   ''   Sequence Input          Sequence input with 12 dimensions
  2   ''   LSTM                    LSTM with 100 hidden units
  3   ''   Fully Connected         9 fully connected layer
  4   ''   Softmax                 softmax
  5   ''   Classification Output   crossentropyex

Specify the training options. Specify the solver as 'adam' and 'GradientThreshold' as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 100.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set 'ExecutionEnvironment' to 'cpu'. To train on a GPU, if available, set 'ExecutionEnvironment' to 'auto' (the default value).

maxEpochs = 100;
miniBatchSize = 27;

options = trainingOptions('adam', ...
  'ExecutionEnvironment','cpu', ...
  'MaxEpochs',maxEpochs, ...
  'MiniBatchSize',miniBatchSize, ...
  'GradientThreshold',1, ...
  'Verbose',false, ...
  'Plots','training-progress');

Train the LSTM network with the specified training options.

net = trainNetwork(XTrain,YTrain,layers,options);
Load the test set and classify the sequences into speakers.

\[
[\text{XTest}, \text{YTest}] = \text{japaneseVowelsTestData};
\]

Classify the test data. Specify the same mini-batch size used for training.

\[
\text{YPred} = \text{classify(} \text{net, XTest, 'MiniBatchSize', miniBatchSize});
\]

Calculate the classification accuracy of the predictions.
acc = sum(YPred == YTest)./numel(YTest)
acc = 0.9486

**Input Arguments**

**imds — Image datastore**

*ImageDatastore object*

Image datastore, specified as an *ImageDatastore* object.

*ImageDatastore* allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then *ImageDatastore* does not prefetch.

**Tip** Use *augmentedImageDatastore* for efficient preprocessing of images for deep learning including image resizing.

Do not use the *readFcn* option of *ImageDatastore* as this option is usually significantly slower.

**ds — Datastore**

*Datastore*

Datastore for out-of-memory data and preprocessing.

For networks with a single input, the table or cell array returned by the datastore has two columns that specify the network inputs and expected responses, respectively.

For networks with multiple inputs, the datastore must be a combined or transformed datastore that returns a cell array with (*numInputs*+1) columns containing the predictors and the responses, where *numInputs* is the number of network inputs and *numResponses* is the number of responses. For *i* less than or equal to *numInputs*, the *i*th element of the cell array corresponds to the input layers.*InputNames*(i), where *layers* is the layer graph defining the network architecture. The last column of the cell array corresponds to the responses.

The table below lists the datastores that are directly compatible with *trainNetwork*. You can use other built-in datastores for training deep learning networks by using the
transform and combine functions. These functions can convert the data read from datastores to the table or cell array format required by trainNetwork. For more information, see “Datastores for Deep Learning”.

<table>
<thead>
<tr>
<th>Type of datastore</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CombinedDatastore</td>
<td>Horizontally concatenate the data read from two or more underlying datastores.</td>
</tr>
<tr>
<td>TransformedDatastore</td>
<td>Transform batches of read data from an underlying datastore according to your own preprocessing pipeline.</td>
</tr>
<tr>
<td>AugmentedImageDatastore</td>
<td>Apply random affine geometric transformations, including resizing, rotation, reflection, shear, and translation, for training deep neural networks.</td>
</tr>
<tr>
<td>PixelLabelImageDatastore</td>
<td>Apply identical affine geometric transformations to images and corresponding ground truth labels for training semantic segmentation networks (requires Computer Vision Toolbox™).</td>
</tr>
<tr>
<td>RandomPatchExtractionDatastore</td>
<td>Extract pairs of random patches from images or pixel label images (requires Image Processing Toolbox™). You optionally can apply identical random affine geometric transformations to the pairs of patches.</td>
</tr>
<tr>
<td>DenoisingImageDatastore</td>
<td>Apply randomly generated Gaussian noise for training denoising networks (requires Image Processing Toolbox).</td>
</tr>
<tr>
<td>Custom mini-batch datastore</td>
<td>Create mini-batches of sequence, time series, or text data. For details, see “Develop Custom Mini-Batch Datastore”.</td>
</tr>
</tbody>
</table>

**X — Image data**
numeric array

Image data, specified as a numeric array. The size of the array depends on the type of image input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D images</td>
<td>A h-by-w-by-c-by-N numeric array, where h, w, and c are the height, width, and number of channels of the images, respectively, and N is the number of images.</td>
</tr>
</tbody>
</table>
### Input Description

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-D images</td>
<td>A $h$-by-$w$-by-$d$-by-$c$-by-$N$ numeric array, where $h$, $w$, $d$, and $c$ are the height, width, depth, and number of channels of the images, respectively, and $N$ is the number of images.</td>
</tr>
</tbody>
</table>

If the array contains NaNs, then they are propagated through the network.

**sequences — Sequence or time series data**

cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an $N$-by-1 cell array of numeric arrays, where $N$ is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector sequences</td>
<td>$c$-by-$s$ matrices, where $c$ is the number of features of the sequences and $s$ is the sequence length.</td>
</tr>
<tr>
<td>2-D image sequences</td>
<td>$h$-by-$w$-by-$c$-by-$s$ arrays, where $h$, $w$, and $c$ correspond to the height, width, and number of channels of the images, respectively, and $s$ is the sequence length.</td>
</tr>
<tr>
<td>3-D image sequences</td>
<td>$h$-by-$w$-by-$d$-by-$c$-by-$s$, where $h$, $w$, $d$, and $c$ correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and $s$ is the sequence length.</td>
</tr>
</tbody>
</table>

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

**Y — Responses**
categorical vector of labels | numeric array | cell array of categorical sequences | cell array of numeric sequences
Responses, specified as a categorical vector of labels, a numeric array, a cell array of categorical sequences, or cell array of numeric sequences. The format of \( Y \) depends on the type of task. Responses must not contain NaNs.

**Classification**

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>( N )-by-1 categorical vector of labels, where ( N ) is the number of observations.</td>
</tr>
<tr>
<td>Sequence-to-label classification</td>
<td></td>
</tr>
<tr>
<td>Sequence-to-sequence classification</td>
<td>( \text{cell array of categorical sequences of labels, where } N \text{ is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after applying the } \text{SequenceLength} \text{ option to each mini-batch independently.} )</td>
</tr>
</tbody>
</table>

For sequence-to-sequence classification problems with one observation, sequences can also be a vector. In this case, \( Y \) must be a categorical sequence of labels.

**Regression**

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
</table>
| 2-D image regression        | • \( N \)-by-\( R \) matrix, where \( N \) is the number of images and \( R \) is the number of responses.  
• \( h \)-by-\( w \)-by-\( c \)-by-\( N \) numeric array, where \( h \), \( w \), and \( c \) are the height, width, and number of channels of the images, respectively, and \( N \) is the number of images. |
Task | Format
--- | ---
3-D image regression | • $N$-by-$R$ matrix, where $N$ is the number of images and $R$ is the number of responses.
• $h$-by-$w$-by-$d$-by-$c$-by-$N$ numeric array, where $h$, $w$, $d$, and $c$ are the height, width, depth, and number of channels of the images, respectively, and $N$ is the number of images.

Sequence-to-one regression | $N$-by-$R$ matrix, where $N$ is the number of sequences and $R$ is the number of responses.

Sequence-to-sequence regression | $N$-by-1 cell array of numeric sequences, where $N$ is the number of sequences. The sequences are matrices with $R$ rows, where $R$ is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently.

For sequence-to-sequence regression problems with one observation, `sequences` can be a matrix. In this case, `Y` must be a matrix of responses.

Normalizing the responses often helps to stabilize and speed up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

**tbl — Input data**

*table*

Input data, specified as a table containing predictors in the first column and responses in the remaining column or columns. Each row in the table corresponds to an observation.

The arrangement of predictors and responses in the table columns depends on the type of problem.
## Classification

<table>
<thead>
<tr>
<th>Task</th>
<th>Predictors</th>
<th>Responses</th>
</tr>
</thead>
</table>
| Image classification     | • Absolute or relative file path to an image, specified as a character vector  
                           | • Image specified as a 3-D numeric array                                     | Categorical label      |
| Sequence-to-label        | Absolute or relative file path to a MAT file containing sequence or time series data. | Categorical label      |
| classification           |                                                                             |                         |
| Sequence-to-sequence     | Absolute or relative file path to a MAT file. The MAT file must contain a time series represented by a categorical vector, with entries corresponding to labels for each time step. |                         |
| classification           |                                                                             |                         |

For classification problems, if you do not specify `responseName`, then the function, by default, uses the responses in the second column of `tbl`. 
Regression

<table>
<thead>
<tr>
<th>Task</th>
<th>Predictors</th>
<th>Responses</th>
</tr>
</thead>
</table>
| Image regression         | • Absolute or relative file path to an image, specified as a character vector  
                          | • Image specified as a 3-D numeric array                                     | • One or more columns of scalar values                                      |
|                          |                                                                             | • Numeric vector                                                            |
|                          |                                                                             | • 1-by-1 cell array containing a 3-D numeric array                          |
| Sequence-to-one regression | Absolute or relative file path to a MAT file containing sequence or time series data. | • One or more columns of scalar values                                      |
|                          |                                                                             | • Numeric vector                                                            |
| Sequence-to-sequence regression | The MAT file must contain a time series represented by a matrix with rows corresponding to data points and columns corresponding to time steps. | Absolute or relative file path to a MAT file. The MAT file must contain a time series represented by a matrix, where rows correspond to responses and columns correspond to time steps. |

For regression problems, if you do not specify `responseName`, then the function, by default, uses the remaining columns of `tbl`. Normalizing the responses often helps to stabilize and speed up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

Responses cannot contain NaNs. If the predictor data contains NaNs, then they are propagated through the training. However, in most cases, the training fails to converge.

Data Types: `table`

`responseName — Names of response variables in the input table`
character vector | cell array of character vectors | string array

Names of the response variables in the input table, specified as a character vector, cell array of character vectors, or a string array. For problems with one response, `responseName` is the corresponding variable name in `tbl`. For regression problems with multiple response variables, `responseName` is an array of the corresponding variable names in `tbl`.

Data Types: `char` | `cell` | `string`
layers — Network layers
Layer array | LayerGraph object

Network layers, specified as a Layer array or a LayerGraph object.

To create a network with all layers connected sequentially, you can use a Layer array as the input argument. In this case, the returned network is a SeriesNetwork object.

A directed acyclic graph (DAG) network has a complex structure in which layers can have multiple inputs and outputs. To create a DAG network, specify the network architecture as a LayerGraph object and then use that layer graph as the input argument to trainNetwork.

For a list of built-in layers, see “List of Deep Learning Layers”.

options — Training options
TrainingOptionsSGDM | TrainingOptionsRMSProp | TrainingOptionsADAM

Training options, specified as a TrainingOptionsSGDM, TrainingOptionsRMSProp, or TrainingOptionsADAM object returned by the trainingOptions function. To specify solver and other options for network training, use trainingOptions.

Output Arguments

net — Trained network
SeriesNetwork object | DAGNetwork object

Trained network, returned as a SeriesNetwork object or a DAGNetwork object.

If you train the network using a Layer array as the layers input argument, then net is a SeriesNetwork object. If you train the network using a LayerGraph object as the input argument, then net is a DAGNetwork object.

info — Training information
structure

Training information, returned as a structure, where each field is a numeric vector with one element per training iteration.

For classification problems, info contains the following fields:
• **TrainingLoss** — Loss function values
• **TrainingAccuracy** — Training accuracies
• **ValidationLoss** — Loss function values
• **ValidationAccuracy** — Validation accuracies
• **BaseLearnRate** — Learning rates

For regression problems, info contains the following fields:

• **TrainingLoss** — Loss function values
• **TrainingRMSE** — Training RMSE values
• **ValidationLoss** — Loss function values
• **ValidationAccuracy** — Validation accuracies
• **ValidationRMSE** — Validation RMSE values
• **BaseLearnRate** — Learning rates

The structure only contains the fields **ValidationLoss**, **ValidationAccuracy**, and **ValidationRMSE** when options specifies validation data. The 'ValidationFrequency' option of trainingOptions determines which iterations the software calculates validation metrics. For iterations when the software does not calculate validation metrics, the corresponding values in the structure are NaN.

**More About**

**Save Checkpoint Networks and Resume Training**

Deep Learning Toolbox enables you to save networks as .mat files after each epoch during training. This periodic saving is especially useful when you have a large network or a large data set, and training takes a long time. If the training is interrupted for some reason, you can resume training from the last saved checkpoint network. If you want trainNetwork to save checkpoint networks, then you must specify the name of the path by using the 'CheckpointPath' name-value pair argument of trainingOptions. If the path that you specify does not exist, then trainingOptions returns an error.

trainNetwork automatically assigns unique names to checkpoint network files. In the example name, `net_checkpoint__351__2018_04_12__18_09_52.mat`, 351 is the iteration number, 2018_04_12 is the date, and 18_09_52 is the time at which trainNetwork saves the network. You can load a checkpoint network file by double-clicking it or using the load command at the command line. For example:
load net_checkpoint__351__2018_04_12__18_09_52.mat

You can then resume training by using the layers of the network as an input argument to \texttt{trainNetwork}. For example:

\texttt{trainNetwork(XTrain,YTrain,net.Layers,options)}

You must manually specify the training options and the input data, because the checkpoint network does not contain this information. For an example, see “Resume Training from Checkpoint Network”.

\section*{Floating-Point Arithmetic}

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include \texttt{trainNetwork}, \texttt{predict}, \texttt{classify}, and \texttt{activations}. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

\section*{References}


\section*{Extended Capabilities}

\section*{Automatic Parallel Support}

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run computation in parallel, set the 'ExecutionEnvironment' option to 'multi-gpu' or 'parallel'.

1-722
Use `trainingOptions` to set the 'ExecutionEnvironment' and supply the options to `trainNetwork`. If you do not set 'ExecutionEnvironment', then `trainNetwork` runs on a GPU if available.

For details, see “Scale Up Deep Learning in Parallel and in the Cloud”.

**See Also**

DAGNetwork | LayerGraph | SeriesNetwork | analyzeNetwork | assembleNetwork | classify | predict | trainingOptions

**Topics**

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Train Convolutional Neural Network for Regression”
“Sequence Classification Using Deep Learning”
“Deep Learning in MATLAB”
“Define Custom Deep Learning Layers”
“List of Deep Learning Layers”

**Introduced in R2016a**
**deepDreamImage**

Visualize network features using deep dream

**Syntax**

\[
I = \text{deepDreamImage}(\text{net}, \text{layer}, \text{channels}) \\
I = \text{deepDreamImage}(\text{net}, \text{layer}, \text{channels}, \text{Name}, \text{Value})
\]

**Description**

\[
I = \text{deepDreamImage}(\text{net}, \text{layer}, \text{channels})
\]
returns an array of images that strongly activate the channels \(\text{channels}\) within the network \(\text{net}\) of the layer with numeric index or name given by \(\text{layer}\). These images highlight the features learned by a network.

\[
I = \text{deepDreamImage}(\text{net}, \text{layer}, \text{channels}, \text{Name}, \text{Value})
\]
returns an image with additional options specified by one or more \(\text{Name}, \text{Value}\) pair arguments.

**Examples**

**Visualize Convolutional Neural Network Features**

Load a pretrained AlexNet network.

\[
\text{net} = \text{alexnet};
\]

Visualize the first 25 features learned by the first convolutional layer ("conv1") using \text{deepDreamImage}. Set 'PyramidLevels' to 1 so that the images are not scaled.

\[
\text{layer} = \text{'conv1'}; \\
\text{channels} = 1:25; \\
I = \text{deepDreamImage}(\text{net}, \text{layer}, \text{channels}, ...) \\
\quad \text{'PyramidLevels', 1, ...}
\]
deepDreamImage

```
'verbose', 0);

figure
for i = 1:25
    subplot(5, 5, i)
    imshow(I(:, :, :, i))
end

Input Arguments

net — Trained network
    SeriesNetwork object | DAGNetwork object
Trained network, specified as a `SeriesNetwork` object or a `DAGNetwork` object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function. For more information about pretrained networks, see “Pretrained Deep Neural Networks”.

deepDreamImage only supports networks with an image input layer.

**layer — Layer index or name**

positive integer | character vector | string scalar

Layer to visualize, specified as a positive integer, a character vector, or a string scalar. If `net` is a `DAGNetwork` object, specify `layer` as a character vector or string scalar only. Specify `layer` as the index or the name of the layer you want to visualize the activations of. To visualize classification layer features, select the last fully connected layer before the classification layer.

**Tip** Selecting ReLU or dropout layers for visualization may not produce useful images because of the effect that these layers have on the network gradients.

**channels — Channel index**

numeric index | vector of numeric indices

Queried channels, specified as scalar or vector of channel indices. If `channels` is a vector, the layer activations for each channel are optimized independently. The possible choices for `channels` depend on the selected layer. For convolutional layers, the `NumFilters` property specifies the number of output channels. For fully connected layers, the `OutputSize` property specifies the number of output channels.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`deepDreamImage(net,layer,channels,'NumItetations',100,'ExecutionEnvironment','gpu')` generates images using 100 iterations per pyramid level and uses the GPU.
**InitialImage — Image to initialize Deep Dream**

array

Image to initialize Deep Dream. Use this syntax to see how an image is modified to maximize network layer activations. The minimum height and width of the initial image depend on all the layers up to and including the selected layer:

- For layers towards the end of the network, the initial image must be at least the same height and width as the image input layer.
- For layers towards the beginning of the network, the height and width of the initial image can be smaller than the image input layer. However, it must be large enough to produce a scalar output at the selected layer.
- The number of channels of the initial image must match the number of channels in the image input layer of the network.

If you do not specify an initial image, the software uses a random image with pixels drawn from a standard normal distribution. See also 'PyramidLevels' on page 1-0.

**PyramidLevels — Number of pyramid levels**

3 (default) | positive integer

Number of multi-resolution image pyramid levels to use to generate the output image, specified as a positive integer. Increase the number of pyramid levels to produce larger output images at the expense of additional computation. To produce an image of the same size as the initial image, set the number of levels to 1.

Example: 'PyramidLevels',3

**PyramidScale — Scale between pyramid levels**

1.4 (default) | scalar with value > 1

Scale between each pyramid level, specified as a scalar with value > 1. Reduce the pyramid scale to incorporate fine grain details into the output image. Adjusting the pyramid scale can help generate more informative images for layers at the beginning of the network.

Example: 'PyramidScale',1.4

**NumIterations — Number of iterations per pyramid level**

10 (default) | positive integer
Number of iterations per pyramid level, specified as a positive integer. Increase the number of iterations to produce more detailed images at the expense of additional computation.

Example: 'NumIterations',10

**OutputScaling — Type of scaling to apply to output**

'linear' (default) | 'none'

Type of scaling to apply to output image, specified as the comma-separated pair consisting of 'OutputScaling' and one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'linear'</td>
<td>Scale output pixel values in the interval [0,1]. The output image corresponding to each layer channel, ( \mathbf{I}(:, :, :, \text{channel}) ), is scaled independently.</td>
</tr>
<tr>
<td>'none'</td>
<td>Disable output scaling.</td>
</tr>
</tbody>
</table>

Scaling the pixel values can cause the network can misclassify the output image. If you want to classify the output image, set the 'OutputScaling' value to 'none'.

Example: 'OutputScaling','linear'

**Verbose — Indicator to display progress information**

1 (default) | 0

Indicator to display progress information in the command window, specified as the comma-separated pair consisting of 'Verbose' and either 1 (true) or 0 (false). The displayed information includes the pyramid level, iteration, and the activation strength.

Example: 'Verbose',0

Data Types: logical

**ExecutionEnvironment — Hardware resource**

'auto' (default) | 'gpu' | 'cpu'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
• 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

• 'cpu' — Use the CPU.

Example: 'ExecutionEnvironment','cpu'

Output Arguments

I — Output image array

Output image, specified by a sequence of grayscale or truecolor (RGB) images stored in a 4-D array. Images are concatenated along the fourth dimension of I such that the image that maximizes the output of channels(k) is I(:,:,,:,k). You can display the output image using imshow.

Algorithms

This function implements a version of deep dream that uses a multi-resolution image pyramid and Laplacian Pyramid Gradient Normalization to generate high-resolution images. For more information on Laplacian Pyramid Gradient Normalization, see this blog post: DeepDreaming with TensorFlow.

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include trainNetwork, predict, classify, and activations. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

References

See Also
activations | alexnet | googlenet | squeezenet | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Deep Dream Images Using AlexNet”
“Visualize Features of a Convolutional Neural Network”
“Visualize Activations of a Convolutional Neural Network”
“Visualize Activations of LSTM Network”

Introduced in R2017a
occlusionSensitivity

Determine how input data affects output activations by occluding input

Syntax

scoreMap = occlusionSensitivity(net,X,label)
activationMap = occlusionSensitivity(net,X,layer,channel)
___ = occlusionSensitivity(____,Name,Value)

Description

scoreMap = occlusionSensitivity(net,X,label) computes a map of the change in classification score for the classes specified by label when parts of the input data X are occluded with a mask. The change in classification score is relative to the original data without occlusion. The occluding mask is moved across the input data, giving a change in classification score for each mask location. Use an occlusion map to identify the parts of your input data that most impact the classification score. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified classification label. The network must contain a softmaxLayer followed by a classificationLayer.

activationMap = occlusionSensitivity(net,X,layer,channel) computes a map of the change in total activation for the specified layer and channel when parts of the input data X are occluded with a mask. The change in activation score is relative to the original data without occlusion. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified channel activation, obtained by summing over all spatial dimensions for that channel.

___ = occlusionSensitivity(____,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, 'Stride',50 sets the stride of the occluding mask to 50 pixels.

Examples
Visualize Which Parts of an Image Influence Classification Score

Import the pretrained network GoogLeNet.

```matlab
net = googlenet;
```

Import the image and resize to match the input size for the network.

```matlab
X = imread("sherlock.jpg");
inputSize = net.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
```

Display the image.

```matlab
imshow(X)
```

Classify the image to get the class label.

```matlab
label = classify(net,X)
```
label = *categorical*
golden retriever

Use `occlusionSensitivity` to determine which parts of the image positively influence the classification result.

```matlab
scoreMap = occlusionSensitivity(net,X,label);
```

Plot the result over the original image with transparency to see which areas of the image affect the classification score.

```matlab
figure
imshow(X)
hold on
imagesc(scoreMap,'AlphaData',0.5);
colormap jet
```

The red parts of the map show the areas which have a positive contribution to the specified label. The dog's left eye and ear strongly influence the network's prediction of golden retriever.
You can get similar results using the gradient class activation mapping (Grad-CAM) technique. Grad-CAM uses the gradient of the classification score with respect to the last convolutional layer in a network in order to understand which parts of the image are most important for classification. For an example, see “Grad-CAM Reveals the Why Behind Deep Learning Decisions”.

**Input Arguments**

**net — Trained network**  
SeriesNetwork object | DAGNetwork object

Trained network, specified as a **SeriesNetwork** object or a **DAGNetwork** object. You can get a trained network by importing a pretrained network or by training your own network using the **trainNetwork** function. For more information about pretrained networks, see “Pretrained Deep Neural Networks”.

net must contain a single input layer. The input layer must be an **imageInputLayer**.

**X — Observation to occlude**  
numeric array

Observation to occlude, specified as a numeric array. You can calculate the occlusion map of one observation at a time. For example, specify a single image to understand which parts of that image affect classification results.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**label — Class label used to calculate change in classification score**  
categorical array | character array | string array

Class label used to calculate change in classification score, specified as a categorical, a character array, or a string array.

If label is specified as a vector, the change in classification score for each class label is calculated independently. In that case, **scoreMap(:, :, i)** corresponds to the occlusion map for the i-th element in label.

Data Types: char | string | categorical
Layer used to calculate change in activation, specified as a character vector or a string scalar. Specify `layer` as the name of the layer in `net` for which you want to compute the change in activations.

Data Types: `char` | `string`

Channel used to calculate change in activation, specified as scalar or vector of channel indices. The possible choices for `channel` depend on the selected layer. For example, for convolutional layers, the `NumFilters` property specifies the number of output channels. You can use `analyzeNetwork` to inspect the network and find out the number of output channels for each layer.

If `channel` is specified as a vector, the change in total activation for each specified channel is calculated independently. In that case, `activationMap(:,:,i)` corresponds to the occlusion map for the `i`th element in `channel`.

The function computes the change in total activation due to occlusion. The total activation is computed by summing over all spatial dimensions of the activation of that channel. The occlusion map corresponds to the difference between the total activation of the original data with no occlusion and the total activation for the occluded data. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified channel activation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'MaskSize',75,'OutputUpsampling','nearest' uses an occluding mask with size 75 pixels along each side, and uses nearest-neighbor interpolation to upsample the output to the same size as the input data.
**MaskSize — Size of occluding mask**

'auto' (default) | vector | scalar

Size of occluding mask, specified as the comma-separated pair consisting of 'MaskSize' and one of the following.

- 'auto' — Use a mask size of 20% of the input size, rounded to the nearest integer.
- A vector of the form [h w] — Use a rectangular mask with height h and width w.
- A scalar — Use a square mask with height and width equal to the specified value.

Example: 'MaskSize',[50 60]

**Stride — Step size for traversing mask across input data**

'auto' (default) | vector | scalar

Step size for traversing the mask across the input data, specified as the comma-separated pair consisting of 'Stride' and one of the following.

- 'auto' — Use a stride of 10% of the input size, rounded to the nearest integer.
- A vector of the form [a b] — Use a vertical stride of a and a horizontal stride of b.
- A scalar — Use a stride of the specified value in both the vertical and horizontal directions.

Example: 'Stride',30

**MaskValue — Replacement value of occluded region**

'auto' (default) | scalar | vector

Replacement value of occluded region, specified as the comma-separated pair consisting of 'MaskValue' and one of the following.

- 'auto' — Replace occluded pixels with the channel-wise mean of the input data.
- A scalar — Replace occluded pixels with the specified value.
- A vector — Replace occluded pixels with the value specified for each channel. The vector must contain the same number of elements as the number of output channels of the layer.

Example: 'MaskValue',0.5

**OutputUpsampling — Output upsampling method**

'bicubic' (default) | 'nearest' | 'none'
Output upsampling method, specified as the comma-separated pair consisting of 'OutputUpsampling' and one of the following.

- 'bicubic' — Use bicubic interpolation to produce a smooth map the same size as the input data.
- 'nearest' — Use nearest-neighbor interpolation expand the map to the same size as the input data. The map indicates the resolution of the occlusion computation with respect to the size of the input data.
- 'none' — Use no upsampling. The map can be smaller than the input data.

If 'OutputUpsampling' is 'bicubic' or 'bicubic', the computed map is upsampled to the size of the input data using the `imresize` function.

Example: 'OutputUpsampling','nearest'

**MaskClipping — Edge handling of the occluding mask**

'on' (default) | 'off'

Edge handling of the occluding mask, specified as the comma-separated pair consisting of 'MaskClipping' and one of the following.

- 'on' — Place the center of the first mask at the top-left corner of the input data. Masks at the edges of the data are not full size.
- 'off' — Place the top-left corner of the first mask at the top-left corner of the input data. Masks are always full size. If the values of the `MaskSize` and `Stride` options mean that some masks extend past the boundaries of the data, those masks are excluded.

For non-image input data, you can ensure you always occlude the same amount of input data using the option 'MaskClipping','off'. For example, for word embeddings data, you can ensure the same number of words are occluded at each point.

Example: 'MaskClipping','off'

**MiniBatchSize — Size of mini-batch**

128 (default) | positive integer

Size of the mini-batch to use for each training iteration, specified as the comma-separated pair consisting of 'MiniBatchSize' and a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights. See “Stochastic Gradient Descent” on page 1-691.
Example: 'MiniBatchSize',256

**ExecutionEnvironment — Hardware resource for training network**

'auto' (default) | 'cpu' | 'gpu'

Hardware resource for training network, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following.

- 'auto' — Use a GPU if one is available. Otherwise, use the CPU.
- 'cpu' — Use the CPU.
- 'gpu' — Use the GPU.

The GPU option requires Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If you choose the 'ExecutionEnvironment','gpu' option and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

Example: 'ExecutionEnvironment','gpu'

**Output Arguments**

**scoreMap — Map of change of classification score**

numeric matrix | numeric array

Map of change of classification score, returned as a numeric matrix or a numeric array. The change in classification score is calculated relative to the original input data without occlusion. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified classification label.

If `label` is specified as a vector, the change in classification score for each class label is calculated independently. In that case, `scoreMap(:,:,i)` corresponds to the occlusion map for the ith element in `label`.

**activationMap — Map of change of total activation**

numeric matrix | numeric array

Map of change of total activation, returned as a numeric matrix or a numeric array.

The function computes the change in total activation due to occlusion. The total activation is computed by summing over all spatial dimensions of the activation of that channel.
occlusion map corresponds to the difference between the total activation of the original data with no occlusion and the total activation for the occluded data. Areas in the map with higher positive values correspond to regions of input data that contribute positively to the specified channel activation.

If `channels` is specified as a vector, the change in total activation for each specified channel is calculated independently. In that case, `activationMap(:,:,i)` corresponds to the occlusion map for the `i`th element in `channel`.

### See Also

- `activations` | `classify`

### Topics

- “Understand Network Predictions Using Occlusion”
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions”
- “Investigate Network Predictions Using Class Activation Mapping”
- “Visualize Features of a Convolutional Neural Network”
- “Visualize Activations of a Convolutional Neural Network”

### Introduced in R2019b
regressionLayer

Create a regression output layer

Syntax

layer = regressionLayer
layer = regressionLayer(Name,Value)

Description

A regression layer computes the half-mean-squared-error loss for regression problems.

layer = regressionLayer returns a regression output layer for a neural network as a RegressionOutputLayer object.

Predict responses of a trained regression network using predict. Normalizing the responses often helps stabilizing and speeding up training of neural networks for regression. For more information, see “Train Convolutional Neural Network for Regression”.

layer = regressionLayer(Name,Value) sets the optional Name and ResponseNames properties using name-value pairs. For example, regressionLayer('Name','output') creates a regression layer with the name 'output'. Enclose each property name in single quotes.

Examples

Create Regression Output Layer

Create a regression output layer with the name 'routput'.

layer = regressionLayer('Name','routput')

layer =
    RegressionOutputLayer with properties:
The default loss function for regression is mean-squared-error.

Include a regression output layer in a Layer array.

```matlab
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer
];
```

### Input Arguments

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `regressionLayer('Name','output')` creates a regression layer with the name `'output'`

**Name — Layer name**

Character vector | string scalar
void | empty

`''` (default)
Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

ResponseNames — Names of responses

{} (default) | cell array of character vectors | string array

Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is {}.

Data Types: cell

Output Arguments

Layer — Regression output layer

RegressionOutputLayer object

Regression output layer, returned as a RegressionOutputLayer object.

More About

Regression Output Layer

A regression layer computes the half-mean-squared-error loss for regression problems. For typical regression problems, a regression layer must follow the final fully connected layer.

For a single observation, the mean-squared-error is given by:

\[ \text{MSE} = \sum_{i=1}^{R} \frac{(t_i - y_i)^2}{R}, \]

where \( R \) is the number of responses, \( t_i \) is the target output, and \( y_i \) is the network’s prediction for response \( i \).
For image and sequence-to-one regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses, not normalized by $R$:

$$\text{loss} = \frac{1}{2} \sum_{i=1}^{R} (t_i - y_i)^2.$$ 

For image-to-image regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses for each pixel, not normalized by $R$:

$$\text{loss} = \frac{1}{2} \sum_{p=1}^{HWC} (t_p - y_p)^2,$$

where $H$, $W$, and $C$ denote the height, width, and number of channels of the output respectively, and $p$ indexes into each element (pixel) of $t$ and $y$ linearly.

For sequence-to-sequence regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses for each time step, not normalized by $R$:

$$\text{loss} = \frac{1}{2S} \sum_{i=1}^{S} \sum_{j=1}^{R} (t_{ij} - y_{ij})^2,$$

where $S$ is the sequence length.

When training, the software calculates the mean loss over the observations in the mini-batch.

**Extended Capabilities**

**GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.
See Also
RegressionOutputLayer | classificationLayer | fullyConnectedLayer

Topics
“Deep Learning in MATLAB”
“Train Convolutional Neural Network for Regression”

Introduced in R2017a
alexnet

Pretrained AlexNet convolutional neural network

Syntax

net = alexnet

Description

AlexNet is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 8 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 227-by-227. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the AlexNet network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with AlexNet.

For a free hands-on introduction to practical deep learning methods, see Deep Learning Onramp.

net = alexnet returns a pretrained AlexNet network.

This function requires Deep Learning Toolbox Model for AlexNet Network support package. If this support package is not installed, the function provides a download link. Alternatively, see Deep Learning Toolbox Model for AlexNet Network.

For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

Examples
**Download AlexNet Support Package**

Download and install Deep Learning Toolbox Model for AlexNet Network support package.

Type `alexnet` at the command line.

```matlab
alexnet
```

If Deep Learning Toolbox Model for AlexNet Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `alexnet` at the command line.

```matlab
alexnet
ans =

    SeriesNetwork with properties:

        Layers: [25×1 nnet.cnn.layer.Layer]
```

If the required support package is installed, then the function returns a `SeriesNetwork` object.

**Interactive Transfer Learning Using AlexNet**

This example shows how to fine-tune a pretrained AlexNet network to classify a new collection of images. This process is called transfer learning and is usually much faster and easier than training a new network, because you can apply learned features to a new task using a smaller number of training images. To interactively prepare a network for transfer learning, use Deep Network Designer.

**Load Pretrained Network**

Load a pretrained AlexNet network. If you need to download the network, use the download link.

```matlab
net = alexnet;
```

**Import Network into Deep Network Designer**

Open Deep Network Designer.
Click **Import** and select the network from the workspace. Deep Network Designer displays a zoomed out view of the whole network. Explore the network plot. To zoom in with the mouse, use **Ctrl**+scroll wheel.

**Edit Network for Transfer Learning**

To retrain a pretrained network to classify new images, replace the final layers with new layers adapted to the new data set. You must change the number of classes to match your data.

Drag a new **fullyConnectedLayer** from the **Layer Library** onto the canvas. Edit the **OutputSize** to the number of classes in the new data, in this example, 5.
Edit learning rates to learn faster in the new layers than in the transferred layers. Set WeightLearnRateFactor and BiasLearnRateFactor to 10. Delete the original layer and connect up your new layer instead.

Replace the output layer. Scroll to the end of the Layer Library and drag a new classificationLayer onto the canvas. Delete the original output layer and connect up your new layer instead.
**Check Network**

To make sure your edited network is ready for training, click **Analyze** and ensure that the Deep Learning Network Analyzer reports zero errors.
Export Network for Training

Return to the Deep Network Designer and click **Export**. The Deep Network Designer exports the network to a new variable called `layers_1` containing the edited network layers. You can now supply the layer variable to the `trainNetwork` function.

Load Data and Train Network

Unzip and load the new images as an image datastore. Divide the data into 70% training data and 30% validation data.

```matlab
unzip('MerchData.zip');
imds = imageDatastore('MerchData','IncludeSubfolders',true,'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

Resize images to match the pretrained network input size.

```matlab
augimdsTrain = augmentedImageDatastore([227 227],imdsTrain);
augimdsValidation = augmentedImageDatastore([227 227],imdsValidation);
```
Specify training options.

- Specify the mini-batch size, that is, how many images to use in each iteration.
- Specify a small number of epochs. An epoch is a full training cycle on the entire training data set. For transfer learning, you do not need to train for as many epochs. Shuffle the data every epoch.
- Set InitialLearnRate to a small value to slow down learning in the transferred layers.
- Specify validation data and validation frequency so that the accuracy on the validation data is calculated once every epoch.
- Turn on the training plot to monitor progress while you train.

```matlab
miniBatchSize = 10;
valFrequency = floor(numel(augimdsTrain.Files)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'MiniBatchSize',miniBatchSize, ...
    'MaxEpochs',5, ...
    'InitialLearnRate',3e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',valFrequency, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

To train the network, supply the layers exported from the app, layers_1, the training images, and options, to the `trainNetwork` function. By default, `trainNetwork` uses a GPU if available (requires Parallel Computing Toolbox™). Otherwise, it uses a CPU. Training is fast because the data set is so small.

```matlab
trainedNet = trainNetwork(augimdsTrain,layers_1,options);
```
Test Trained Network

Classify the validation images using the fine-tuned network, and calculate the classification accuracy.

```matlab
YPred,probs = classify(trainedNet,augimdsValidation);
accuracy = mean(YPred == imdsValidation.Labels)
```

accuracy = 1

Display four sample validation images with predicted labels and predicted probabilities.

```matlab
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label) +", " + num2str(100*max(probs(idx(i),:)),3) + "%")
end
```
Transfer Learning Using AlexNet

This example shows how to fine-tune a pretrained AlexNet convolutional neural network to perform classification on a new collection of images.

AlexNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.
Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

Load Data

Unzip and load the new images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```matlab
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ... 'IncludeSubfolders',true, ... 'LabelSource','foldernames');
```

Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the images datastore into two new datastores.

```matlab
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

This very small data set now contains 55 training images and 20 validation images. Display some sample images.

```matlab
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
```
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end

Load Pretrained Network

Load the pretrained AlexNet neural network. If Deep Learning Toolbox™ Model for
AlexNet Network is not installed, then the software provides a download link. AlexNet is
trained on more than one million images and can classify images into 1000 object
categories, such as keyboard, mouse, pencil, and many animals. As a result, the model
has learned rich feature representations for a wide range of images.
net = alexnet;

Use `analyzeNetwork` to display an interactive visualization of the network architecture and detailed information about the network layers.

```matlab
analyzeNetwork(net)
```

The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

```matlab
inputSize = net.Layers(1).InputSize
```

```matlab
inputSize = 1×3
227   227     3
```
Replace Final Layers

The last three layers of the pretrained network `net` are configured for 1000 classes. These three layers must be fine-tuned for the new classification problem. Extract all layers, except the last three, from the pretrained network.

```matlab
layersTransfer = net.Layers(1:end-3);
```

Transfer the layers to the new classification task by replacing the last three layers with a fully connected layer, a softmax layer, and a classification output layer. Specify the options of the new fully connected layer according to the new data. Set the fully connected layer to have the same size as the number of classes in the new data. To learn faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```matlab
numClasses = numel(categories(imdsTrain.Labels))
numClasses = 5
layers = [layersTransfer
        fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20)
        softmaxLayer
        classificationLayer];
```

Train Network

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```matlab
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augImdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ... 
    'DataAugmentation',imageAugmenter);
```
To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```matlab
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. The software validates the network every ValidationFrequency iterations during training.

```matlab
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ... 
    'MaxEpochs',6, ... 
    'InitialLearnRate',1e-4, ... 
    'Shuffle','every-epoch', ... 
    'ValidationData',augimdsValidation, ... 
    'ValidationFrequency',3, ... 
    'Verbose',false, ... 
    'Plots','training-progress');
```

Train the network that consists of the transferred and new layers. By default, trainNetwork uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the 'ExecutionEnvironment' name-value pair argument of trainingOptions.

```matlab
netTransfer = trainNetwork(augimdsTrain,layers,options);
```
Classify Validation Images

Classify the validation images using the fine-tuned network.

\[ [\text{YPred},\text{scores}] = \text{classify(netTransfer,augimdsValidation)}; \]

Display four sample validation images with their predicted labels.

\[ \text{idx} = \text{randperm(numel(imdsValidation.Files)},4); \]
\[ \text{figure} \]
\[ \text{for } i = 1:4 \]
\[ \text{subplot}(2,2,i) \]
\[ \text{I} = \text{readimage(imdsValidation,idx(i))}; \]
\[ \text{imshow(I)} \]
\[ \text{label} = \text{YPred(idx(i))}; \]
\[ \text{title(string(label))}; \]
\[ \text{end} \]
Calculate the classification accuracy on the validation set. Accuracy is the fraction of labels that the network predicts correctly.

\[
Y_{\text{Validation}} = \text{imdsValidation.Labels}; \\
\text{accuracy} = \text{mean}(\text{YPred} == Y_{\text{Validation}})
\]

\[
\text{accuracy} = 1
\]

For tips on improving classification accuracy, see “Deep Learning Tips and Tricks”.

**Classify an Image Using AlexNet**

Read, resize, and classify an image using AlexNet. First, load a pretrained AlexNet model.
net = alexnet;

Read the image using `imread`.

I = imread('peppers.png');
figure
imshow(I)

The pretrained model requires the image size to be the same as the input size of the network. Determine the input size of the network using the `InputSize` property of the first layer of the network.

sz = net.Layers(1).InputSize
sz = 1×3
    227   227     3

Crop the image to the input size of the network. Alternatively, you can resize the image using `imresize`.

I = I(1:sz(1),1:sz(2),1:sz(3));
figure
imshow(I)

Classify the image using `classify`.

label = classify(net,I)

label = `categorical`
    bell pepper

Show the image and classification result together.
Feature Extraction Using AlexNet

This example shows how to extract learned image features from a pretrained convolutional neural network, and use those features to train an image classifier. Feature extraction is the easiest and fastest way use the representational power of pretrained deep networks. For example, you can train a support vector machine (SVM) using fitcecoc (Statistics and Machine Learning Toolbox™) on the extracted features. Because feature extraction only requires a single pass through the data, it is a good starting point if you do not have a GPU to accelerate network training with.

Load Data

Unzip and load the sample images as an image datastore. imageDatastore automatically labels the images based on folder names and stores the data as an
ImageDatastore object. An image datastore lets you store large image data, including data that does not fit in memory. Split the data into 70% training and 30% test data.

```matlab
unzip('MerchData.zip');

imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...  
    'LabelSource','foldernames');

[imdsTrain,imdsTest] = splitEachLabel(imds,0.7,'randomized');
```

There are now 55 training images and 20 validation images in this very small data set. Display some sample images.

```matlab
numImagesTrain = numel(imdsTrain.Labels);  
idx = randperm(numImagesTrain,16);

for i = 1:16
    I{i} = readimage(imdsTrain,idx(i));
end

figure
imshow(imtile(I))
```
Load Pretrained Network

Load a pretrained AlexNet network. If the Deep Learning Toolbox Model for AlexNet Network support package is not installed, then the software provides a download link. AlexNet is trained on more than a million images and can classify images into 1000 object categories. For example, keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```matlab
net = alexnet;
```

Display the network architecture. The network has five convolutional layers and three fully connected layers.

```matlab
net.Layers
```

```
test = LayerConfiguration
ans =
    25x1 Layer array with layers:
        1   'data'     Image Input                   227x227x3 images with 'zerocenter' normalization
        2   'conv1'    Convolution                   96 11x11x3 convolutions with stride [4  4] and padding [0  0  0  0] ReLU
        3   'relu1'    ReLU                          ReLU
        4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per element
        5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0] ReLU
        6   'conv2'    Grouped Convolution           2 groups of 128 5x5x48 convolutions with stride [1  1] and padding [2  2  2  2] ReLU
        7   'relu2'    ReLU                          ReLU
        8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per element
        9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0] ReLU
       10   'conv3'    Convolution                   384 3x3x256 convolutions with stride [1  1] and padding [1  1  1  1] ReLU
       11   'relu3'    ReLU                          ReLU
       12   'conv4'    Grouped Convolution           2 groups of 192 3x3x192 convolutions with stride [1  1] and padding [1  1  1  1] ReLU
       13   'relu4'    ReLU                          ReLU
       14   'conv5'    Grouped Convolution           2 groups of 128 3x3x192 convolutions with stride [1  1] and padding [1  1  1  1] ReLU
       15   'relu5'    ReLU                          ReLU
       16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0] ReLU
       17   'fc6'      Fully Connected               4096 fully connected layer
       18   'relu6'    ReLU                          ReLU
       19   'drop6'    Dropout                       50% dropout
       20   'fc7'      Fully Connected               4096 fully connected layer
       21   'relu7'    ReLU                          ReLU
       22   'drop7'    Dropout                       50% dropout
       23   'fc8'      Fully Connected               1000 fully connected layer ReLU
       24   'prob'     Softmax                      softmax
       25   'output'   Classification Output         crossentropyex with 'tench' and 999 other classes
```
The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

```MATLAB
inputSize = net.Layers(1).InputSize
inputSize = 1×3
     227     227     3
```

**Extract Image Features**

The network constructs a hierarchical representation of input images. Deeper layers contain higher-level features, constructed using the lower-level features of earlier layers. To get the feature representations of the training and test images, use `activations` on the fully connected layer 'fc7'. To get a lower-level representation of the images, use an earlier layer in the network.

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. To automatically resize the training and test images before they are input to the network, create augmented image datastores, specify the desired image size, and use these datastores as input arguments to `activations`.

```MATLAB
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain); augimdsTest = augmentedImageDatastore(inputSize(1:2),imdsTest);

layer = 'fc7';
featuresTrain = activations(net,augimdsTrain,layer,'OutputAs','rows'); featuresTest = activations(net,augimdsTest,layer,'OutputAs','rows');
```

Extract the class labels from the training and test data.

```MATLAB
YTrain = imdsTrain.Labels;
YTest = imdsTest.Labels;
```

**Fit Image Classifier**

Use the features extracted from the training images as predictor variables and fit a multiclass support vector machine (SVM) using `fitcecoc` (Statistics and Machine Learning Toolbox).

```MATLAB
mdl = fitcecoc(featuresTrain,YTrain);
Classify Test Images

Classify the test images using the trained SVM model and the features extracted from the test images.

YPred = predict(mdl,featuresTest);

Display four sample test images with their predicted labels.

idx = [1 5 10 15];
figure
for i = 1:numel(idx)
    subplot(2,2,i)
    I = readimage(imdsTest,idx(i));
    label = YPred(idx(i));
    imshow(I)
    title(label)
end
Calculate the classification accuracy on the test set. Accuracy is the fraction of labels that the network predicts correctly.

\[
\text{accuracy} = \text{mean}(\text{YPred} == \text{YTest})
\]

\[
\text{accuracy} = 1
\]
This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

**Output Arguments**

- **net** — Pretrained AlexNet convolutional neural network
  
  SeriesNetwork object

  Pretrained AlexNet convolutional neural network returned as a SeriesNetwork object.

**Tips**

- For a free hands-on introduction to practical deep learning methods, see Deep Learning Onramp.

**References**


**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.
For code generation, you can load the network by using the syntax `net = alexnet` or by passing the `alexnet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('alexnet')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

**GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = alexnet` or by passing the `alexnet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('alexnet')`.

**See Also**

- Deep Network Designer
- densenet201
- googlenet
- importCaffeNetwork
- importKerasNetwork
- inceptionresnetv2
- resnet18
- resnet50
- squeezenet
- vgg16
- vgg19

**Topics**

- “Deep Learning in MATLAB”
- “Classify Webcam Images Using Deep Learning”
- “Pretrained Deep Neural Networks”
- “Train Deep Learning Network to Classify New Images”
- “Deep Dream Images Using AlexNet”
- “Visualize Features of a Convolutional Neural Network”
- “Visualize Activations of a Convolutional Neural Network”
- “Deep Learning Tips and Tricks”

**Introduced in R2017a**
vgg16

Pretrained VGG-16 convolutional neural network

Syntax

net = vgg16

Description

VGG-16 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 16 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the VGG-16 network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with VGG-16.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load VGG-16 instead of GoogLeNet.

net = vgg16 returns a pretrained VGG-16 network.

This function requires Deep Learning Toolbox Model for VGG-16 Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download VGG-16 Support Package

Download and install Deep Learning Toolbox Model for VGG-16 Network support package.
Type `vgg16` at the command line.

`vgg16`

If Deep Learning Toolbox Model for VGG-16 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `vgg16` at the command line.

`vgg16`

```
ans =
    SeriesNetwork with properties:
        Layers: [41x1 nnet.cnn.layer.Layer]
```

**Load Pretrained VGG-16 Convolutional Neural Network**

Load a pretrained VGG-16 convolutional neural network and examine the layers and classes.

Use `vgg16` to load the pretrained VGG-16 network. The output `net` is a `SeriesNetwork` object.

```
net = vgg16
```

```
net =
    SeriesNetwork with properties:
        Layers: [41x1 nnet.cnn.layer.Layer]
```

View the network architecture using the `Layers` property. The network has 41 layers. There are 16 layers with learnable weights: 13 convolutional layers, and 3 fully connected layers.

```
net.Layers
```

```
ans =
    41x1 Layer array with layers:
        1{'input'}     Image Input    224x224x3 images with 'zerocenter' normalization
```
To view the names of the classes learned by the network, you can view the Classes property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.
net.Layers(end).Classes(1:10)

ans = 10×1 categorical array
tench
goldfish
great white shark	
tiger shark
hammerhead
electric ray
stingray
cock
hen
ostrich

Output Arguments

net — Pretrained VGG-16 convolutional neural network
SeriesNetwork object

Pretrained VGG-16 convolutional neural network returned as a SeriesNetwork object.

References


Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = vgg16` or by passing the `vgg16` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg16')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

**GPU Code Generation**
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = vgg16` or by passing the `vgg16` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg16')`

See Also
alexnet | densenet201 | googlenet | inceptionresnetv2 | resnet101 | resnet18 | resnet50 | squeezenet | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Transfer Learning Using AlexNet”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2017a
vgg19

Pretrained VGG-19 convolutional neural network

Syntax

net = vgg19

Description

VGG-19 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 19 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the VGG-19 network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with VGG-19.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load VGG-19 instead of GoogLeNet.

net = vgg19 returns a pretrained VGG-19 network.

This function requires Deep Learning Toolbox Model for VGG-19 Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download VGG-19 Support Package

This example shows how to download and install Deep Learning Toolbox Model for VGG-19 Network support package.
Type `vgg19` at the command line.

`vgg19`

If Deep Learning Toolbox Model for VGG-19 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `vgg19` at the command line.

`vgg19`

```matlab
ans =
    SeriesNetwork with properties:
        Layers: [47×1 nnet.cnn.layer.Layer]
```

**Load Pretrained VGG-19 Convolutional Neural Network**

Load a pretrained VGG-19 convolutional neural network and examine the layers and classes.

Use `vgg19` to load a pretrained VGG-19 network. The output `net` is a `SeriesNetwork` object.

```matlab
net = vgg19
```

```matlab
net =
    SeriesNetwork with properties:
        Layers: [47×1 nnet.cnn.layer.Layer]
```

View the network architecture using the `Layers` property. The network has 47 layers. There are 19 layers with learnable weights: 16 convolutional layers, and 3 fully connected layers.

```matlab
net.Layers
```

```matlab
ans =
    47x1 Layer array with layers:
        1       'input'   Image Input   224x224x3 images with 'zerocenter' normalization:
          2           'conv1_1'  Convolution  224x224x3 images with 'zerocenter' normalization:
          3           'relu1_1'  ReLU         224x224x3 images with 'zerocenter' normalization:
          4           'conv1_2'  Convolution  224x224x3 images with 'zerocenter' normalization:
          5           'relu1_2'  ReLU         224x224x3 images with 'zerocenter' normalization:
          6           'Pool1'    Pooling      224x112x3 images with 'zerocenter' normalization:
          7           'conv2_1'  Convolution  224x112x3 images with 'zerocenter' normalization:
          8           'relu2_1'  ReLU         224x112x3 images with 'zerocenter' normalization:
          9           'conv2_2'  Convolution  224x112x3 images with 'zerocenter' normalization:
         10           'relu2_2'  ReLU         224x112x3 images with 'zerocenter' normalization:
         11           'Pool2'    Pooling      224x56x3 images with 'zerocenter' normalization:
         12           'conv3_1'  Convolution  224x56x3 images with 'zerocenter' normalization:
         13           'relu3_1'  ReLU         224x56x3 images with 'zerocenter' normalization:
         14           'conv3_2'  Convolution  224x56x3 images with 'zerocenter' normalization:
         15           'relu3_2'  ReLU         224x56x3 images with 'zerocenter' normalization:
         16           'conv3_3'  Convolution  224x56x3 images with 'zerocenter' normalization:
         17           'relu3_3'  ReLU         224x56x3 images with 'zerocenter' normalization:
         18           'Pool3'    Pooling      224x28x3 images with 'zerocenter' normalization:
         19           'conv4_1'  Convolution  224x28x3 images with 'zerocenter' normalization:
         20           'relu4_1'  ReLU         224x28x3 images with 'zerocenter' normalization:
         21           'conv4_2'  Convolution  224x28x3 images with 'zerocenter' normalization:
         22           'relu4_2'  ReLU         224x28x3 images with 'zerocenter' normalization:
         23           'conv4_3'  Convolution  224x28x3 images with 'zerocenter' normalization:
         24           'relu4_3'  ReLU         224x28x3 images with 'zerocenter' normalization:
         25           'Pool4'    Pooling      224x14x3 images with 'zerocenter' normalization:
         26           'conv5_1'  Convolution  224x14x3 images with 'zerocenter' normalization:
         27           'relu5_1'  ReLU         224x14x3 images with 'zerocenter' normalization:
         28           'conv5_2'  Convolution  224x14x3 images with 'zerocenter' normalization:
         29           'relu5_2'  ReLU         224x14x3 images with 'zerocenter' normalization:
         30           'conv5_3'  Convolution  224x14x3 images with 'zerocenter' normalization:
         31           'relu5_3'  ReLU         224x14x3 images with 'zerocenter' normalization:
         32           'Pool5'    Pooling      224x7x3 images with 'zerocenter' normalization:
         33           'fc6'      Fully Connected 32768 images with 'zerocenter' normalization:
         34           'fc7'      Fully Connected 4096 images with 'zerocenter' normalization:
         35           'fc8'      Fully Connected 1000 images with 'zerocenter' normalization:
         36           'softmax'  Softmax      1000 classes:
<table>
<thead>
<tr>
<th></th>
<th>Layer</th>
<th>Operation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>'conv1_1'</td>
<td>Convolution</td>
<td>64 3x3x3</td>
</tr>
<tr>
<td>3</td>
<td>'relu1_1'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>'conv1_2'</td>
<td>Convolution</td>
<td>64 3x3x64</td>
</tr>
<tr>
<td>5</td>
<td>'relu1_2'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>'pool1'</td>
<td>Max Pooling</td>
<td>2x2 max pool</td>
</tr>
<tr>
<td>7</td>
<td>'conv2_1'</td>
<td>Convolution</td>
<td>128 3x3x64</td>
</tr>
<tr>
<td>8</td>
<td>'relu2_1'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>'conv2_2'</td>
<td>Convolution</td>
<td>128 3x3x128</td>
</tr>
<tr>
<td>10</td>
<td>'relu2_2'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>'pool2'</td>
<td>Max Pooling</td>
<td>2x2 max pool</td>
</tr>
<tr>
<td>12</td>
<td>'conv3_1'</td>
<td>Convolution</td>
<td>256 3x3x128</td>
</tr>
<tr>
<td>13</td>
<td>'relu3_1'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>'conv3_2'</td>
<td>Convolution</td>
<td>256 3x3x256</td>
</tr>
<tr>
<td>15</td>
<td>'relu3_2'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>'conv3_3'</td>
<td>Convolution</td>
<td>256 3x3x256</td>
</tr>
<tr>
<td>17</td>
<td>'relu3_3'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>'conv3_4'</td>
<td>Convolution</td>
<td>256 3x3x256</td>
</tr>
<tr>
<td>19</td>
<td>'relu3_4'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>'pool3'</td>
<td>Max Pooling</td>
<td>2x2 max pool</td>
</tr>
<tr>
<td>21</td>
<td>'conv4_1'</td>
<td>Convolution</td>
<td>512 3x3x256</td>
</tr>
<tr>
<td>22</td>
<td>'relu4_1'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>'conv4_2'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>24</td>
<td>'relu4_2'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>'conv4_3'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>26</td>
<td>'relu4_3'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>'conv4_4'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>28</td>
<td>'relu4_4'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>'pool4'</td>
<td>Max Pooling</td>
<td>2x2 max pool</td>
</tr>
<tr>
<td>30</td>
<td>'conv5_1'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>31</td>
<td>'relu5_1'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>'conv5_2'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>33</td>
<td>'relu5_2'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>'conv5_3'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>35</td>
<td>'relu5_3'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>'conv5_4'</td>
<td>Convolution</td>
<td>512 3x3x512</td>
</tr>
<tr>
<td>37</td>
<td>'relu5_4'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>'pool5'</td>
<td>Max Pooling</td>
<td>2x2 max pool</td>
</tr>
<tr>
<td>39</td>
<td>'fc6'</td>
<td>Fully Connected</td>
<td>4096</td>
</tr>
<tr>
<td>40</td>
<td>'relu6'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>'drop6'</td>
<td>Dropout</td>
<td>50% dropout</td>
</tr>
<tr>
<td>42</td>
<td>'fc7'</td>
<td>Fully Connected</td>
<td>4096</td>
</tr>
<tr>
<td>43</td>
<td>'relu7'</td>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>'drop7'</td>
<td>Dropout</td>
<td>50% dropout</td>
</tr>
<tr>
<td>45</td>
<td>'fc8'</td>
<td>Fully Connected</td>
<td>1000</td>
</tr>
</tbody>
</table>
46 'prob'    Softmax    softmax
47 'output'  Classification Output  crossentropyex with 'tench' and 999 other classes

To view the names of the classes learned by the network, you can view the Classes property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical array
tench
goldfish
great white shark
tiger shark
hammerhead
electric ray
stingray
cock
hen
ostrich
```

## Output Arguments

**net** — Pretrained VGG-19 convolutional neural network

`SeriesNetwork` object

Pretrained VGG-19 convolutional neural network returned as a `SeriesNetwork` object.

## References


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = vgg19` or by passing the `vgg19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg19')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = vgg19` or by passing the `vgg19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg19')`

See Also
`alexnet | deepDreamImage | densenet201 | googlenet | inceptionresnetv2 | resnet101 | resnet18 | resnet50 | squeezenet | vgg16`

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Transfer Learning Using AlexNet”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2017a
importCaffeLayers

Import convolutional neural network layers from Caffe

Syntax

`layers = importCaffeLayers(protofile)`  
`layers = importCaffeLayers(protofile,'InputSize',sz)`

Description

`layers = importCaffeLayers(protofile)` imports the layers of a Caffe [1] network. The function returns the layers defined in the `.prototxt` file `protofile`.

This function requires Deep Learning Toolbox Importer for Caffe Models support package. If this support package is not installed, then the function provides a download link.

You can download pretrained networks from Caffe Model Zoo [2].

`layers = importCaffeLayers(protofile,'InputSize',sz)` specifies the size of the input data. If the `.prototxt` file does not specify the size of the input data, then you must specify the input size.

Examples

Download Importer for Caffe Models Support Package

Download and install Deep Learning Toolbox Importer for Caffe Models support package.

Download the required support package by typing `importCaffeLayers` at the command line.

`importCaffeLayers`
If Deep Learning Toolbox Importer for Caffe Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install.

**Import Layers from Caffe Network**

Specify the example file 'digitsnet.prototxt' to import.

```plaintext
protofile = 'digitsnet.prototxt';
```

Import the network layers.

```plaintext
layers = importCaffeLayers(protofile)
```

```
layers =
1x7 Layer array with layers:
   1   'testdata'   Image Input             28x28x1 images
   2   'conv1'      Convolution             20 5x5x1 convolutions with stride [1  1] and padding [0  0]
   3   'relu1'      ReLU                    ReLU
   4   'pool1'      Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0]
   5   'ipl1'       Fully Connected         10 fully connected layer
   6   'loss'       Softmax                 softmax
   7   'output'     Classification Output   crossentropyex with 'class1', 'class2', and 8 other classes
```

**Input Arguments**

protofile — File name

character vector | string scalar

File name of the .prototxt file containing the network architecture, specified as a character vector or a string scalar. protofile must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file. If the .prototxt file does not specify the size of the input data, you must specify the size using the sz input argument.

Example: 'digitsnet.prototxt'

sz — Size of input data

row vector
Size of input data, specified as a row vector. Specify a vector of two or three integer values \([h,w]\), or \([h,w,c]\) corresponding to the height, width, and the number of channels of the input data.

Example: \([28\ 28\ 1]\)

## Output Arguments

**layers** — Network architecture

Layer array | LayerGraph object

Network architecture, returned as a Layer array or a LayerGraph object. Caffe networks that take color images as input expect the images to be in BGR format. During import, `importCaffeLayers` modifies the network so that the imported MATLAB network takes RGB images as input.

## Tips

- `importCaffeLayers` can import networks with the following Caffe layer types, with some limitations:

  - Input
  - Convolution
  - Deconvolution
  - Batch Norm
  - ReLU
  - Tanh
  - Local Response Normalization (LRN)
  - Pooling
  - Inner Product/Fully Connected
  - Dropout
  - Concat
  - Eltwise (only sum)
  - Softmax With Loss
- Euclidean Loss
- Sigmoid

If the network contains any other type of layer, then the software returns an error.

The function imports only the layers that `protofile` specifies with the include-phase `TEST`. The function ignores any layers that `protofile` specifies with the include-phase `TRAIN`.

**References**


**See Also**

assembleNetwork | exportONNXNetwork | importCaffeNetwork |
importKerasLayers | importKerasNetwork | importONNXLayers |
importONNXNetwork

**Topics**

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”

**Introduced in R2017a**
importCaffeNetwork

Import pretrained convolutional neural network models from Caffe

**Syntax**

```matlab
net = importCaffeNetwork(protofile,datafile)
net = importCaffeNetwork(____,Name,Value)
```

**Description**

`net = importCaffeNetwork(protofile,datafile)` imports a pretrained network from Caffe [1]. The function returns the pretrained network with the architecture specified by the `.prototxt` file `protofile` and with network weights specified by the `.caffemodel` file `datafile`.

This function requires Deep Learning Toolbox Importer for Caffe Models support package. If this support package is not installed, the function provides a download link.

You can download pretrained networks from Caffe Model Zoo [2].

`net = importCaffeNetwork(____,Name,Value)` returns a network with additional options specified by one or more `Name,Value` pair arguments using any of the previous syntaxes.

**Examples**

**Download Importer for Caffe Models Support Package**

Download and install Deep Learning Toolbox Importer for Caffe Models support package.

To download the required support package, type `importCaffeNetwork` at the command line.

`importCaffeNetwork`
If Deep Learning Toolbox Importer for Caffe Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install.

Import Caffe Network

Specify files to import.

```matlab
protofile = 'digitsnet.prototxt';
datafile = 'digits_iter_10000.caffemodel';
```

Import network.

```matlab
net = importCaffeNetwork(protofile,datafile)
```

net = SeriesNetwork with properties:

- Layers: [7x1 nnet.cnn.layer.Layer]

Input Arguments

**protofile** — File name
character vector | string scalar

File name of the .prototxt file containing the network architecture, specified as a character vector or a string scalar. protofile must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file. If the .prototxt file does not specify the size of the input data, you must specify the size using the 'InputSize' name-value pair argument.

Example: 'digitsnet.prototxt'

**datafile** — File name
character vector | string scalar

File name of the .caffemodel file containing the network weights, specified as a character vector or a string scalar. datafile must be in the current folder, in a folder on
the MATLAB path, or you must include a full or relative path to the file. To import network layers without weights, use `importCaffeLayers`.

Example: 'digits_iter_10000.caffemodel'

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importCaffeNetwork(protofile,datafile,'AverageImage',I)` imports a pretrained network using the average image `I` for zero-center normalization.

**InputSize — Size of input data**

row vector

Size of input data, specified as a row vector. Specify a vector of two or three integer values `[h,w]`, or `[h,w,c]` corresponding to the height, width, and the number of channels of the input data. If the `.prototxt` file does not specify the size of the input data, then you must specify the input size.

Example: `[28 28 1]`

**AverageImage — Average image**

matrix

Average image for zero-center normalization, specified as a matrix. If you specify an image, then you must specify an image of the same size as the input data. If you do not specify an image, the software uses the data specified in the `.prototxt` file, if present. Otherwise, the function sets the `Normalization` property of the image input layer of the network to 'none'.

**Classes — Classes of the output layer**

'auto' (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If you specify a string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str,str)`. If `Classes` is 'auto', then the function sets the classes to `categorical(1:N)`, where `N` is the number of classes.
Data Types: char | categorical | string | cell

Output Arguments

**net — Imported pretrained Caffe network**

SeriesNetwork object | DAGNetwork object

Imported pretrained Caffe network, returned as a SeriesNetwork object or DAGNetwork object. Caffe networks that take color images as input expect the images to be in BGR format. During import, `importCaffeNetwork` modifies the network so that the imported MATLAB network takes RGB images as input.

Tips

- `importCaffeNetwork` can import networks with the following Caffe layer types, with some limitations:
  - Input
  - Convolution
  - Deconvolution
  - Batch Norm
  - ReLU
  - Tanh
  - Local Response Normalization (LRN)
  - Pooling
  - Inner Product/Fully Connected
  - Dropout
  - Concat
  - Eltwise (only sum)
  - Softmax With Loss
  - Euclidean Loss
  - Sigmoid

If the network contains any other type of layer, then the software returns an error.
The function imports only the layers that `protofile` specifies with the include-phase `TEST`. The function ignores any layers that `protofile` specifies with the include-phase `TRAIN`.

**Compatibility Considerations**

'`ClassNames'` option will be removed  
*Not recommended starting in R2018b*

'`ClassNames'` will be removed. Use 'Classes' instead. To update your code, replace all instances of 'ClassNames' with 'Classes'. There are some differences between the corresponding properties in classification output layers that require additional updates to your code.

The `ClassNames` property of a classification output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.

**References**


**Extended Capabilities**

**GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = importCaffeNetwork`. 
See Also
assembleNetwork | exportONNXNetwork | importCaffeLayers |
importKerasLayers | importKerasNetwork | importONNXLayers |
importONNXNetwork

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”

Introduced in R2017a
augmentedImageDatastore

Transform batches to augment image data

Description

An augmented image datastore transforms batches of training, validation, test, and prediction data, with optional preprocessing such as resizing, rotation, and reflection. Resize images to make them compatible with the input size of your deep learning network. Augment training image data with randomized preprocessing operations to help prevent the network from overfitting and memorizing the exact details of the training images.

To train a network using augmented images, supply the augmentedImageDatastore to trainNetwork. For more information, see “Preprocess Images for Deep Learning”.

- When you use an augmented image datastore as a source of training images, the datastore randomly perturbs the training data for each epoch, so that each epoch uses a slightly different data set. The actual number of training images at each epoch does not change. The transformed images are not stored in memory.

- An imageInputLayer normalizes images using the mean of the augmented images, not the mean of the original data set. This mean is calculated once for the first augmented epoch. All other epochs use the same mean, so that the average image does not change during training.

By default, an augmentedImageDatastore only resizes images to fit the output size. You can configure options for additional image transformations using an imageDataAugmenter.

Creation

Syntax

auimds = augmentedImageDatastore(outputSize,imds)
auimds = augmentedImageDatastore(outputSize,X,Y)
auimds = augmentedImageDatastore(outputSize,X)
auimds = augmentedImageDatastore(outputSize,tbl)
auimds = augmentedImageDatastore(outputSize,tbl,responseName)
auimds = augmentedImageDatastore(____,Name,Value)

Description

auimds = augmentedImageDatastore(outputSize,imds) creates an augmented image datastore for classification problems using images from image datastore imds, and sets the OutputSize property.

auimds = augmentedImageDatastore(outputSize,X,Y) creates an augmented image datastore for classification and regression problems. The array X contains the predictor variables and the array Y contains the categorical labels or numeric responses.

auimds = augmentedImageDatastore(outputSize,X) creates an augmented image datastore for predicting responses of image data in array X.

auimds = augmentedImageDatastore(outputSize,tbl) creates an augmented image datastore for classification and regression problems. The table, tbl, contains predictors and responses.

auimds = augmentedImageDatastore(outputSize,tbl,responseName) creates an augmented image datastore for classification and regression problems. The table, tbl, contains predictors and responses. The responseName argument specifies the response variables in tbl.

auimds = augmentedImageDatastore(____,Name,Value) creates an augmented image datastore, using name-value pairs to set the ColorPreprocessing, DataAugmentation, OutputSizeMode, and DispatchInBackground properties. You can specify multiple name-value pairs. Enclose each property name in quotes.

For example,
augmentedImageDatastore([28,28],myTable,'OutputSizeMode','centercrop') creates an augmented image datastore that crops images from the center.

Input Arguments

imds — Image datastore
ImageDatastore object
Image datastore, specified as an ImageDatastore object.

ImageDatastore allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then ImageDatastore does not prefetch.

**Tip** Use augmentedImageDatastore for efficient preprocessing of images for deep learning including image resizing.

Do not use the readFcn option of imageDatastore as this option is usually significantly slower.

**X — Images**
4-D numeric array

Images, specified as a 4-D numeric array. The first three dimensions are the height, width, and channels, and the last dimension indexes the individual images.

If the array contains NaNs, then they are propagated through the training. However, in most cases, the training fails to converge.

Data Types: single | double | uint8 | int8 | uint16 | int16 | uint32 | int32

**Y — Responses for classification or regression**
array of categorical responses | numeric matrix | 4-D numeric array

Responses for classification or regression, specified as one of the following:

- For a classification problem, Y is a categorical vector containing the image labels.
- For a regression problem, Y can be an:
  - n-by-\(r\) numeric matrix. \(n\) is the number of observations and \(r\) is the number of responses.
  - h-by-w-by-c-by-\(n\) numeric array. h-by-w-by-c is the size of a single response and \(n\) is the number of observations.

Responses must not contain NaNs.

Data Types: categorical | double
Input data, specified as a table. `tbl` must contain the predictors in the first column as either absolute or relative image paths or images. The type and location of the responses depend on the problem:

- For a classification problem, the response must be a categorical variable containing labels for the images. If the name of the response variable is not specified in the call to `augmentedImageDatastore`, the responses must be in the second column. If the responses are in a different column of `tbl`, then you must specify the response variable name using the `responseName` positional argument.

- For a regression problem, the responses must be numerical values in the column or columns after the first one. The responses can be either in multiple columns as scalars or in a single column as numeric vectors or cell arrays containing numeric 3-D arrays. When you do not specify the name of the response variable or variables, `augmentedImageDatastore` accepts the remaining columns of `tbl` as the response variables. You can specify the response variable names using the `responseName` positional argument.

Responses must not contain NaNs. If there are NaNs in the predictor data, they are propagated through the training, however, in most cases the training fails to converge.

Data Types: `table`

`responseName` — Names of response variables in the input table

Character vector | Cell array of character vectors | String array

Names of the response variables in the input table, specified as a character vector, cell array of character vectors, or a string array. For problems with one response, `responseName` is the corresponding variable name in `tbl`. For regression problems with multiple response variables, `responseName` is an array of the corresponding variable names in `tbl`.

Data Types: `char` | `cell` | `string`
Preprocessing color operations performed on input grayscale or RGB images, specified as 'none', 'gray2rgb', or 'rgb2gray'. When the image datastore contains a mixture of grayscale and RGB images, use ColorPreprocessing to ensure that all output images have the number of channels required by imageInputLayer.

No color preprocessing operation is performed when an input image already has the required number of color channels. For example, if you specify the value 'gray2rgb' and an input image already has three channels, then no color preprocessing occurs.

**Note** The augmentedImageDatastore object converts RGB images to grayscale by using the rgb2gray function. If an image has three channels that do not correspond to red, green, and blue channels (such as an image in the L*a*b* color space), then using ColorPreprocessing can give poor results.

No color preprocessing operation is performed when the input images do not have 1 or 3 channels, such as for multispectral or hyperspectral images. In this case, all input images must have the same number of channels.

Data Types: char | string

**DataAugmentation — Preprocessing applied to input images**

'none' (default) | imageDataAugmenter object

Preprocessing applied to input images, specified as an imageDataAugmenter object or 'none'. When DataAugmentation is 'none', no preprocessing is applied to input images.

**DispatchInBackground — Dispatch observations in background**

false (default) | true

Dispatch observations in the background during training, prediction, or classification, specified as false or true. To use background dispatching, you must have Parallel Computing Toolbox.

Augmented image datastores only perform background dispatching when used with trainNetwork and inference functions such as predict and classify. Background dispatching does not occur when you call the read function of the datastore directly.

**MiniBatchSize — Number of observations in each batch**

128 | positive integer
Number of observations that are returned in each batch. You can change the value of MiniBatchSize only after you create the datastore. For training, prediction, and classification, the MiniBatchSize property is set to the mini-batch size defined in trainingOptions.

**NumObservations — Total number of observations in the datastore**

positive integer

This property is read-only.

Total number of observations in the augmented image datastore. The number of observations is the length of one training epoch.

**OutputSize — Size of output images**

vector of two positive integers

Size of output images, specified as a vector of two positive integers. The first element specifies the number of rows in the output images, and the second element specifies the number of columns.

**Note** If you create an augmentedImageDatastore by specifying the image output size as a three-element vector, then the datastore ignores the third element. Instead, the datastore uses the value of ColorPreprocessing to determine the dimensionality of output images. For example, if you specify OutputSize as [28 28 1] but set ColorPreprocessing as 'gray2rgb', then the output images have size 28-by-28-by-3.

**OutputSizeMode — Method used to resize output images**

'resize' (default) | 'centercrop' | 'randcrop'

Method used to resize output images, specified as one of the following.

- 'resize' — Scale the image using bilinear interpolation to fit the output size.

**Note** augmentedImageDatastore uses the bilinear interpolation method of imresize with antialiasing. Bilinear interpolation enables fast image processing while avoiding distortions such as caused by nearest-neighbor interpolation. In contrast, by default imresize uses bicubic interpolation with antialiasing to produce a high-quality resized image at the cost of longer processing time.
• 'centercrop' — Take a crop from the center of the training image. The crop has the same size as the output size.
• 'randcrop' — Take a random crop from the training image. The random crop has the same size as the output size.

Data Types: char | string

Object Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine</td>
<td>Combine data from multiple datastores</td>
</tr>
<tr>
<td>hasdata</td>
<td>Determine if data is available to read</td>
</tr>
<tr>
<td>partitionByIndex</td>
<td>Partition augmentedImageDatastore according to indices</td>
</tr>
<tr>
<td>preview</td>
<td>Subset of data in datastore</td>
</tr>
<tr>
<td>read</td>
<td>Read data from augmentedImageDatastore</td>
</tr>
<tr>
<td>readall</td>
<td>Read all data in datastore</td>
</tr>
<tr>
<td>readByIndex</td>
<td>Read data specified by index from augmentedImageDatastore</td>
</tr>
<tr>
<td>reset</td>
<td>Reset datastore to initial state</td>
</tr>
<tr>
<td>shuffle</td>
<td>Shuffle data in augmentedImageDatastore</td>
</tr>
<tr>
<td>transform</td>
<td>Transform datastore</td>
</tr>
</tbody>
</table>

Examples

Train Network with Augmented Images

Train a convolutional neural network using augmented image data. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

Load the sample data, which consists of synthetic images of handwritten digits.

[XTrain,YTrain] = digitTrain4DArrayData;

digitTrain4DArrayData loads the digit training set as 4-D array data. XTrain is a 28-by-28-by-1-by-5000 array, where:

• 28 is the height and width of the images.
• 1 is the number of channels.
• 5000 is the number of synthetic images of handwritten digits.

\( YTrain \) is a categorical vector containing the labels for each observation.

Set aside 1000 of the images for network validation.

```matlab
idx = randperm(size(XTrain,4),1000); XValidation = XTrain(:, :, :, idx); XTrain(:, :, :, idx) = []; YValidation = YTrain(idx); YTrain(idx) = [];
```

Create an `imageDataAugmenter` object that specifies preprocessing options for image augmentation, such as resizing, rotation, translation, and reflection. Randomly translate the images up to three pixels horizontally and vertically, and rotate the images with an angle up to 20 degrees.

```matlab
imageAugmenter = imageDataAugmenter( ... ’RandRotation’, [-20, 20], ... ’RandXTranslation’, [-3 3], ... ’RandYTranslation’, [-3 3])
```

Create an `augmentedImageDatastore` object to use for network training and specify the image output size. During training, the datastore performs image augmentation and resizes the images. The datastore augments the images without saving any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.
Specifying the convolutional neural network architecture:

```matlab
layers = [
    imageInputLayer(imageSize)
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Specify training options for stochastic gradient descent with momentum:

```matlab
opts = trainingOptions('sgdm', ...
    'MaxEpochs',15, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation});
```

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

```matlab
net = trainNetwork(augimds,layers,opts);
```
Tips

- You can visualize many transformed images in the same figure by using the `imtile` function. For example, this code displays one mini-batch of transformed images from an augmented image datastore called `auimds`. 
minibatch = read(auimds);
imshow(imtile(minibatch.input))

• By default, resizing is the only image preprocessing operation performed on images. Enable additional preprocessing operations by using the DataAugmentation name-value pair argument with an imageDataAugmenter object. Each time images are read from the augmented image datastore, a different random combination of preprocessing operations are applied to each image.

See Also
imageDataAugmenter | imageInputLayer | trainNetwork

Topics
“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”

Introduced in R2018a
read

Read data from augmentedImageDatastore

Syntax

data = read(auimds)
[data,info] = read(auimds)

Description

data = read(auimds) returns a batch of data from an augmented image datastore, auimds. Subsequent calls to the read function continue reading from the endpoint of the previous call.

[data,info] = read(auimds) also returns information about the extracted data, including metadata, in info.

Input Arguments

auimds — Augmented image datastore
augmentedImageDatastore

Augmented image datastore, specified as an augmentedImageDatastore object. The datastore specifies a MiniBatchSize number of observations in each batch, and a numObservations total number of observations.

Output Arguments

data — Output data
table

Output data, returned as a table with MiniBatchSize number of rows.
For the last batch of data in the datastore `auimds`, if `numObservations` is not cleanly divisible by `MiniBatchSize`, then `read` returns a partial batch containing all the remaining observations in the datastore.

**info — Information about read data**
structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename</td>
<td>Filename is a fully resolved path containing the path string, name of the file, and file extension.</td>
</tr>
<tr>
<td>FileSize</td>
<td>Total file size, in bytes. For MAT-files, <code>FileSize</code> is the total number of key-value pairs in the file.</td>
</tr>
</tbody>
</table>

**See Also**
`read (Datastore)` | `readByIndex` | `readall`

**Introduced in R2018a**
readByIndex

Read data specified by index from augmentedImageDatastore

Syntax

data = readByIndex(auimds,ind)
[data,info] = readByIndex(auimds,ind)

Description

data = readByIndex(auimds,ind) returns a subset of observations from an augmented image datastore, auimds. The desired observations are specified by indices, ind.

[data,info] = readByIndex(auimds,ind) also returns information about the observations, including metadata, in info.

Input Arguments

auimds — Augmented image datastore
augmentedImageDatastore

Augmented image datastore, specified as an augmentedImageDatastore object.

ind — Indices
vector of positive integers

Indices of observations, specified as a vector of positive integers.

Output Arguments

data — Observations from datastore
table
Observations from the datastore, returned as a table with \texttt{length(ind)} number of rows.

\texttt{info} — Information about read data

structure array

Information about read data, returned as a structure array with the following fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniBatchIndices</td>
<td>Numeric vector of indices.</td>
</tr>
</tbody>
</table>

\textbf{See Also}

\texttt{partitionByIndex} | \texttt{read} | \texttt{readall}

\textbf{Introduced in R2018a}
shuffle

Shuffle data in augmentedImageDatastore

Syntax

auimds2 = shuffle(auimds)

Description

auimds2 = shuffle(auimds) returns an augmentedImageDatastore object containing a random ordering of the data from augmented image datastore auimds.

Input Arguments

auimds — Augmented image datastore
augmentedImageDatastore

Augmented image datastore, specified as an augmentedImageDatastore object.

Output Arguments

auimds2 — Output datastore
augmentedImageDatastore object

Output datastore, returned as an augmentedImageDatastore object containing randomly ordered files from auimds.

See Also

read | readByIndex | readall
Introduced in R2018a
partitionByIndex

Partition augmentedImageDatastore according to indices

Syntax

auimds2 = partitionByIndex(auimds,ind)

Description

auimds2 = partitionByIndex(auimds,ind) partitions a subset of observations in an augmented image datastore, auimds, into a new datastore, auimds2. The desired observations are specified by indices, ind.

Input Arguments

auimds — Augmented image datastore
augmentedImageDatastore

Augmented image datastore, specified as an augmentedImageDatastore object.

ind — Indices
vector of positive integers

Indices of observations, specified as a vector of positive integers.

Output Arguments

auimds2 — Output datastore
augmentedImageDatastore object

Output datastore, returned as an augmentedImageDatastore object containing a subset of files from auimds.
See Also
read | readByIndex | readall

Introduced in R2018a
augmentedImageSource

(To be removed) Generate batches of augmented image data

**Note** augmentedImageSource will be removed in a future release. Create an augmented image datastore using the augmentedImageDatastore function instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
auimds = augmentedImageSource(outputSize,imds)
auimds = augmentedImageSource(outputSize,X,Y)
auimds = augmentedImageSource(outputSize,tbl)
auimds = augmentedImageSource(outputSize,tbl,responseName)
auimds = augmentedImageSource(____,Name,Value)
```

**Description**

`auimds = augmentedImageSource(outputSize,imds)` creates an augmented image datastore, `auimds`, for classification problems using images from image datastore `imds`, with output image size `outputSize`.

`auimds = augmentedImageSource(outputSize,X,Y)` creates an augmented image datastore for classification and regression problems. The array `X` contains the predictor variables and the array `Y` contains the categorical labels or numeric responses.

`auimds = augmentedImageSource(outputSize,tbl)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses.

`auimds = augmentedImageSource(outputSize,tbl,responseName)` creates an augmented image datastore for classification and regression problems. The table, `tbl`, contains predictors and responses. The `responseName` argument specifies the response variable in `tbl`. 
auimds = augmentedImageSource(___,Name,Value) creates an augmented image datastore, using name-value pairs to configure the image preprocessing done by the augmented image datastore. You can specify multiple name-value pairs.

Examples

Train Network with Rotational Invariance Using augmentedImageSource

Preprocess images using random rotation so that the trained convolutional neural network has rotational invariance. This example uses the augmentedImageSource function to create an augmented image datastore object. For an example of the recommended workflow that uses the augmentedImageDatastore function to create an augmented image datastore object, see “Train Network with Augmented Images” on page 1-799.

Load the sample data, which consists of synthetic images of handwritten numbers.

[XTrain,YTrain] = digitTrain4DArrayData;

digitTrain4DArrayData loads the digit training set as 4-D array data. XTrain is a 28-by-28-by-1-by-5000 array, where:

• 28 is the height and width of the images.
• 1 is the number of channels
• 5000 is the number of synthetic images of handwritten digits.

YTrain is a categorical vector containing the labels for each observation.

Create an image augmenter that rotates images during training. This image augmenter rotates each image by a random angle.

imageAugmenter = imageDataAugmenter('RandRotation',[-180 180])

imageAugmenter = imageDataAugmenter with properties:
    FillValue: 0
    RandXReflection: 0
    RandYReflection: 0
RandRotation: [-180 180]
RandScale: [1 1]
RandXScale: [1 1]
RandYScale: [1 1]
RandXShear: [0 0]
RandYShear: [0 0]
RandXTranslation: [0 0]
RandYTranslation: [0 0]

Use the `augmentedImageSource` function to create an augmented image datastore. Specify the size of augmented images, the training data, and the image augmenter.

```matlab
imageSize = [28 28 1];
auimds = augmentedImageSource(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter);
```

```matlab
auimds =
    augmentedImageDatastore with properties:
    NumObservations: 5000
    MiniBatchSize: 128
    DataAugmentation: [1x1 imageDataAugmenter]
    ColorPreprocessing: 'none'
        OutputSize: [28 28]
        OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

Specify the convolutional neural network architecture.

```matlab
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16,'Padding',1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding',1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
]```
convolution2dLayer(3, 64, 'Padding', 1)
batchNormalizationLayer
reluLayer

fullyConnectedLayer(10)
softmaxLayer
classificationLayer];

Set the training options for stochastic gradient descent with momentum.

opts = trainingOptions('sgdm', ...
'MaxEpochs', 10, ...
'Shuffle', 'every-epoch', ...
'InitialLearnRate', 1e-3);

Train the network.

net = trainNetwork(auimds, layers, opts);

Training on single CPU.
Initializing image normalization.

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Iteration</th>
<th>Time Elapsed (hh:mm:ss)</th>
<th>Mini-batch Accuracy</th>
<th>Mini-batch Loss</th>
<th>Base Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>00:00:01</td>
<td>7.81%</td>
<td>2.4151</td>
<td>0.0010</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>00:00:23</td>
<td>52.34%</td>
<td>1.4930</td>
<td>0.0010</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>00:00:44</td>
<td>74.22%</td>
<td>1.0148</td>
<td>0.0010</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>00:01:05</td>
<td>78.13%</td>
<td>0.8153</td>
<td>0.0010</td>
</tr>
<tr>
<td>6</td>
<td>200</td>
<td>00:01:26</td>
<td>76.56%</td>
<td>0.6903</td>
<td>0.0010</td>
</tr>
<tr>
<td>7</td>
<td>250</td>
<td>00:01:45</td>
<td>87.50%</td>
<td>0.4891</td>
<td>0.0010</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
<td>00:02:06</td>
<td>87.50%</td>
<td>0.4874</td>
<td>0.0010</td>
</tr>
<tr>
<td>9</td>
<td>350</td>
<td>00:02:30</td>
<td>87.50%</td>
<td>0.4866</td>
<td>0.0010</td>
</tr>
<tr>
<td>10</td>
<td>390</td>
<td>00:02:46</td>
<td>89.06%</td>
<td>0.4021</td>
<td>0.0010</td>
</tr>
</tbody>
</table>

Input Arguments

outputSize — Size of output images
vector of two positive integers

Size of output images, specified as a vector of two positive integers. The first element specifies the number of rows in the output images, and the second element specifies the
number of columns. This value sets the **OutputSize** on page 1-9 property of the returned augmented image datastore, **auimds**.

**imds — Image datastore**

*ImageDatastore* object

Image datastore, specified as an *ImageDatastore* object.

*ImageDatastore* allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then *ImageDatastore* does not prefetch.

**Tip** Use **augmentedImageDatastore** for efficient preprocessing of images for deep learning including image resizing.

Do not use the **readFcn** option of *ImageDatastore* as this option is usually significantly slower.

**X — Images**

*ImageDatastore* object

Images, specified as a 4-D numeric array. The first three dimensions are the height, width, and channels, and the last dimension indexes the individual images.

If the array contains **NaNs**, then they are propagated through the training. However, in most cases, the training fails to converge.

Data Types: **single** | **double** | **uint8** | **int8** | **uint16** | **int16** | **uint32** | **int32**

**Y — Responses for classification or regression**

array of categorical responses | numeric matrix | 4-D numeric array

Responses for classification or regression, specified as one of the following:

- For a classification problem, *Y* is a categorical vector containing the image labels.
- For a regression problem, *Y* can be an:
  - *n*-by-*r* numeric matrix. *n* is the number of observations and *r* is the number of responses.
  - *h*-by-*w*-by-*c*-by-*n* numeric array. *h*-by-*w*-by-*c* is the size of a single response and *n* is the number of observations.
Responses must not contain NaNs.

Data Types: categorical | double

**tbl — Input data**

table

Input data, specified as a table. `tbl` must contain the predictors in the first column as either absolute or relative image paths or images. The type and location of the responses depend on the problem:

- For a classification problem, the response must be a categorical variable containing labels for the images. If the name of the response variable is not specified in the call to `augmentedImageSource`, the responses must be in the second column. If the responses are in a different column of `tbl`, then you must specify the response variable name using the `responseName` positional argument.

- For a regression problem, the responses must be numerical values in the column or columns after the first one. The responses can be either in multiple columns as scalars or in a single column as numeric vectors or cell arrays containing numeric 3-D arrays. When you do not specify the name of the response variable or variables, `augmentedImageSource` accepts the remaining columns of `tbl` as the response variables. You can specify the response variable names using the `responseName` positional argument.

Responses must not contain NaNs. If there are NaNs in the predictor data, they are propagated through the training, however, in most cases the training fails to converge.

Data Types: table

**responseName — Names of response variables in the input table**

character vector | cell array of character vectors | string array

Names of the response variables in the input table, specified as a character vector, cell array of character vectors, or a string array. For problems with one response, `responseName` is the corresponding variable name in `tbl`. For regression problems with multiple response variables, `responseName` is an array of the corresponding variable names in `tbl`.

Data Types: char | cell | string
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:
augmentedImageSource([28,28], myTable, 'OutputSizeMode', 'centercrop')
creates an augmented image datastore that sets the OutputSizeMode property to crop images from the center.

ColorPreprocessing — Preprocessing color operations

'none' (default) | 'gray2rgb' | 'rgb2gray'

Preprocessing operations performed on color channels of input images, specified as the comma-separated pair consisting of 'ColorPreprocessing' and 'none', 'gray2rgb', or 'rgb2gray'. This argument sets the ColorPreprocessing on page 1-0 property of the returned augmented image datastore, auimds. The ColorPreprocessing property ensures that all output images from the augmented image datastore have the number of color channels required by inputImageLayer.

DataAugmentation — Preprocessing applied to input images

'none' (default) | imageDataAugmenter object

Preprocessing applied to input images, specified as the comma-separated pair consisting of 'DataAugmentation' and an imageDataAugmenter object or 'none'. This argument sets the DataAugmentation on page 1-0 property of the returned augmented image datastore, auimds. When DataAugmentation is 'none', no preprocessing is applied to input images.

OutputSizeMode — Method used to resize output images

'resize' (default) | 'centercrop' | 'randcrop'

Method used to resize output images, specified as the comma-separated pair consisting of 'OutputSizeMode' and one of the following. This argument sets the OutputSizeMode on page 1-0 property of the returned augmented image datastore, auimds.

- 'resize' — Scale the image to fit the output size. For more information, see imresize.
- 'centercrop' — Take a crop from the center of the training image. The crop has the same size as the output size.
• 'randcrop' — Take a random crop from the training image. The random crop has the same size as the output size.

Data Types: char | string

**BackgroundExecution — Perform augmentation in parallel**

false (default) | true

Perform augmentation in parallel, specified as the comma-separated pair consisting of 'BackgroundExecution' and false or true. This argument sets the DispatchInBackground on page 1-0 property of the returned augmented image datastore, auimds. If 'BackgroundExecution' is true, and you have Parallel Computing Toolbox software installed, then the augmented image datastore auimds performs image augmentation in parallel.

**Output Arguments**

**auimds — Augmented image datastore**

augmentedImageDatastore object

Augmented image datastore, returned as an augmentedImageDatastore object.

**Compatibility Considerations**

**augmentedImageSource object is removed**

In R2017b, you could create an augmentedImageSource object to preprocess images for training deep learning networks. Starting in R2018a, the augmentedImageSource object has been removed. Use an augmentedImageDatastore object instead.

An augmentedImageDatastore has additional properties and methods to assist with data preprocessing. Unlike augmentedImageSource, which could be used for training only, you can use an augmentedImageDatastore for both training and prediction.

To create an augmentedImageDatastore object, you can use either the augmentedImageDatastore function (recommended) or the augmentedImageSource function.
**augmentedImageSource function will be removed**

*Not recommended starting in R2018a*

The `augmentedImageSource` function will be removed in a future release. Create an `augmentedImageDatastore` using the `augmentedImageDatastore` function instead.

To update your code, change instances of the function name `augmentedImageSource` to `augmentedImageDatastore`. You do not need to change the input arguments.

**See Also**

`augmentedImageDatastore`

*Introduced in R2017b*
imageDataAugmenter

Configure image data augmentation

Description

An image data augmenter configures a set of preprocessing options for image augmentation, such as resizing, rotation, and reflection.

The imageDataAugmenter is used by an augmentedImageDatastore to generate batches of augmented images. For more information, see “Augment Images for Training with Random Geometric Transformations”.

Creation

Syntax

\[
\text{aug} = \text{imageDataAugmenter} \\
\text{aug} = \text{imageDataAugmenter(Name,Value)}
\]

Description

\( \text{aug} = \text{imageDataAugmenter} \) creates an imageDataAugmenter object with default property values consistent with the identity transformation.

\( \text{aug} = \text{imageDataAugmenter(Name,Value)} \) configures a set of image augmentation options using name-value pairs to set properties on page 1-821. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

**FillValue** — Fill value

numeric scalar | numeric vector
Fill value used to define out-of-bounds points when resampling, specified as a numeric scalar or numeric vector.

- If the augmented images are single channel, then FillValue must be a scalar.
- If the augmented images are multichannel, then FillValue can be a scalar or a vector with length equal to the number of channels of the input image. For example, if the input image is an RGB image, FillValue can be a vector of length 3.

For grayscale and color images, the default fill value is 0. For categorical images, the default fill value is an '<undefined>' label and trainNetwork ignores filled pixels when training.

Example: 128

**RandXReflection — Random reflection**
false (default) | true

Random reflection in the left-right direction, specified as a logical scalar. When RandXReflection is true (1), each image is reflected horizontally with 50% probability. When RandXReflection is false (0), no images are reflected.

**RandYReflection — Random reflection**
false (default) | true

Random reflection in the top-bottom direction, specified as a logical scalar. When RandYReflection is true (1), each image is reflected vertically with 50% probability. When RandYReflection is false (0), no images are reflected.

**RandRotation — Range of rotation**
[0 0] (default) | 2-element numeric vector | function handle

Range of rotation, in degrees, applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The rotation angle is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the rotation angle as a numeric scalar. Use a function handle to pick rotation angles from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).
By default, augmented images are not rotated.

Example: [-45  45]

**RandScale — Range of uniform scaling**

[1  1] (default) | 2-element numeric vector | function handle

Range of uniform (isotropic) scaling applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The scale factor is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the scale factor as a numeric scalar. Use a function handle to pick scale factors from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not scaled.

Example: [0.5  4]

**RandXScale — Range of horizontal scaling**

[1  1] (default) | 2-element vector of positive numbers | function handle

Range of horizontal scaling applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The horizontal scale factor is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the horizontal scale factor as a numeric scalar. Use a function handle to pick horizontal scale factors from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not scaled in the horizontal direction.

**Note** If you specify RandScale, then imageDataAugmenter ignores the value of RandXScale when scaling images.
Example: [0.5 4]

**RandYScale — Range of vertical scaling**

[1 1] (default) | 2-element vector of positive numbers | function handle

Range of vertical scaling applied to the input image, specified as one of the following.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The vertical scale factor is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the vertical scale factor as a numeric scalar. Use a function handle to pick vertical scale factors from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not scaled in the vertical direction.

**Note** If you specify RandScale, then imageDataAugmenter ignores the value of RandYScale when scaling images.

Example: [0.5 4]

**RandXShear — Range of horizontal shear**

[0 0] (default) | 2-element numeric vector | function handle

Range of horizontal shear applied to the input image, specified as one of the following. Shear is measured as an angle in degrees, and is in the range (–90, 90).

- 2-element numeric vector. The second element must be larger than or equal to the first element. The horizontal shear angle is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the horizontal shear angle as a numeric scalar. Use a function handle to pick horizontal shear angles from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not sheared in the horizontal direction.

Example: [0 45]
RandYShear — Range of vertical shear
[0 0] (default) | 2-element numeric vector | function handle

Range of vertical shear applied to the input image, specified as one of the following. Shear is measured as an angle in degrees, and is in the range (–90, 90).

- 2-element numeric vector. The second element must be larger than or equal to the first element. The vertical shear angle is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the vertical shear angle as a numeric scalar. Use a function handle to pick vertical shear angles from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not sheared in the vertical direction.

Example: [0 45]

RandXTranslation — Range of horizontal translation
[0 0] (default) | 2-element numeric vector | function handle

Range of horizontal translation applied to the input image, specified as one of the following. Translation distance is measured in pixels.

- 2-element numeric vector. The second element must be larger than or equal to the first element. The horizontal translation distance is picked randomly from a continuous uniform distribution within the specified interval.
- function handle. The function must accept no input arguments and return the horizontal translation distance as a numeric scalar. Use a function handle to pick horizontal translation distances from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not translated in the horizontal direction.

Example: [-5 5]

RandYTranslation — Range of vertical translation
[0 0] (default) | 2-element numeric vector | function handle

Range of vertical translation applied to the input image, specified as one of the following. Translation distance is measured in pixels.
• 2-element numeric vector. The second element must be larger than or equal to the first element. The vertical translation distance is picked randomly from a continuous uniform distribution within the specified interval.

• function handle. The function must accept no input arguments and return the vertical translation distance as a numeric scalar. Use a function handle to pick vertical translation distances from a disjoint interval or using a nonuniform probability distribution. For more information about function handles, see “Create Function Handle” (MATLAB).

By default, augmented images are not translated in the vertical direction.

Example: [-5 5]

### Object Functions

**augment**  Apply identical random transformations to multiple images

### Examples

#### Create Image Data Augmenter to Resize and Rotate Images

Create an image data augmenter that preprocesses images before training. This augmenter rotates images by random angles in the range [0, 360] degrees and resizes images by random scale factors in the range [0.5, 1].

```matlab
augmenter = imageDataAugmenter( ...  
    'RandRotation', [0 360], ...  
    'RandScale', [0.5 1])
```

```matlab
augmenter =  
    imageDataAugmenter with properties:  
        FillValue: 0  
    RandXReflection: 0  
    RandYReflection: 0  
    RandRotation: [0 360]  
    RandScale: [0.5000 1]  
    RandXScale: [1 1]  
    RandYScale: [1 1]  
    RandXShear: [0 0]
```
Create an augmented image datastore using the image data augmenter. The augmented image datastore also requires sample data, labels, and an output image size.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
imageSize = [56 56 1];
auidms = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',augmenter)
```

```
auimds =
    augmentedImageDatastore with properties:

        NumObservations: 5000
        MiniBatchSize: 128
        DataAugmentation: [1x1 imageDataAugmenter]
        ColorPreprocessing: 'none'
            OutputSize: [56 56]
            OutputSizeMode: 'resize'
        DispatchInBackground: 0
```

Preview the random transformations applied to the first eight images in the image datastore.

```matlab
minibatch = preview(auimds);
imshow(imtile(minibatch.input));
```
Preview different random transformations applied to the same set of images.

```matlab
minibatch = preview(auimds);
imshow(imtile(minibatch.input));
```
Train Network with Augmented Images

Train a convolutional neural network using augmented image data. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

Load the sample data, which consists of synthetic images of handwritten digits.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
```

digitTrain4DArrayData loads the digit training set as 4-D array data. XTrain is a 28-by-28-by-1-by-5000 array, where:

- 28 is the height and width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

YTrain is a categorical vector containing the labels for each observation.

Set aside 1000 of the images for network validation.

```matlab
idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,,:,idx);
XTrain(:,:,,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

Create an imageDataAugmenter object that specifies preprocessing options for image augmentation, such as resizing, rotation, translation, and reflection. Randomly translate the images up to three pixels horizontally and vertically, and rotate the images with an angle up to 20 degrees.

```matlab
imageAugmenter = imageDataAugmenter( ... 
    'RandRotation',[-20,20], ... 
    'RandXTranslation',[-3 3], ... 
    'RandYTranslation',[-3 3])
```

imageAugmenter = imageDataAugmenter with properties:

- FillValue: 0
- RandXReflection: 0
- RandYReflection: 0
Create an augmentedImageDatastore object to use for network training and specify the image output size. During training, the datastore performs image augmentation and resizes the images. The datastore augments the images without saving any images to memory. trainNetwork updates the network parameters and then discards the augmented images.

```plaintext
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter);
```

Specify the convolutional neural network architecture.

```plaintext
layers = [
    imageInputLayer(imageSize)
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Specify training options for stochastic gradient descent with momentum.
opts = trainingOptions('sgdm', ...
'MaxEpochs',15, ...
'Shuffle','every-epoch', ...
'Plots','training-progress', ...
'Verbose',false, ...
'ValidationData',{XValidation,YValidation});

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

net = trainNetwork(augimds,layers,opts);
**Tips**

- To preview the transformations applied to sample images, use the `augment` function.
- To perform image augmentation during training, create an `augmentedImageDatastore` and specify preprocessing options by using the `'DataAugmentation'` name-value pair with an `ImageDataAugmenter`. The
augmented image datastore automatically applies random transformations to the training data.

**See Also**
augmentedImageDatastore | imageInputLayer | trainNetwork

**Topics**
“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”
“Create Function Handle” (MATLAB)

**Introduced in R2017b**
augment

Apply identical random transformations to multiple images

Syntax

augI = augment(augmenter,I)

Description

augI = augment(augmenter,I) augments image I using a random transformation from the set of image preprocessing options defined by image data augmenter, augmenter. If I consists of multiple images, then augment applies an identical transformation to all images.

Examples

Augment Image Data with Custom Rotation Range

Create an image augmenter that rotates images by a random angle. To use a custom range of valid rotation angles, you can specify a function handle when you create the augmenter. This example specifies a function called myrange (defined at the end of the example) that selects an angle from within two disjoint intervals.

imageAugmenter = imageDataAugmenter('RandRotation',@myrange);

Read multiple images into the workspace, and display the images.

img1 = imread('peppers.png');
img2 = imread('corn.tif',2);
inImg = imtile({img1,img2});
imshow(inImg)
Augment the images with identical augmentations. The randomly selected rotation angle is returned in a temporary variable, `angle`.

```matlab
outCellArray = augment(imageAugmenter,{img1,img2});
```

`angle = 8.1158`

View the augmented images.

```matlab
outImg = imtile(outCellArray); imshow(outImg);
```

1-835
Supporting Function

This example defines the myrange function that first randomly selects one of two intervals (-10, 10) and (170, 190) with equal probability. Within the selected interval, the function returns a single random number from a uniform distribution.

```matlab
function angle = myrange()
    if randi([0 1],1)
        a = -10;
        b = 10;
    else
        a = 170;
        b = 190;
    end
    angle = a + (b-a).*rand(1)
end
```

Input Arguments

**augmenter — Augmentation options**

`imageDataAugmenter` object

Augmentation options, specified as an `imageDataAugmenter` object.
**I — Images to augment**
numeric array | cell array of numeric and categorical images

Images to augment, specified as one of the following.

- Numeric array, representing a single grayscale or color image.
- Cell array of numeric and categorical images. Images can be different sizes and types.

**Output Arguments**

**augI — Augmented images**
numeric array | cell array of numeric and categorical images

Augmented images, returned as a numeric array or cell array of numeric and categorical images, consistent with the format of the input images I.

**Tips**

- You can use the `augment` function to preview the transformations applied to sample images.
- To perform image augmentation during training, create an `augmentedImageDatastore` and specify preprocessing options by using the 'DataAugmentation' name-value pair with an `imageDataAugmenter`. The augmented image datastore automatically applies random transformations to the training data.

**See Also**

`augmentedImageDatastore` | `trainNetwork`

**Topics**

“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”

Introduced in R2018b
averagePooling2dLayer

Average pooling layer

Description

An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.

Creation

Syntax

layer = averagePooling2dLayer(poolSize)
layer = averagePooling2dLayer(poolSize,Name,Value)

Description

layer = averagePooling2dLayer(poolSize) creates an average pooling layer and sets the PoolSize property.

layer = averagePooling2dLayer(poolSize,Name,Value) sets the optional Stride and Name properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example, averagePooling2dLayer(2,'Stride',2) creates an average pooling layer with pool size [2 2] and stride [2 2]. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the Stride and Name properties. Enclose names in single quotes.

Example: averagePooling2dLayer(2,'Stride',2) creates an average pooling layer with pool size [2 2] and stride [2 2].

Padding — Input edge padding
[0 0 0 0] (default) | vector of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

- 'same' — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is ceil(inputSize/stride), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.
- Nonnegative integer p — Add padding of size p to all the edges of the input.
- Vector [a b] of nonnegative integers — Add padding of size a to the top and bottom of the input and padding of size b to the left and right.
- Vector [t b l r] of nonnegative integers — Add padding of size t to the top, b to the bottom, l to the left, and r to the right of the input.

Example: 'Padding',1 adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: 'Padding','same' adds padding so that the output has the same size as the input (if the stride equals 1).
Properties

Average Pooling

**PoolSize — Dimensions of pooling regions**
vector of two positive integers

Dimensions of the pooling regions, specified as a vector of two positive integers \([h\ w]\), where \(h\) is the height and \(w\) is the width. When creating the layer, you can specify `PoolSize` as a scalar to use the same value for both dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: \([2\ 1]\) specifies pooling regions of height 2 and width 1.

**Stride — Step size for traversing input**
\([1\ 1]\) (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector of two positive integers \([a\ b]\), where \(a\) is the vertical step size and \(b\) is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both dimensions.

If the stride dimensions `Stride` are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions `PaddingSize` must be less than the pooling region dimensions `PoolSize`.

Example: \([2\ 3]\) specifies a vertical step size of 2 and a horizontal step size of 3.

**PaddingSize — Size of padding**
\([0\ 0\ 0\ 0]\) (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector \([t\ b\ l\ r]\) of four nonnegative integers, where \(t\) is the padding applied to the top, \(b\) is the padding applied to the bottom, \(l\) is the padding applied to the left, and \(r\) is the padding applied to the right.
When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: [1 1 2 2] adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

**PaddingMode — Method to determine padding size**

'manual' (default) | 'same'

Method to determine padding size, specified as 'manual' or 'same'.

The software automatically sets the value of PaddingMode based on the 'Padding' value you specify when creating a layer.

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets PaddingMode to 'manual'.
- If you set the 'Padding' option to 'same', then the software automatically sets PaddingMode to 'same' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is \( \text{ceil}(\text{inputSize} / \text{stride}) \), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

**Padding — Size of padding**

[0 0] (default) | vector of two nonnegative integers

**Note** Padding property will be removed in a future release. Use PaddingSize instead.

When creating a layer, use the 'Padding' name-value pair argument to specify the padding size.

Size of padding to apply to input borders vertically and horizontally, specified as a vector [a b] of two nonnegative integers, where a is the padding applied to the top and bottom of the input data and b is the padding applied to the left and right.

Example: [1 1] adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.
Layer

Name — Layer name
' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

NumInputs — Number of inputs
1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

InputNames — Input names
{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

NumOutputs — Number of outputs
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

OutputNames — Output names
{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

Examples
Create Average Pooling Layer

Create an average pooling layer with the name 'avg1'.

```matlab
layer = averagePooling2dLayer(2,'Name','avg1')
layer = 
  AveragePooling2DLayer with properties:

  Name: 'avg1'

  Hyperparameters
  PoolSize: [2 2]
  Stride: [1 1]
  PaddingMode: 'manual'
  PaddingSize: [0 0 0 0]
```

Include an average pooling layer in a Layer array.

```matlab
layers = [ ... 
  imageInputLayer([28 28 1])
  convolution2dLayer(5,20)
  reluLayer
  averagePooling2dLayer(2)
  fullyConnectedLayer(10)
  softmaxLayer
  classificationLayer]

layers = 
  7x1 Layer array with layers:

  1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
  2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding
  3   ''   ReLU                    ReLU
  4   ''   Average Pooling         2x2 average pooling with stride [1  1] and padding
  5   ''   Fully Connected         10 fully connected layer
  6   ''   Softmax                 softmax
  7   ''   Classification Output   crossentropyex
```

Create Average Pooling Layer with Nonoverlapping Pooling Regions

Create an average pooling layer with nonoverlapping pooling regions.
layer = averagePooling2dLayer(2,'Stride',2)

layer =
    AveragePooling2DLayer with properties:

    Name: ''

    Hyperparameters
    PoolSize: [2 2]
    Stride: [2 2]
    PaddingMode: 'manual'
    PaddingSize: [0 0 0 0]

The height and width of the rectangular regions (pool size) are both 2. The pooling regions do not overlap because the step size for traversing the images vertically and horizontally (stride) is also 2.

Include an average pooling layer with nonoverlapping regions in a Layer array.

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    averagePooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

layers =
  7x1 Layer array with layers:

    1  ''  Image Input  28x28x1 images with 'zerocenter' normalization
    2  ''  Convolution  20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
    3  ''  ReLU        ReLU
    4  ''  Average Pooling 2x2 average pooling with stride [2 2] and padding [0 0 0 0]
    5  ''  Fully Connected 10 fully connected layer
    6  ''  Softmax      softmax
    7  ''  Classification Output  crossentropyex

Create Average Pooling Layer with Overlapping Pooling Regions

Create an average pooling layer with overlapping pooling regions.
layer = averagePooling2dLayer([3 2], 'Stride', 2)

layer =
    AveragePooling2DLayer with properties:

        Name: ''

    Hyperparameters
        PoolSize: 3x2
        Stride: 2x2
        PaddingMode: 'manual'
        PaddingSize: [0 0 0 0]

This layer creates pooling regions of size 3x2 and takes the average of the six elements in each region. The pooling regions overlap because Stride includes dimensions that are less than the respective pooling dimensions PoolSize.

Include an average pooling layer with overlapping pooling regions in a Layer array.

layers = [
    ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5, 20)
    reluLayer
    averagePooling2dLayer([3 2], 'Stride', 2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

layers =
    7x1 Layer array with layers:

    1   ''   Image Input       28x28x1 images with 'zerocenter' normalization
    2   ''   Convolution       20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
    3   ''   ReLU              ReLU
    4   ''   Average Pooling   3x2 average pooling with stride [2 2] and padding [0 0 0 0]
    5   ''   Fully Connected   10 fully connected layer
    6   ''   Softmax           softmax
    7   ''   Classification Output crossentropyex
More About

Average Pooling Layer

An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.

Pooling layers follow the convolutional layers for down-sampling, hence, reducing the number of connections to the following layers. They do not perform any learning themselves, but reduce the number of parameters to be learned in the following layers. They also help reduce overfitting.

An average pooling layer outputs the average values of rectangular regions of its input. The size of the rectangular regions is determined by the poolSize argument of averagePoolingLayer. For example, if poolSize is [2,3], then the layer returns the average value of regions of height 2 and width 3.

Pooling layers scan through the input horizontally and vertically in step sizes you can specify using the 'Stride' name-value pair argument. If the pool size is smaller than or equal to the stride, then the pooling regions do not overlap.

For nonoverlapping regions (Pool Size and Stride are equal), if the input to the pooling layer is n-by-n, and the pooling region size is h-by-h, then the pooling layer down-samples the regions by h [1]. That is, the output of a max or average pooling layer for one channel of a convolutional layer is n/h-by-n/h. For overlapping regions, the output of a pooling layer is (Input Size – Pool Size + 2*Padding)/Stride + 1.

References

Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
convolution2dLayer | globalAveragePooling2dLayer | maxPooling2dLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a
averagePooling3dLayer

3-D average pooling layer

Description

A 3-D average pooling layer performs down-sampling by dividing three-dimensional input into cuboidal pooling regions and computing the average values of each region.

Creation

Syntax

layer = averagePooling3dLayer(poolSize)
layer = averagePooling3dLayer(poolSize,Name,Value)

Description

layer = averagePooling3dLayer(poolSize) creates an average pooling layer and sets the PoolSize property.

layer = averagePooling3dLayer(poolSize,Name,Value) sets the optional Stride and Name properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example,

averagePooling3dLayer(2,'Stride',2) creates a 3-D average pooling layer with pool size [2 2 2] and stride [2 2 2]. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the Stride and Name properties. Enclose names in single quotes.

Example: `averagePooling3dLayer(2,'Stride',2)` creates a 3-D average pooling layer with pool size `[2 2 2]` and stride `[2 2 2].`

Padding — Input edge padding

0 (default) | array of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

• 'same' — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

• Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.

• Three-element vector `[a b c]` of nonnegative integers — Add padding of size `a` to the top and bottom, padding of size `b` to the left and right, and padding of size `c` to the front and back of the input.

• 2-by-3 matrix `[t l f; b r k]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, `r` to the right, `f` to the front, and `k` to the back of the input. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

Example: 'Padding',1 adds one row of padding to the top and bottom, one column of padding to the left and right, and one plane of padding to the front and back of the input.

Example: 'Padding','same' adds padding so that the output has the same size as the input (if the stride equals 1).
Properties

Average Pooling

PoolSize — Dimensions of pooling regions
vector of three positive integers

Dimensions of the pooling regions, specified as a vector of three positive integers \([h \ w \ d]\), where \(h\) is the height, \(w\) is the width, and \(d\) is the depth. When creating the layer, you can specify PoolSize as a scalar to use the same value for all three dimensions.

If the stride dimensions Stride are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions PaddingSize must be less than the pooling region dimensions PoolSize.

Example: \([2 \ 1 \ 1]\) specifies pooling regions of height 2, width 1, and depth 1.

Stride — Step size for traversing input

\([1 \ 1 \ 1]\) (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector \([a \ b \ c]\) of three positive integers, where \(a\) is the vertical step size, \(b\) is the horizontal step size, and \(c\) is the step size along the depth direction. When creating the layer, you can specify Stride as a scalar to use the same value for step sizes in all three directions.

If the stride dimensions Stride are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions PaddingSize must be less than the pooling region dimensions PoolSize.

Example: \([2 \ 3 \ 1]\) specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

PaddingSize — Size of padding

\([0 \ 0 \ 0;0 \ 0 \ 0]\) (default) | 2-by-3 matrix of nonnegative integers

Size of padding to apply to input borders, specified as 2-by-3 matrix \([t \ l \ f;b \ r \ k]\) of nonnegative integers, where \(t\) and \(b\) are the padding applied to the top and bottom in the vertical direction, \(l\) and \(r\) are the padding applied to the left and right in the horizontal
direction, and f and k are the padding applied to the front and back along the depth. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: [1 2 4;1 2 4] adds one row of padding to the top and bottom, two columns of padding to the left and right, and four planes of padding to the front and back of the input.

PaddingMode — Method to determine padding size

'manual' (default) | 'same'

Method to determine padding size, specified as 'manual' or 'same'.

The software automatically sets the value of PaddingMode based on the 'Padding' value you specify when creating a layer:

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets PaddingMode to 'manual'.
- If you set the 'Padding' option to 'same', then the software automatically sets PaddingMode to 'same' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is ceil(inputSize/stride), where inputSize is the height, width, or depth of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

Layer

Name — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.
Data Types: char | string

**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**
{'in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create 3-D Average Pooling Layer**

Create a 3-D average pooling layer with nonoverlapping pooling regions that downsamples by a factor of 2.

```matlab
layer = averagePooling3dLayer(2,'Stride',2)
```

layer =

```
AveragePooling3DLayer with properties:
```
Include a 3-D average pooling layer in a Layer array.

```matlab
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    averagePooling3dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

Create 3-D Average Pooling Layer with Overlapping Pooling Regions

Create a 3-D average pooling layer with overlapping pooling regions and padding for the top and bottom of the input.

```matlab
layer = averagePooling3dLayer([3 2 2],'Stride',2,'Padding',[1 0 0])
```

```matlab
layer =
    AveragePooling3DLayer with properties:
        Name: ''
```
Hyperparameters

- PoolSize: [3 2 2]
- Stride: [2 2 2]
- PaddingMode: 'manual'
- PaddingSize: [2x3 double]

This layer creates pooling regions of size 3-by-2-by-2 and takes the average of the twelve elements in each region. The stride is 2 in all dimensions. The pooling regions overlap because there are stride dimensions Stride that are less than the respective pooling dimensions PoolSize.

More About

3-D Average Pooling Layer

A 3-D average pooling layer extends the functionality of an average pooling layer to a third dimension, depth. An average pooling layer performs down-sampling by dividing the input into rectangular or cuboidal pooling regions, and computing the average of each region. To learn more, see the definition of average pooling layer on page 1-846 on the averagePooling2dLayer reference page.

See Also

averagePooling2dLayer | convolution3dLayer | maxPooling3dLayer

Topics

- “Deep Learning in MATLAB”
- “Specify Layers of Convolutional Neural Network”
- “List of Deep Learning Layers”

Introduced in R2019a
globalAveragePooling2dLayer

Global average pooling layer

Description

A global average pooling layer performs down-sampling by computing the mean of the height and width dimensions of the input.

Creation

Syntax

layer = globalAveragePooling2dLayer
layer = globalAveragePooling2dLayer('Name',name)

Description

layer = globalAveragePooling2dLayer creates a global average pooling layer.

layer = globalAveragePooling2dLayer('Name',name) sets the optional Name property.

Properties

Name — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string
**NumInputs — Number of inputs**  
1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**  
{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**  
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**  
{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

### Examples

**Create Global Average Pooling Layer**

Create a global average pooling layer with the name 'gap1'.

```matlab
layer = globalAveragePooling2dLayer('Name','gap1')
```

```markdown
layer =  
    GlobalAveragePooling2DLayer with properties:  

    Name: 'gap1'
```

Include a global average pooling layer in a Layer array.
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    globalAveragePooling2dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

layers =
7x1 Layer array with layers:

1 '' Image Input          28x28x1 images with 'zerocenter' normalization
2 '' Convolution         20 5x5 convolutions with stride [1  1] and padding [0  0  0  0]
3 '' ReLU                 ReLU
4 '' Global Average Pooling Global average pooling
5 '' Fully Connected      10 fully connected layer
6 '' Softmax              softmax
7 '' Classification Output crossentropyex

See Also
averagePooling2dLayer | convolution2dLayer |
globalAveragePooling3dLayer | maxPooling2dLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019b
globalAveragePooling3dLayer

3-D global average pooling layer

Description

A global average pooling layer performs down-sampling by computing the mean of the height, width, and depth dimensions of the input.

Creation

Syntax

layer = globalAveragePooling3dLayer
layer = globalAveragePooling3dLayer('Name',name)

Description

layer = globalAveragePooling3dLayer creates a 3-D global average pooling layer.

layer = globalAveragePooling3dLayer('Name',name) sets the optional Name property.

Properties

Name — Layer name

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string
**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**
{'in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create 3-D Global Average Pooling Layer**

Create a 3-D global average pooling layer with name 'gap1'.

```matlab
layer = globalAveragePooling3dLayer('Name','gap1')
```

Include a 3-D average pooling layer in a `Layer` array.
layers = [...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    globalAveragePooling3dLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
]

layers =
7x1 Layer array with layers:

1 ' ' 3-D Image Input  28x28x28x3 images with 'zerocenter' normalization
2 ' '  Convolution  20 5x5x5 convolutions with stride [1 1 1]
3 ' '   ReLU  ReLU
4 ' ' 3-D Global Average Pooling  3-D global average pooling
5 ' '  Fully Connected  10 fully connected layer
6 ' '   Softmax  softmax
7 ' ' Classification Output  crossentropyex

See Also
averagePooling3dLayer | convolution3dLayer |
globalAveragePooling2dLayer | maxPooling3dLayer

Topics
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019b
ClassificationOutputLayer

Classification layer

Description

A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.

Creation

Create a classification layer using `classificationLayer`.

Properties

Classification Output

Classes — Classes of the output layer

'auto' (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If `Classes` is 'auto', then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str,str)`. The default value is 'auto'.

Data Types: `char` | `categorical` | `string` | `cell`

OutputSize — Size of the output

'auto' (default) | positive integer

This property is read-only.

Size of the output, specified as a positive integer. This value is the number of labels in the data. Before the training, the output size is set to 'auto'.

**LossFunction — Loss function for training**

'\textit{crossentropyex}'

This property is read-only.

Loss function for training, specified as '\textit{crossentropyex}', which stands for \textit{Cross Entropy Function for k Mutually Exclusive Classes}.

**Layer**

**Name — Layer name**

'\textit{'}' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and \texttt{Name} is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

\{\texttt{'in'}\} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

0 (default)

Number of outputs of the layer. The layer has no outputs.

Data Types: double

**OutputNames — Output names**

\{\} (default)
Output names of the layer. The layer has no outputs.

Data Types: cell

**Examples**

**Create Classification Layer**

Create a classification layer with the name 'output'.

```matlab
layer = classificationLayer('Name','output')
```

```
layer = ClassificationOutputLayer with properties:
    Name: 'output'
    Classes: 'auto'
    OutputSize: 'auto'

Hyperparameters
    LossFunction: 'crossentropyex'
```

Include a classification output layer in a Layer array.

```matlab
layers = [ ... 
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers = 
    7x1 Layer array with layers:
1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
2   ''   Convolution             20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
3   ''   ReLU                    ReLU
4   ''   Max Pooling             2x2 max pooling with stride [2 2] and padding [0 0 0 0]
5   ''   Fully Connected         10 fully connected layer
6   ''   Softmax                 softmax
7   ''   Classification Output   crossentropyex
```
More About

Classification Output Layer

A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.

For typical classification networks, the classification layer must follow the softmax layer. In the classification layer, `trainNetwork` takes the values from the softmax function and assigns each input to one of the $K$ mutually exclusive classes using the cross entropy function for a 1-of-$K$ coding scheme [1]:

$$\text{loss} = - \sum_{i=1}^{N} \sum_{j=1}^{K} t_{ij} \ln y_{ij},$$

where $N$ is the number of samples, $K$ is the number of classes, $t_{ij}$ is the indicator that the $i$th sample belongs to the $j$th class, and $y_{ij}$ is the output for sample $i$ for class $j$, which in this case, is the value from the softmax function. That is, it is the probability that the network associates the $i$th input with class $j$.

Compatibility Considerations

**ClassNames property will be removed**

*Not recommended starting in R2018b*

ClassNames will be removed. Use `Classes` instead. To update your code, replace all instances of `ClassNames` with `Classes`. There are some differences between the properties that require additional updates to your code.

The `ClassNames` property of the output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.
References


See Also
regressionLayer | softmaxLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a
convolution2dLayer

2-D convolutional layer

Description

A 2-D convolutional layer applies sliding convolutional filters to the input. The layer convolves the input by moving the filters along the input vertically and horizontally and computing the dot product of the weights and the input, and then adding a bias term.

Creation

Syntax

layer = convolution2dLayer(filterSize,numFilters)
layer = convolution2dLayer(filterSize,numFilters,Name,Value)

Description

layer = convolution2dLayer(filterSize,numFilters) creates a 2-D convolutional layer and sets the FilterSize and NumFilters properties.

layer = convolution2dLayer(filterSize,numFilters,Name,Value) sets the optional Stride, DilationFactor, NumChannels, “Parameters and Initialization” on page 1-870, “Learn Rate and Regularization” on page 1-872, and Name properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example, convolution2dLayer(11,96,'Stride',4,'Padding',1) creates a 2-D convolutional layer with 96 filters of size [11 11], a stride of [4 4], and zero padding of size 1 along all edges of the layer input. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the Stride, DilationFactor, NumChannels, “Parameters and Initialization” on page 1-870, “Learn Rate and Regularization” on page 1-872, and Name properties. Enclose names in single quotes.

Example: convolution2dLayer(3,16,'Padding','same') creates a 2-D convolutional layer with 16 filters of size [3 3] and 'same' padding. At training time, the software calculates and sets the size of the zero padding so that the layer output has the same size as the input.

Padding — Input edge padding
[0 0 0 0] (default) | vector of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

- 'same' — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is ceil(inputSize/stride), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

- Nonnegative integer p — Add padding of size p to all the edges of the input.

- Vector [a b] of nonnegative integers — Add padding of size a to the top and bottom of the input and padding of size b to the left and right.

- Vector [t b l r] of nonnegative integers — Add padding of size t to the top, b to the bottom, l to the left, and r to the right of the input.

Example: 'Padding',1 adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: 'Padding','same' adds padding so that the output has the same size as the input (if the stride equals 1).
Properties

Convolution

FilterSize — Height and width of filters
vector of two positive integers

Height and width of the filters, specified as a vector \([h \ w]\) of two positive integers, where \(h\) is the height and \(w\) is the width. FilterSize defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify FilterSize as a scalar to use the same value for the height and width.

Example: \([5 \ 5]\) specifies filters with a height of 5 and a width of 5.

NumFilters — Number of filters
positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96

Stride — Step size for traversing input
\([1 \ 1]\) (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector \([a \ b]\) of two positive integers, where \(a\) is the vertical step size and \(b\) is the horizontal step size. When creating the layer, you can specify Stride as a scalar to use the same value for both step sizes.

Example: \([2 \ 3]\) specifies a vertical step size of 2 and a horizontal step size of 3.

DilationFactor — Factor for dilated convolution
\([1 \ 1]\) (default) | vector of two positive integers

Factor for dilated convolution (also known as atrous convolution), specified as a vector \([h \ w]\) of two positive integers, where \(h\) is the vertical dilation and \(w\) is the horizontal dilation. When creating the layer, you can specify DilationFactor as a scalar to use the same value for both horizontal and vertical dilations.
Use dilated convolutions to increase the receptive field (the area of the input which the layer can see) of the layer without increasing the number of parameters or computation.

The layer expands the filters by inserting zeros between each filter element. The dilation factor determines the step size for sampling the input or equivalently the upsampling factor of the filter. It corresponds to an effective filter size of \((\text{Filter Size} - 1) \times \text{Dilation Factor} + 1\). For example, a 3-by-3 filter with the dilation factor \([2 \ 2]\) is equivalent to a 5-by-5 filter with zeros between the elements.

Example: \([2 \ 3]\)

**PaddingSize — Size of padding**

\([0 \ 0 \ 0 \ 0]\) (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector \([t \ b \ l \ r]\) of four nonnegative integers, where \(t\) is the padding applied to the top, \(b\) is the padding applied to the bottom, \(l\) is the padding applied to the left, and \(r\) is the padding applied to the right.

When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: \([1 \ 1 \ 2 \ 2]\) adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

**PaddingMode — Method to determine padding size**

'manual' (default) | 'same'

Method to determine padding size, specified as 'manual' or 'same'.

The software automatically sets the value of PaddingMode based on the 'Padding' value you specify when creating a layer.

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets PaddingMode to 'manual'.
- If you set the 'Padding' option to 'same', then the software automatically sets PaddingMode to 'same' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is \(\text{ceil}(\text{inputSize}/\text{stride})\), where \(\text{inputSize}\) is the height or width of the input and \(\text{stride}\) is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an
odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

**Padding — Size of padding**

\[ [0 \ 0] \text{ (default)} \mid \text{vector of two nonnegative integers} \]

**Note** Padding property will be removed in a future release. Use PaddingSize instead. When creating a layer, use the 'Padding' name-value pair argument to specify the padding size.

Size of padding to apply to input borders vertically and horizontally, specified as a vector \([a \ b]\) of two nonnegative integers, where \(a\) is the padding applied to the top and bottom of the input data and \(b\) is the padding applied to the left and right.

Example: \([1 \ 1]\) adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

**NumChannels — Number of channels for each filter**

'auto' (default) | positive integer

Number of channels for each filter, specified as 'auto' or a positive integer.

This parameter is always equal to the number of channels of the input to the convolutional layer. For example, if the input is a color image, then the number of channels for the input is 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for the current layer is 16.

If NumChannels is 'auto', then the software determines the number of channels at training time.

Example: 256

**Parameters and Initialization**

**WeightsInitializer — Function to initialize weights**

'glorot' (default) | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:
• 'glorot' – Initialize the weights with the Glorot initializer [4] (also known as Xavier
  initializer). The Glorot initializer independently samples from a uniform distribution
  with zero mean and variance $2/(\text{numIn} + \text{numOut})$, where
  \[ \text{numIn} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{NumChannels} \]
  and
  \[ \text{numOut} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{NumFilters}. \]
• 'he' – Initialize the weights with the He initializer [5]. The He initializer samples from
  a normal distribution with zero mean and variance $2/\text{numIn}$, where
  \[ \text{numIn} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{NumChannels}. \]
• 'narrow-normal' – Initialize the weights by independently sampling from a normal
  distribution with zero mean and standard deviation 0.01.
• 'zeros' – Initialize the weights with zeros.
• 'ones' – Initialize the weights with ones.
• Function handle – Initialize the weights with a custom function. If you specify a
  function handle, then the function must be of the form \( \text{weights} = \text{func}(sz) \), where
  \( sz \) is the size of the weights. For an example, see “Specify Custom Weight
  Initialization Function”.

The layer only initializes the weights when the \texttt{Weights} property is empty.

Data Types: char | string | function_handle

**BiasInitializer — Function to initialize bias**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

• 'zeros' – Initialize the bias with zeros.
• 'ones' – Initialize the bias with ones.
• 'narrow-normal' – Initialize the bias by independently sampling from a normal
  distribution with zero mean and standard deviation 0.01.
• Function handle – Initialize the bias with a custom function. If you specify a function
  handle, then the function must be of the form \( \text{bias} = \text{func}(sz) \), where \( sz \) is the size
  of the bias.

The layer only initializes the bias when the \texttt{Bias} property is empty.

Data Types: char | string | function_handle

**Weights — Layer weights**

[ ] (default) | numeric array
Layer weights for the convolutional layer, specified as a numeric array.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the Weights property of the layer. When training a network, if the Weights property of the layer is nonempty, then trainNetwork uses the Weights property as the initial value. If the Weights property is empty, then trainNetwork uses the initializer specified by the WeightsInitializer property of the layer.

At training time, Weights is a FilterSize(1)-by-FilterSize(2)-by-NumChannels-by-NumFilters array.

Data Types: single | double

**Bias — Layer biases**

[] (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if Bias is nonempty, then trainNetwork uses the Bias property as the initial value. If Bias is empty, then trainNetwork uses the initializer specified by BiasInitializer.

At training time, Bias is a 1-by-1-by-NumFilters array.

Data Types: single | double

**Learn Rate and Regularization**

**WeightLearnRateFactor — Learning rate factor for weights**

1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if WeightLearnRateFactor is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

Example: 2

**BiasLearnRateFactor — Learning rate factor for biases**

1 (default) | nonnegative scalar
Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if BiasLearnRateFactor is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

Example: 2

WeightL2Factor — L2 regularization factor for weights
1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if WeightL2Factor is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the trainingOptions function.

Example: 2

BiasL2Factor — L2 regularization factor for biases
0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if BiasL2Factor is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the trainingOptions function.

Example: 2

Layer

Name — Layer name
'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with
the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'}in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'}out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**

**Create Convolutional Layer**

Create a convolutional layer with 96 filters, each with a height and width of 11. Use a stride (step size) of 4 in the horizontal and vertical directions.

```
layer = convolution2dLayer(11,96,'Stride',4)
```

```
layer = Convolution2DLayer with properties:
```
Include a convolutional layer in a Layer array.

```
layers = [imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
classificationLayer]
```

layers =
  7x1 Layer array with layers:

  1  ''  Image Input  28x28x1 images with 'zerocenter' normalization
  2  ''  Convolution  20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
  3  ''  ReLU         ReLU
  4  ''  Max Pooling  2x2 max pooling with stride [2 2] and padding [0 0 0 0]
  5  ''  Fully Connected 10 fully connected layer
  6  ''  Softmax       softmax
  7  ''  Classification Output  crossentropyex
Specify Initial Weights and Biases in Convolutional Layer

To specify the weights and bias initializer functions, use the `WeightsInitializer` and `BiasInitializer` properties respectively. To specify the weights and biases directly, use the `Weights` and `Bias` properties respectively.

Specify Initialization Functions

Create a convolutional layer with 32 filters, each with a height and width of 5 and specify the weights initializer to be the He initializer:

```matlab
filterSize = 5;
numFilters = 32;
layer = convolution2dLayer(filterSize,numFilters, ...
    'WeightsInitializer','he')
```

```matlab
layer = 
    Convolution2DLayer with properties:

    Hyperparameters
    FilterSize: [5 5]
    NumChannels: 'auto'
    NumFilters: 32
    Stride: [1 1]
    DilationFactor: [1 1]
    PaddingMode: 'manual'
    PaddingSize: [0 0 0 0]

    Learnable Parameters
    Weights: []
    Bias: []
```

Note that the `Weights` and `Bias` properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Custom Initialization Functions

To specify your own initialization function for the weights and biases, set the `WeightsInitializer` and `BiasInitializer` properties to a function handle. For
these properties, specify function handles that take the size of the weights and biases as input and output the initialized value.

Create a convolutional layer with 32 filters, each with a height and width of 5 and specify initializers that sample the weights and biases from a Gaussian distribution with a standard deviation of 0.0001.

```matlab
filterSize = 5;
numFilters = 32;

layer = convolution2dLayer(filterSize,numFilters, ... 
    'WeightsInitializer', @(sz) rand(sz) * 0.0001, ... 
    'BiasInitializer', @(sz) rand(sz) * 0.0001)
```

```
layer = 
    Convolution2DLayer with properties:
        Name: ''

    Hyperparameters
        FilterSize: [5 5]
        NumChannels: 'auto'
        NumFilters: 32
        Stride: [1 1]
        DilationFactor: [1 1]
        PaddingMode: 'manual'
        PaddingSize: [0 0 0 0]

    Learnable Parameters
        Weights: []
        Bias: []
```

Again, the Weights and Bias properties are empty. At training time, the software initializes these properties using the specified initialization functions.

**Specify Weights and Bias Directly**

Create a fully connected layer with an output size of 10 and set the weights and bias to \( W \) and \( b \) in the MAT file Conv2dWeights.mat respectively.

```matlab
filterSize = 5;
numFilters = 32;
```
load Conv2dWeights

layer = convolution2dLayer(filterSize,numFilters, ... 
    'Weights',W, ... 
    'Bias',b)

layer = 
Convolution2DLayer with properties:

    Name: '

    Hyperparameters
    FilterSize: [5 5]
    NumChannels: 3
    NumFilters: 32
    Stride: [1 1]
    DilationFactor: [1 1]
    PaddingMode: 'manual'
    PaddingSize: [0 0 0 0]

    Learnable Parameters
    Weights: [5x5x3x32 double]
    Bias: [1x1x32 double]

    Show all properties

Here, the Weights and Bias properties contain the specified values. At training time, if these properties are non-empty, then the software uses the specified values as the initial weights and biases. In this case, the software does not use the initializer functions.

**Create Convolutional Layer That Fully Covers Input**

Suppose the size of the input is 28-by-28-by-1. Create a convolutional layer with 16 filters, each with a height of 6 and a width of 4. Set the horizontal and vertical stride to 4.

Make sure the convolution covers the input completely. For the convolution to fully cover the input, both the horizontal and vertical output dimensions must be integer numbers. For the horizontal output dimension to be an integer, one row of zero padding is required on the top and bottom of the image: \((28 - 6 + 2 \times 1)/4 + 1 = 7\). For the vertical output dimension to be an integer, no zero padding is required: \((28 - 4 + 2 \times 0)/4 + 1 = 7\).

Construct the convolutional layer.
layer = convolution2dLayer([6 4],16,'Stride',4,'Padding',[1 0])

layer = 
    Convolution2DLayer with properties:
        Name: ''

        Hyperparameters
            FilterSize: [6 4]
            NumChannels: 'auto'
            NumFilters: 16
            Stride: [4 4]
            DilationFactor: [1 1]
            PaddingMode: 'manual'
            PaddingSize: [1 1 0 0]

        Learnable Parameters
            Weights: []
            Bias: []

        Show all properties

More About

Convolutional Layer

A 2-D convolutional layer applies sliding convolutional filters to the input. The layer convolves the input by moving the filters along the input vertically and horizontally, computing the dot product of the weights and the input, and then adding a bias term.

The convolutional layer consists of various components.¹

Filters and Stride

A convolutional layer consists of neurons that connect to subregions of the input images or the outputs of the previous layer. The layer learns the features localized by these regions while scanning through an image. When creating a layer using the

¹. Image credit: Convolution arithmetic (License)
convolution2dLayer function, you can specify the size of these regions using the filterSize input argument.

For each region, the trainNetwork function computes a dot product of the weights and the input, and then adds a bias term. A set of weights that is applied to a region in the image is called a filter. The filter moves along the input image vertically and horizontally, repeating the same computation for each region. In other words, the filter convolves the input.

This image shows a 3-by-3 filter scanning through the input. The lower map represents the input and the upper map represents the output.

The step size with which the filter moves is called a stride. You can specify the step size with the Stride name-value pair argument. The local regions that the neurons connect to can overlap depending on the filterSize and 'Stride' values.

This image shows a 3-by-3 filter scanning through the input with a stride of 2. The lower map represents the input and the upper map represents the output.
The number of weights in a filter is $h \times w \times c$, where $h$ is the height, and $w$ is the width of the filter, respectively, and $c$ is the number of channels in the input. For example, if the input is a color image, the number of color channels is 3. The number of filters determines the number of channels in the output of a convolutional layer. Specify the number of filters using the `numFilters` argument with the `convolution2dLayer` function.

**Dilated Convolutions**

A dilated convolution is a convolution in which the filters are expanded by spaces inserted between the elements of the filter. Specify the dilation factor using the `'DilationFactor'` property.

Use dilated convolutions to increase the receptive field (the area of the input which the layer can see) of the layer without increasing the number of parameters or computation.

The layer expands the filters by inserting zeros between each filter element. The dilation factor determines the step size for sampling the input or equivalently the upsampling factor of the filter. It corresponds to an effective filter size of $(\text{Filter Size} - 1) \times \text{Dilation Factor} + 1$. For example, a 3-by-3 filter with the dilation factor $[2\ 2]$ is equivalent to a 5-by-5 filter with zeros between the elements.
This image shows a 3-by-3 filter dilated by a factor of two scanning through the input. The lower map represents the input and the upper map represents the output.

**Feature Maps**

As a filter moves along the input, it uses the same set of weights and the same bias for the convolution, forming a *feature map*. Each feature map is the result of a convolution using a different set of weights and a different bias. Hence, the number of feature maps is equal to the number of filters. The total number of parameters in a convolutional layer is $((h \times w \times c + 1) \times \text{Number of Filters})$, where 1 is the bias.

**Zero Padding**

You can also apply zero padding to input image borders vertically and horizontally using the 'Padding' name-value pair argument. Padding is rows or columns of zeros added to
the borders of an image input. By adjusting the padding, you can control the output size of the layer.

This image shows a 3-by-3 filter scanning through the input with padding of size 1. The lower map represents the input and the upper map represents the output.

**Output Size**

The output height and width of a convolutional layer is \((Input\ Size - ((Filter\ Size - 1)\times Dilation\ Factor + 1) + 2\times Padding)/Stride + 1\). This value must be an integer for the whole image to be fully covered. If the combination of these options does not lead the
image to be fully covered, the software by default ignores the remaining part of the image along the right and bottom edges in the convolution.

**Number of Neurons**

The product of the output height and width gives the total number of neurons in a feature map, say *Map Size*. The total number of neurons (output size) in a convolutional layer is *Map Size* * Number of Filters.

For example, suppose that the input image is a 32-by-32-by-3 color image. For a convolutional layer with eight filters and a filter size of 5-by-5, the number of weights per filter is $5 \times 5 \times 3 = 75$, and the total number of parameters in the layer is $(75 + 1) \times 8 = 608$. If the stride is 2 in each direction and padding of size 2 is specified, then each feature map is 16-by-16. This is because $(32 - 5 + 2 \times 2)/2 + 1 = 16.5$, and some of the outermost zero padding to the right and bottom of the image is discarded. Finally, the total number of neurons in the layer is $16 \times 16 \times 8 = 2048$.

Usually, the results from these neurons pass through some form of nonlinearity, such as rectified linear units (ReLU).

**Learnable Parameters**

You can adjust the learning rates and regularization options for the layer using name-value pair arguments while defining the convolutional layer. If you choose not to specify these options, then `trainNetwork` uses the global training options defined with the `trainingOptions` function. For details on global and layer training options, see “Set Up Parameters and Train Convolutional Neural Network”.

**Number of Layers**

A convolutional neural network can consist of one or multiple convolutional layers. The number of convolutional layers depends on the amount and complexity of the data.

**Compatibility Considerations**

**Default weights initialization is Glorot**

*Behavior changed in R2019a*
Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'WeightsInitializer' option of the layer to 'narrow-normal'.

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.
See Also
batchNormalizationLayer | fullyConnectedLayer |
groupedConvolution2dLayer | maxPooling2dLayer | reluLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2016a
convolution3dLayer

3-D convolutional layer

Description

A 3-D convolutional layer applies sliding cuboidal convolution filters to three-dimensional input. The layer convolves the input by moving the filters along the input vertically, horizontally, and along the depth, computing the dot product of the weights and the input, and then adding a bias term.

Creation

Syntax

layer = convolution3dLayer(filterSize,numFilters)
layer = convolution3dLayer(filterSize,numFilters,Name,Value)

Description

layer = convolution3dLayer(filterSize,numFilters) creates a 3-D convolutional layer and sets the FilterSize and NumFilters properties.

layer = convolution3dLayer(filterSize,numFilters,Name,Value) sets the optional Stride, DilationFactor, NumChannels, “Parameters and Initialization” on page 1-891, “Learn Rate and Regularization” on page 1-893, and Name properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example, convolution3dLayer(11,96,'Stride',4,'Padding',1) creates a 3-D convolutional layer with 96 filters of size [11 11 11], a stride of [4 4 4], and zero padding of size 1 along all edges of the layer input. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the Stride, DilationFactor, NumChannels, “Parameters and Initialization” on page 1-891, “Learn Rate and Regularization” on page 1-893, and Name properties. Enclose names in single quotes.

Example: convolution3dLayer(3,16,'Padding','same') creates a 3-D convolutional layer with 16 filters of size [3 3 3] and 'same' padding. At training time, the software calculates and sets the size of the zero padding so that the layer output has the same size as the input.

Padding — Input edge padding
0 (default) | array of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

- 'same' — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is ceil(inputSize/stride), where inputSize is the height, width, or depth of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

- Nonnegative integer p — Add padding of size p to all the edges of the input.

- Three-element vector [a b c] of nonnegative integers — Add padding of size a to the top and bottom, padding of size b to the left and right, and padding of size c to the front and back of the input.

- 2-by-3 matrix [t l f;b r k] of nonnegative integers — Add padding of size t to the top, b to the bottom, l to the left, r to the right, f to the front, and k to the back of the input. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

Example: 'Padding',1 adds one row of padding to the top and bottom, one column of padding to the left and right, and one plane of padding to the front and back of the input.
Example: 'Padding', 'same' adds padding so that the output has the same size as the input (if the stride equals 1).

**Properties**

**Convolution**

**FilterSize — Height, width, and depth of filters**
vector of three positive integers

Height, width, and depth of the filters, specified as a vector [h w d] of three positive integers, where h is the height, w is the width, and d is the depth. FilterSize defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify FilterSize as a scalar to use the same value for the height, width, and depth.

Example: [5 5 5] specifies filters with a height, width, and depth of 5.

**NumFilters — Number of filters**
positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96

**Stride — Step size for traversing input**
[1 1 1] (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector [a b c] of three positive integers, where a is the vertical step size, b is the horizontal step size, and c is the step size along the depth. When creating the layer, you can specify Stride as a scalar to use the same value for step sizes in all three directions.

Example: [2 3 1] specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.
DilationFactor — Factor for dilated convolution

[1 1 1] (default) | vector of three positive integers

Factor for dilated convolution (also known as atrous convolution), specified as a vector \([h \ w \ d]\) of three positive integers, where \(h\) is the vertical dilation, \(w\) is the horizontal dilation, and \(d\) is the dilation along the depth. When creating the layer, you can specify DilationFactor as a scalar to use the same value for dilation in all three directions.

Use dilated convolutions to increase the receptive field (the area of the input which the layer can see) of the layer without increasing the number of parameters or computation.

The layer expands the filters by inserting zeros between each filter element. The dilation factor determines the step size for sampling the input or equivalently the upsampling factor of the filter. It corresponds to an effective filter size of \((\text{Filter Size} - 1) \times \text{Dilation Factor} + 1\). For example, a 3-by-3-by-3 filter with the dilation factor \([2 2 2]\) is equivalent to a 5-by-5-by-5 filter with zeros between the elements.

Example: \([2 3 1]\) dilates the filter vertically by a factor of 2, horizontally by a factor of 3, and along the depth by a factor of 1.

PaddingSize — Size of padding

[0 0 0;0 0 0] (default) | 2-by-3 matrix of nonnegative integers

Size of padding to apply to input borders, specified as 2-by-3 matrix \([t \ l \ f; b \ r \ k]\) of nonnegative integers, where \(t\) and \(b\) are the padding applied to the top and bottom in the vertical direction, \(l\) and \(r\) are the padding applied to the left and right in the horizontal direction, and \(f\) and \(k\) are the padding applied to the front and back along the depth. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: \([1 2 4;1 2 4]\) adds one row of padding to the top and bottom, two columns of padding to the left and right, and four planes of padding to the front and back of the input.

PaddingMode — Method to determine padding size

'manual' (default) | 'same'

Method to determine padding size, specified as 'manual' or 'same'.

1-890
The software automatically sets the value of PaddingMode based on the 'Padding' value you specify when creating a layer.

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets PaddingMode to 'manual'.
- If you set the 'Padding' option to 'same', then the software automatically sets PaddingMode to 'same' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is \( \text{ceil} \left( \frac{\text{inputSize}}{\text{stride}} \right) \), where inputSize is the height, width, or depth of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

**NumChannels — Number of channels for each filter**

'auto' (default) | positive integer

Number of channels for each filter, specified as 'auto' or a positive integer.

This parameter is always equal to the number of channels of the input to the convolutional layer. For example, if the input is a color image, then the number of channels for the input is 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for the current layer is 16.

If NumChannels is 'auto', then the software determines the number of channels at training time.

Example: 256

**Parameters and Initialization**

**WeightsInitializer — Function to initialize weights**

'glorot' (default) | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:

- 'glorot' - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution
with zero mean and variance \( 2/(\text{numIn} + \text{numOut}) \), where \( \text{numIn} = \text{FilterSize}(1) \times \text{FilterSize}(2) \times \text{FilterSize}(3) \times \text{NumChannels} \) and \( \text{numOut} = \text{FilterSize}(1) \times \text{FilterSize}(2) \times \text{FilterSize}(3) \times \text{NumFilters} \).

- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance \( 2/\text{numIn} \), where \( \text{numIn} = \text{FilterSize}(1) \times \text{FilterSize}(2) \times \text{FilterSize}(3) \times \text{NumChannels} \).

- 'narrow-normal' - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

- 'zeros' - Initialize the weights with zeros.

- 'ones' - Initialize the weights with ones.

- Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form \( \text{weights} = \text{func}(\text{sz}) \), where \( \text{sz} \) is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the \text{Weights} property is empty.

Data Types: char | string | function_handle

BiasInitializer — Function to initialize bias

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

- 'zeros' - Initialize the bias with zeros.

- 'ones' - Initialize the bias with ones.

- 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

- Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form \( \text{bias} = \text{func}(\text{sz}) \), where \( \text{sz} \) is the size of the bias.

The layer only initializes the bias when the \text{Bias} property is empty.

Data Types: char | string | function_handle

Weights — Layer weights

[] (default) | numeric array

Layer weights for the convolutional layer, specified as a numeric array.
The layer weights are learnable parameters. You can specify the initial value for the weights directly using the `Weights` property of the layer. When training a network, if the `Weights` property of the layer is nonempty, then `trainNetwork` uses the `Weights` property as the initial value. If the `Weights` property is empty, then `trainNetwork` uses the initializer specified by the `WeightsInitializer` property of the layer.

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`FilterSize(3)`-by-`NumChannels`-by-`NumFilters` array.

Data Types: `single` | `double`

**Bias — Layer biases**

`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-1-by-`NumFilters` array.

Data Types: `single` | `double`

**Learn Rate and Regularization**

**WeightLearnRateFactor — Learning rate factor for weights**

`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: `2`

**BiasLearnRateFactor — Learning rate factor for biases**

`1` (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.
The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**WeightL2Factor — L2 regularization factor for weights**
1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**BiasL2Factor — L2 regularization factor for biases**
0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**Layer**

**Name — Layer name**
'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to '', then the software automatically assigns a name to the layer at training time.
Data Types: char | string

**NumInputs — Number of inputs**  
1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**  
{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**  
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**  
{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

## Examples

### Create 3-D Convolution Layer

Create a 3-D convolution layer with 16 filters, each with a height, width, and depth of 5. Use a stride (step size) of 4 in all three directions.

```matlab
classname = 'Convolution3DLayer';
layer = convolution3dLayer(5,16,'Stride',4)
```

```matlab
Class: Convolution3DLayer
```

```matlab
layer = 
    Convolution3DLayer with properties:
```

1-895
Hyperparameters
- FilterSize: [5 5 5]
- NumChannels: 'auto'
- NumFilters: 16
- Stride: [4 4 4]
- DilationFactor: [1 1 1]
- PaddingMode: 'manual'
- PaddingSize: [2x3 double]

Learnable Parameters
- Weights: []
- Bias: []

Show all properties

Include a 3-D convolution layer in a Layer array.

```matlab
layers = [ ... 
    image3dInputLayer([28 28 28 3]) 
    convolution3dLayer(5,16,'Stride',4) 
    reluLayer 
    maxPooling3dLayer(2,'Stride',4) 
    fullyConnectedLayer(10) 
    softmaxLayer 
    classificationLayer
]
```

layers =
7x1 Layer array with layers:

- 1: 3-D Image Input 28x28x28x3 images with 'zerocenter' normalization
- 2: Convolution 16 5x5x5 convolutions with stride [4 4 4] and padding [0 0 0; 0 0 0]
- 3: ReLU ReLU
- 4: 3-D Max Pooling 2x2x2 max pooling with stride [4 4 4] and padding [0 0 0; 0 0 0]
- 5: Fully Connected 10 fully connected layer
- 6: Softmax softmax
- 7: Classification Output crossentropyex
Specify Initial Weights and Biases in 3-D Convolutional Layer

To specify the weights and bias initializer functions, use the WeightsInitializer and BiasInitializer properties respectively. To specify the weights and biases directly, use the Weights and Bias properties respectively.

Specify Initialization Functions

Create a 3-D convolutional layer with 32 filters, each with a height, width, and depth of 5. Specify the weights initializer to be the He initializer.

```matlab
filterSize = 5;
numFilters = 32;
layer = convolution3dLayer(filterSize,numFilters, ...
    'WeightsInitializer','he')
layer = 
    Convolution3DLayer with properties:

    Name: ''

    Hyperparameters
    FilterSize: [5 5 5]
    NumChannels: 'auto'
    NumFilters: 32
    Stride: [1 1 1]
    DilationFactor: [1 1 1]
    PaddingMode: 'manual'
    PaddingSize: [2x3 double]

    Learnable Parameters
    Weights: []
    Bias: []

    Show all properties
```

Note that the Weights and Bias properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Custom Initialization Functions

To specify your own initialization function for the weights and biases, set the WeightsInitializer and BiasInitializer properties to a function handle. For
these properties, specify function handles that take the size of the weights and biases as input and output the initialized value.

Create a convolutional layer with 32 filters, each with a height, width, and depth of 5. Specify initializers that sample the weights and biases from a Gaussian distribution with a standard deviation of 0.0001.

```matlab
filterSize = 5;
umFilters = 32;

layer = convolution3dLayer(filterSize,numFilters, ...  'WeightsInitializer', @(sz) rand(sz) * 0.0001, ...  'BiasInitializer', @(sz) rand(sz) * 0.0001)
```

Again, the **Weights** and **Bias** properties are empty. At training time, the software initializes these properties using the specified initialization functions.

**Specify Weights and Bias Directly**

Create a 3-D convolutional layer compatible with color images. Set the weights and bias to \( W \) and \( b \) in the MAT file `Conv3dWeights.mat` respectively.

```matlab
filterSize = 5;
umFilters = 32;
```
Here, the **Weights** and **Bias** properties contain the specified values. At training time, if these properties are non-empty, then the software uses the specified values as the initial weights and biases. In this case, the software does not use the initializer functions.

### Create Convolutional Layer That Fully Covers 3-D Input

Suppose the size of the input is 28-by-28-by-28-by-1. Create a 3-D convolutional layer with 16 filters, each with a height of 6, a width of 4, and a depth of 5. Set the stride in all dimensions to 4.

Make sure the convolution covers the input completely. For the convolution to fully cover the input, the output dimensions must be integer numbers. When there is no dilation, the $i$-th output dimension is calculated as \((\text{imageSize}(i) - \text{filterSize}(i) + \text{padding}(i)) / \text{stride}(i) + 1\).
- For the horizontal output dimension to be an integer, two rows of zero padding are required: \((28 - 6 + 2)/4 + 1 = 7\). Distribute the padding symmetrically by adding one row of padding at the top and bottom of the image.

- For the vertical output dimension to be an integer, no zero padding is required: \((28 - 4 + 0)/4 + 1 = 7\).

- For the depth output dimension to be an integer, one plane of zero padding is required: \((28 - 5 + 1)/4 + 1 = 7\). You must distribute the padding asymmetrically across the front and back of the image. This example adds one plane of zero padding to the back of the image.

Construct the convolutional layer. Specify 'Padding' as a 2-by-3 matrix. The first row specifies prepadding and the second row specifies postpadding in the three dimensions.

```matlab
layer = convolution3dLayer([6 4 5],16,'Stride',4,'Padding',[1 0 0;1 0 1])
```

```
layer = 
  Convolution3DLayer with properties: 

  Name: ''

  Hyperparameters 
    FilterSize: [6 4 5]
    NumChannels: 'auto'
    NumFilters: 16
    Stride: [4 4 4]
    DilationFactor: [1 1 1]
    PaddingMode: 'manual'
    PaddingSize: [2x3 double]

  Learnable Parameters 
    Weights: []
    Bias: []

Show all properties
```
More About

3-D Convolutional Layer

A convolutional layer applies sliding convolutional filters to the input. A 3-D convolutional layer extends the functionality of a 2-D convolutional layer to a third dimension, depth. The layer convolves the input by moving the filters along the input vertically, horizontally, and along the depth, computing the dot product of the weights and the input, and then adding a bias term. To learn more, see the definition of convolutional layer on page 1-879 on the convolution2dLayer reference page.

References


See Also

convolution2dLayer | globalAveragePooling3dLayer | image3dInputLayer | maxPooling3dLayer

Topics

“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2019a
groupedConvolution2dLayer

2-D grouped convolutional layer

Description

A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution.

For each group, the layer convolves the input by moving the filters along the input vertically and horizontally and computing the dot product of the weights and the input, and then adding a bias term. The layer combines the convolutions for each group independently. If the number of groups is equal to the number of channels, then this layer performs channel-wise convolution.

Creation

Syntax

layer = groupedConvolution2dLayer(filterSize,numFiltersPerGroup, numGroups)
layer = groupedConvolution2dLayer(filterSize, numFiltersPerGroup,'channel-wise')
layer = groupedConvolution2dLayer(____,Name,Value)

Description

layer = groupedConvolution2dLayer(filterSize,numFiltersPerGroup, numGroups) creates a 2-D grouped convolutional layer and sets the FilterSize, NumFiltersPerGroup, and NumGroups properties.

layer = groupedConvolution2dLayer(filterSize, numFiltersPerGroup,'channel-wise') creates a layer for channel-wise convolution
(also known as depth-wise convolution). In this case, the software determines the NumGroups property at training time. This syntax is equivalent to setting NumGroups to the number of input channels.

\[
\text{layer} = \text{groupedConvolution2dLayer}(\_, \text{Name, Value})
\]

sets the optional Stride, DilationFactor, “Parameters and Initialization” on page 1-907, “Learn Rate and Regularization” on page 1-908, and Name properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example, \(\text{groupedConvolution2dLayer}(5, 128, 2, 'Padding', 'same')\) creates a 2-D grouped convolutional layer with 2 groups of 128 filters of size [5 5] and pads the input to so that the output has the same size. You can specify multiple name-value pairs. Enclose each property name in single quotes.

### Input Arguments

**Name-Value Pair Arguments**

Use comma-separated name-value pair arguments to specify the size of the zero padding to add along the edges of the layer input or to set the Stride, DilationFactor, “Parameters and Initialization” on page 1-907, “Learn Rate and Regularization” on page 1-908, and Name properties. Enclose names in single quotes.

Example: \(\text{groupedConvolution2dLayer}(5, 128, 2, 'Padding', 'same')\) creates a 2-D grouped convolutional layer with 2 groups of 128 filters of size [5 5] and pads the input to so that the output has the same size.

**Padding — Input edge padding**

[0 0 0 0] (default) | vector of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

- 'same' — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is \(\text{ceil}((\text{inputSize} / \text{stride}))\), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.
• Nonnegative integer \( p \) — Add padding of size \( p \) to all the edges of the input.
• Vector \([a \ b]\) of nonnegative integers — Add padding of size \( a \) to the top and bottom of the input and padding of size \( b \) to the left and right.
• Vector \([t \ b \ l \ r]\) of nonnegative integers — Add padding of size \( t \) to the top, \( b \) to the bottom, \( l \) to the left, and \( r \) to the right of the input.

Example: 'Padding',1 adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: 'Padding','same' adds padding so that the output has the same size as the input (if the stride equals 1).

Properties

Grouped Convolution

**FilterSize — Height and width of filters**
vector of two positive integers

Height and width of the filters, specified as a vector \([h \ w]\) of two positive integers, where \( h \) is the height and \( w \) is the width. **FilterSize** defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify **FilterSize** as a scalar to use the same value for the height and width.

Example: \([5 \ 5]\) specifies filters with a height of 5 and a width of 5.

**NumFiltersPerGroup — Number of filters per group**
positive integer

Number of filters per group, specified as a positive integer. This property determines the number of channels in the output of the layer. The number of output channels is **FiltersPerGroup** * **NumGroups**.

Example: 10

**NumGroups — Number of groups**
positive integer | 'channel-wise'

Number of groups, specified as a positive integer or 'channel-wise'.

1-904
If `NumGroups` is 'channel-wise', then the software creates a layer for channel-wise convolution (also known as depth-wise convolution). In this case, the layer determines the `NumGroups` property at training time. This value is equivalent to setting `NumGroups` to the number of input channels.

The number of groups must evenly divide the number of channels of the layer input.

Example: 2

**Stride — Step size for traversing input**

`[1 1]` (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector `[a b]` of two positive integers, where `a` is the vertical step size and `b` is the horizontal step size. When creating the layer, you can specify `Stride` as a scalar to use the same value for both step sizes.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

**DilationFactor — Factor for dilated convolution**

`[1 1]` (default) | vector of two positive integers

Factor for dilated convolution (also known as atrous convolution), specified as a vector `[h w]` of two positive integers, where `h` is the vertical dilation and `w` is the horizontal dilation. When creating the layer, you can specify `DilationFactor` as a scalar to use the same value for both horizontal and vertical dilations.

Use dilated convolutions to increase the receptive field (the area of the input which the layer can see) of the layer without increasing the number of parameters or computation.

The layer expands the filters by inserting zeros between each filter element. The dilation factor determines the step size for sampling the input or equivalently the upsampling factor of the filter. It corresponds to an effective filter size of `(Filter Size - 1) * Dilation Factor + 1`. For example, a 3-by-3 filter with the dilation factor `[2 2]` is equivalent to a 5-by-5 filter with zeros between the elements.

Example: `[2 3]`

**PaddingSize — Size of padding**

`[0 0 0 0]` (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector `[t b l r]` of four nonnegative integers, where `t` is the padding applied to the top, `b` is the padding applied
to the bottom, \( l \) is the padding applied to the left, and \( r \) is the padding applied to the right.

When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: \([1 \ 1 \ 2 \ 2]\) adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

**PaddingMode — Method to determine padding size**

'manual' (default) | 'same'

Method to determine padding size, specified as 'manual' or 'same'.

The software automatically sets the value of PaddingMode based on the 'Padding' value you specify when creating a layer.

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets PaddingMode to 'manual'.
- If you set the 'Padding' option to 'same', then the software automatically sets PaddingMode to 'same' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is \( \text{ceil}(\text{inputSize} / \text{stride}) \), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

**NumChannelsPerGroup — Number of channels per group**

'auto' (default) | positive integer

Number of channels per group, specified as 'auto' or a positive integer. The number of channels per group is equal to the number of input channels divided by the number of groups.

The software automatically sets this property at training time.

Example: 256
Parameters and Initialization

WeightsInitializer — Function to initialize weights

'glorot' (default) | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:

• 'glorot' - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance \(2/(\text{numIn} + \text{numOut})\), where \(\text{numIn} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{NumChannelsPerGroup}\) and \(\text{numOut} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{NumFiltersPerGroup}\).

• 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance \(2/\text{numIn}\), where \(\text{numIn} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{NumChannelsPerGroup}\).

• 'narrow-normal' - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

• 'zeros' - Initialize the weights with zeros.

• 'ones' - Initialize the weights with ones.

• Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form \(\text{weights} = \text{func(sz)}\), where \(\text{sz}\) is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the Weights property is empty.

Data Types: char | string | function_handle

BiasInitializer — Function to initialize bias

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

• 'zeros' - Initialize the bias with zeros.

• 'ones' - Initialize the bias with ones.

• 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

• Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form \(\text{bias} = \text{func(sz)}\), where \(\text{sz}\) is the size of the bias.
The layer only initializes the bias when the Bias property is empty.

Data Types: char | string | function_handle

**Weights — Layer weights**

[-] (default) | numeric array

Layer weights for the layer, specified as a numeric array.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the Weights property of the layer. When training a network, if the Weights property of the layer is nonempty, then trainNetwork uses the Weights property as the initial value. If the Weights property is empty, then trainNetwork uses the initializer specified by the WeightsInitializer property of the layer.

At training time, Weights is a FilterSize(1)-by-FilterSize(2)-by-NumChannelsPerGroup-by-NumFiltersPerGroup-by-NumGroups array, where NumInputChannels is the number of channels of the layer input.

Data Types: single | double

**Bias — Layer biases**

[-] (default) | numeric array

Layer biases for the layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if Bias is nonempty, then trainNetwork uses the Bias property as the initial value. If Bias is empty, then trainNetwork uses the initializer specified by BiasInitializer.

At training time, Bias is a 1-by-1-by-NumFiltersPerGroup-by-NumGroups array.

Data Types: single | double

**Learn Rate and Regularization**

**WeightLearnRateFactor — Learning rate factor for weights**

1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if WeightLearnRateFactor is 2, then the...
The learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**BiasLearnRateFactor — Learning rate factor for biases**

1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**WeightL2Factor — L2 regularization factor for weights**

1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**BiasL2Factor — L2 regularization factor for biases**

0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2
Layer

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**
Create Grouped Convolution Layer

Create a grouped convolutional layer with 3 groups of 10 filters, each with a height and width of 11, and the name 'gconv1'.

layer = groupedConvolution2dLayer(11,10,3,'Name','gconv1')

Create Channel-Wise Convolution Layer

Create a channel-wise convolutional (also known as depth-wise convolutional) layer with groups of 10 filters, each with a height and width of 11, and the name 'cwconv1'.

layer = groupedConvolution2dLayer(11,10,'channel-wise','Name','cwconv1')
Create Layers for Channel-Wise Separable Convolution

A typical convolutional neural network contains blocks of convolution, batch normalization, and ReLU layers. For example,

```matlab
cfilterSize = 3;
numFilters = 16;

convLayers = [
    convolution2dLayer(filterSize, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer
];
```

For channel-wise separable convolution (also known as depth-wise separable convolution), replace the convolution block with channel-wise convolution and point-wise convolution blocks.

Specify the filter size and the stride in the channel-wise convolution and the number of filters in the point-wise convolution. For the channel-wise convolution, specify one filter per group. For point-wise convolution, specify filters of size 1 in convolution2dLayer.

```matlab
cwsConvLayers = [
    groupedConvolution2dLayer(filterSize, 1, 'channel-wise', 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer
];
```
convolution2dLayer(1,numFilters,'Padding','same')
batchNormalizationLayer
reluLayer;

Create a network containing layers for channel-wise separable convolution.

layers = [
    imageInputLayer([227 227 3])
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    groupedConvolution2dLayer(3,1,'channel-wise','Stride',2,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(1,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(5)
    softmaxLayer
    classificationLayer];

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
batchNormalizationLayer | convolution2dLayer | fullyConnectedLayer |
maxPooling2dLayer | reluLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2019a
transposedConv2dLayer

Transposed 2-D convolution layer

Syntax

layer = transposedConv2dLayer(filterSize,numFilters)
layer = transposedConv2dLayer(filterSize,numFilters,Name,Value)

Description

A transposed 2-D convolution layer upsamples feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

layer = transposedConv2dLayer(filterSize,numFilters) returns a transposed 2-D convolution layer and sets the filterSize and numFilters properties.

layer = transposedConv2dLayer(filterSize,numFilters,Name,Value) returns a transposed 2-D convolutional layer and specifies additional options using one or more name-value pair arguments.

Examples

Create Transposed Convolutional Layer

Create a transposed convolutional layer with 96 filters, each with a height and width of 11. Use a stride of 4 in the horizontal and vertical directions.

layer = transposedConv2dLayer(11,96,'Stride',4);
**Input Arguments**

**filterSize — Height and width of filters**
vector of two positive integers

Height and width of the filters, specified as a vector of two positive integers \([h\ w]\), where \(h\) is the height and \(w\) is the width. FilterSize defines the size of the local regions to which the neurons connect in the input.

If you set FilterSize using an input argument, then you can specify FilterSize as scalar to use the same value for both dimensions.

Example: \([5\ 5]\) specifies filters of height 5 and width 5.

**numFilters — Number of filters**
positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name,Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as

\[\text{Name1,Value1}, \ldots, \text{NameN,ValueN}\].

Example: `'Cropping',1`

**Transposed Convolution**

**Stride — Up-sampling factor**
1 (default) | vector of two positive integers | positive integer

Up-sampling factor of the input, specified as one of the following:

- A vector of two positive integers \([a\ b]\), where \(a\) is the vertical stride and \(b\) is the horizontal stride.
• A positive integer the corresponds to both the vertical and horizontal stride.

Example: 'Stride',[2 1]

**Cropping — Output size reduction**

0 (default) | 'same' | nonnegative integer | vector of two nonnegative integers

Output size reduction, specified as one of the following:

• 'same' - Set the cropping so that the output size equals inputSize .* Stride, where inputSize is the height and width of the layer input. If you set the 'Cropping' option to 'same', then the software automatically sets the CroppingMode property of the layer to 'same'.

  The software trims an equal amount from the top and bottom, and the left and right, if possible. If the vertical crop amount has an odd value, then the software trims an extra row from the bottom. If the horizontal crop amount has an odd value, then the software trims an extra column from the right.

• A positive integer - Crop the specified amount of data from all the edges.

• A vector of nonnegative integers [a b] - Crop a from the top and bottom and crop b from the left and right.

• A vector [t b l r] - Crop t, b, l, r from the top, bottom, left, and right of the input, respectively.

If you set the 'Cropping' option to a numeric value, then the software automatically sets the CroppingMode property of the layer to 'manual'.

Example: [1 2]

**NumChannels — Number of channels for each filter**

'auto' (default) | positive integer

Number of channels for each filter, specified as 'NumChannels' and 'auto' or a positive integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.
Parameters and Initialization

WeightsInitializer — Function to initialize weights
'glorot' (default) | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:

- 'glorot' - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance $2/(numIn + numOut)$, where $numIn = filterSize(1)\times filterSize(2)\times NumChannels$, $numOut = filterSize(1)\times filterSize(2)\times numFilters$, and $NumChannels$ is the number of input channels.
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance $2/numIn$, where $numIn = filterSize(1)\times filterSize(2)\times NumChannels$ and $NumChannels$ is the number of input channels.
- 'narrow-normal' - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- 'zeros' - Initialize the weights with zeros.
- 'ones' - Initialize the weights with ones.
- Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form weights = func(sz), where sz is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the Weights property is empty.

Data Types: char | string | function_handle

BiasInitializer — Function to initialize bias
'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

- 'zeros' - Initialize the bias with zeros.
- 'ones' - Initialize the bias with ones.
- 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
• Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: `char` | `string` | `function_handle`

**Weights — Layer weights**

`[]` (default) | numeric array

Layer weights for the convolutional layer, specified as a numeric array.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the `Weights` property of the layer. When training a network, if the `Weights` property of the layer is nonempty, then `trainNetwork` uses the `Weights` property as the initial value. If the `Weights` property is empty, then `trainNetwork` uses the initializer specified by the `WeightsInitializer` property of the layer.

At training time, `Weights` is a `filterSize(1)`-by-`filterSize(2)`-by-`numFilters`-by-`NumChannels` array.

Data Types: `single` | `double`

**Bias — Layer biases**

`[]` (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-`numFilters` array.

Data Types: `single` | `double`

**Learn Rate and Regularization**

**WeightLearnRateFactor — Learning rate factor for weights**

`1` (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.
The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**BiasLearnRateFactor — Learning rate factor for biases**

1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**WeightL2Factor — L2 regularization factor for weights**

1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**BiasL2Factor — L2 regularization factor for biases**

0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.
Layer

**Name — Layer name**

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**Output Arguments**

**layer — Transposed 2-D convolution layer**

TransposedConvolution2DLayer object

Transposed 2-D convolution layer, returned as a `TransposedConvolution2DLayer` object.

**Compatibility Considerations**

**Default weights initialization is Glorot**

*Behavior changed in R2019a*

Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'WeightsInitializer' option of the layer to 'narrow-normal'.
References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
SoftmaxLayer | TransposedConvolution2DLayer | averagePooling2dLayer | maxPooling2dLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2017b
TransposedConvolution2DLayer

Transposed 2-D convolution layer

Description

A transposed 2-D convolution layer upsamples feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

Creation

Create a transposed convolution 2-D output layer using `transposedConv2dLayer`.

Properties

Transposed Convolution

FilterSize — Height and width of filters

vector of two positive integers

Height and width of the filters, specified as a vector of two positive integers `[h  w]`, where `h` is the height and `w` is the width. `FilterSize` defines the size of the local regions to which the neurons connect in the input.

If you set `FilterSize` using an input argument, then you can specify `FilterSize` as scalar to use the same value for both dimensions.

Example: `[5 5]` specifies filters of height 5 and width 5.

NumFilters — Number of filters

positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This
parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96

**Stride — Step size for traversing input**  
[1 1] (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector \([a \ b]\) of two positive integers, where \(a\) is the vertical step size and \(b\) is the horizontal step size. When creating the layer, you can specify Stride as a scalar to use the same value for both step sizes.

Example: \([2 \ 3]\) specifies a vertical step size of 2 and a horizontal step size of 3.

**CroppingMode — Method to determine cropping size**  
'manual' (default) | 'same'

Method to determine cropping size, specified as 'manual' or 'same'.

The software automatically sets the value of CroppingMode based on the 'Cropping' value you specify when creating the layer.

- If you set the 'Cropping' option to a numeric value, then the software automatically sets the CroppingMode property of the layer to 'manual'.
- If you set the 'Cropping' option to 'same', then the software automatically sets the CroppingMode property of the layer to 'same' and set the cropping so that the output size equals \(\text{inputSize} \times \text{Stride}\), where \(\text{inputSize}\) is the height and width of the layer input.

To specify the cropping size, use the 'Cropping' option of **transposedConv2dLayer**.

**CroppingSize — Output size reduction**  
[0 0 0 0] (default) | vector of four nonnegative integers

Output size reduction, specified as a vector of four nonnegative integers \([t \ b \ l \ r]\), where \(t\), \(b\), \(l\), \(r\) are the amounts to crop from the top, bottom, left, and right, respectively.

To specify the cropping size manually, use the 'Cropping' option of **transposedConv2dLayer**.

Example: \([0 \ 1 \ 0 \ 1]\)
**Cropping — Output size reduction**

\[ [0 \ 0] \text{ (default)} \mid \text{vector of two nonnegative integers} \]

**Note** Cropping property will be removed in a future release. Use CroppingSize instead. To specify the cropping size manually, use the 'Cropping' option of transposedConv2dLayer.

Output size reduction, specified as a vector of two nonnegative integers \([a \ b]\), where \(a\) corresponds to the cropping from the top and bottom and \(b\) corresponds to the cropping from the left and right.

To specify the cropping size manually, use the 'Cropping' option of transposedConv2dLayer.

Example: \([0 \ 1]\)

**NumChannels — Number of channels for each filter**

'auto' (default) | integer

Number of channels for each filter, specified as 'NumChannels' and 'auto' or an integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

**Parameters and Initialization**

**WeightsInitializer — Function to initialize weights**

'glorot' (default) | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:

- 'glorot' - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance \(2/(\text{numIn} + \text{numOut})\), where \(\text{numIn} = \text{FilterSize}(1) \times \text{FilterSize}(2) \times \text{NumChannels}\) and \(\text{numOut} = \text{FilterSize}(1) \times \text{FilterSize}(2) \times \text{NumFilters}\).
• 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance $2/\text{numIn}$, where $\text{numIn} = \text{FilterSize}(1) \times \text{FilterSize}(2) \times \text{NumChannels}$.

• 'narrow-normal' - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

• 'zeros' - Initialize the weights with zeros.

• 'ones' - Initialize the weights with ones.

• Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form $\text{weights} = \text{func(sz)}$, where $\text{sz}$ is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the Weights property is empty.

Data Types: char | string | function_handle

**BiasInitializer — Function to initialize bias**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

• 'zeros' - Initialize the bias with zeros.

• 'ones' - Initialize the bias with ones.

• 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

• Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form $\text{bias} = \text{func(sz)}$, where $\text{sz}$ is the size of the bias.

The layer only initializes the bias when the Bias property is empty.

Data Types: char | string | function_handle

**Weights — Layer weights**

[] (default) | numeric array

Layer weights for the convolutional layer, specified as a FilterSize(1)-by-
FilterSize(2)-by-NumFilters-by-NumChannels array.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the Weights property of the layer. When training a network, if the
Weights property of the layer is nonempty, then trainNetwork uses the Weights property as the initial value. If the Weights property is empty, then trainNetwork uses the initializer specified by the WeightsInitializer property of the layer.

Data Types: single | double

**Bias — Layer biases**  
[] (default) | numeric array

Layer biases for the convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if Bias is nonempty, then trainNetwork uses the Bias property as the initial value. If Bias is empty, then trainNetwork uses the initializer specified by BiasInitializer.

At training time, Bias is a 1-by-1-by-NumFilters array.

Data Types: single | double

**Learn Rate and Regularization**

**WeightLearnRateFactor — Learning rate factor for weights**  
1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if WeightLearnRateFactor is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

Example: 2

**BiasLearnRateFactor — Learning rate factor for biases**  
1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if BiasLearnRateFactor is 2, then the learning rate for the biases in the layer is twice the current global learning rate.
software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**WeightL2Factor — L2 regularization factor for weights**
1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**BiasL2Factor — L2 regularization factor for biases**
0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**Layer**

**Name — Layer name**
'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: `char` | `string`

**NumInputs — Number of inputs**
1 (default)
Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**

`{'in'}` (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**

`{'out'}` (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create Transposed Convolutional Layer**

Create a transposed convolutional layer with 96 filters, each with a height and width of 11. Use a stride of 4 in the horizontal and vertical directions.

```matlab
code = transposedConv2dLayer(11,96,'Stride',4);
```

**Compatibility Considerations**

**Default weights initialization is Glorot**

*Behavior changed in R2019a*
Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'WeightsInitializer' option of the layer to 'narrow-normal'.

**Cropping property of TransposedConvolution2DLayer will be removed**

*Not recommended starting in R2019a*

Cropping property of TransposedConvolution2DLayer will be removed, use CroppingSize instead. To update your code, replace all instances of the Cropping property with CroppingSize.

**References**


**See Also**

averagePooling2dLayer | convolution2dLayer | maxPooling2dLayer | transposedConv2dLayer

**Topics**

“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”
“Compare Layer Weight Initializers”
Introduced in R2017b
transposedConv3dLayer

Transposed 3-D convolution layer

Syntax

layer = transposedConv3dLayer(filterSize,numFilters)
layer = transposedConv3dLayer(filterSize,numFilters,Name,Value)

Description

A transposed 3-D convolution layer upsamples three-dimensional feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

layer = transposedConv3dLayer(filterSize,numFilters) returns a transposed 3-D convolution layer and sets the FilterSize and NumFilters properties.

layer = transposedConv3dLayer(filterSize,numFilters,Name,Value) returns a transposed 3-D convolutional layer and specifies additional options using one or more name-value pair arguments.

Examples

Create Transposed 3-D Convolutional Layer

Create a transposed 3-D convolutional layer with 32 filters, each with a height, width, and depth of 11. Use a stride of 4 in the horizontal and vertical directions and 2 along the depth.

layer = transposedConv3dLayer(11,32,{'Stride',[4 4 2]})

layer = TransposedConvolution3DLayer with properties:
Name: ''

Hyperparameters
  FilterSize: [11 11 11]
  NumChannels: 'auto'
  NumFilters: 32
  Stride: [4 4 2]
  CroppingMode: 'manual'
  CroppingSize: [2x3 double]

Learnable Parameters
  Weights: []
  Bias: []

Show all properties

**Input Arguments**

**filterSize — Height, width, and depth of filters**
vector of three positive integers

Height, width, and depth of the filters, specified as a vector \([h \ w \ d]\) of three positive integers, where \(h\) is the height, \(w\) is the width, and \(d\) is the depth. **FilterSize** defines the size of the local regions to which the neurons connect in the input.

If you set **FilterSize** using an input argument, then you can specify **FilterSize** as scalar to use the same value for all three dimensions.

Example: \([5 \ 5 \ 5]\) specifies filters with a height, width, and depth of 5.

**numFilters — Number of filters**
positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'Cropping', 1

Transposed Convolution

Stride — Step size for traversing input
[1 1 1] (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector [a b c] of three positive integers, where a is the vertical step size, b is the horizontal step size, and c is the step size along the depth. When creating the layer, you can specify Stride as a scalar to use the same value for step sizes in all three directions.

Example: [2 3 1] specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

Cropping — Output size reduction
0 (default) | 'same' | vector of nonnegative integers | matrix of nonnegative integers

Output size reduction, specified as one of the following:

• 'same' - Set the cropping so that the output size equals inputSize .* Stride, where inputSize is the height, width, and depth of the layer input. If you set the 'Cropping' option to 'same', then the software automatically sets the CroppingMode property of the layer to 'same'.

The software trims an equal amount from the top and bottom, the left and right, and the front and back, if possible. If the vertical crop amount has an odd value, then the software trims an extra row from the bottom. If the horizontal crop amount has an odd value, then the software trims an extra column from the right. If the depth crop amount has an odd value, then the software trims an extra plane from the back.

• A positive integer - Crop the specified amount of data from all the edges.

• A vector of nonnegative integers [a b c] - Crop a from the top and bottom, crop b from the left and right, and crop c from the front and back.
• a matrix of nonnegative integers \([t \ l \ f; \ b \ r \ bk]\) of nonnegative integers — Crop \(t, l, f, b, r, bk\) from the top, left, front, bottom, right, and back of the input, respectively.

Example: \([1 \ 2 \ 2]\)

**NumChannels — Number of channels for each filter**

'auto' (default) | positive integer

Number of channels for each filter, specified as 'NumChannels' and 'auto' or a positive integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

**Parameters and Initialization**

**WeightsInitializer — Function to initialize weights**

'glorot' (default) | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:

• 'glorot' - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance \(2/(\text{numIn} + \text{numOut})\), where \(\text{numIn} = \text{filterSize}(1) \times \text{filterSize}(2) \times \text{filterSize}(3) \times \text{NumChannels}, \text{numOut} = \text{filterSize}(1) \times \text{filterSize}(2) \times \text{filterSize}(3) \times \text{numFilters}\), and \(\text{NumChannels}\) is the number of input channels.

• 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance \(2/\text{numIn}\), where \(\text{numIn} = \text{filterSize}(1) \times \text{filterSize}(2) \times \text{filterSize}(3) \times \text{NumChannels}\) and \(\text{NumChannels}\) is the number of input channels.

• 'narrow-normal' - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

• 'zeros' - Initialize the weights with zeros.

• 'ones' - Initialize the weights with ones.

• Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form \(\text{weights} = \text{func}(\text{sz})\), where
sz is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the Weights property is empty.

Data Types: char | string | function_handle

**BiasInitializer — Function to initialize bias**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

- 'zeros' - Initialize the bias with zeros.
- 'ones' - Initialize the bias with ones.
- 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form bias = func(sz), where sz is the size of the bias.

The layer only initializes the bias when the Bias property is empty.

Data Types: char | string | function_handle

**Weights — Layer weights**

[] (default) | numeric array

Layer weights for the transposed convolutional layer, specified as a numeric array.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the Weights property of the layer. When training a network, if the Weights property of the layer is nonempty, then trainNetwork uses the Weights property as the initial value. If the Weights property is empty, then trainNetwork uses the initializer specified by the WeightsInitializer property of the layer.

At training time, Weights is a FilterSize(1)-by-FilterSize(2)-by-FilterSize(3)-by-numFilters-by-NumChannels array.

Data Types: single | double

**Bias — Layer biases**

[] (default) | numeric array
Layer biases for the transposed convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if Bias is nonempty, then `trainNetwork` uses the Bias property as the initial value. If Bias is empty, then `trainNetwork` uses the initializer specified by BiasInitializer.

At training time, Bias 1-by-1-by-1-by-numFilters array.

Data Types: single | double

Learn Rate and Regularization

WeightLearnRateFactor — Learning rate factor for weights

1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if WeightLearnRateFactor is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

BiasLearnRateFactor — Learning rate factor for biases

1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if BiasLearnRateFactor is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

WeightL2Factor — L2 regularization factor for weights

1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if WeightL2Factor is 2, then
the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the trainingOptions function.

Example: 2

**BiasL2Factor — L2 regularization factor for biases**

0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if BiasL2Factor is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the trainingOptions function.

Example: 2

**Layer**

**Name — Layer name**

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**Output Arguments**

**layer — Transposed 3-D convolution layer**

TransposedConvolution3DLayer object

Transposed 3-D convolution layer, returned as a TransposedConvolution3dLayer object.
References


See Also
SoftmaxLayer | TransposedConvolution3dLayer | averagePooling3dLayer | maxPooling3dLayer | transposedConv2dLayer

Topics
“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a
TransposedConvolution3dLayer

Transposed 3-D convolution layer

Description

A transposed 3-D convolution layer upsamples three-dimensional feature maps.

This layer is sometimes incorrectly known as a "deconvolution" or "deconv" layer. This layer is the transpose of convolution and does not perform deconvolution.

Creation

Create a transposed convolution 3-D output layer using transposedConv3dLayer.

Properties

Transposed Convolution

FilterSize — Height, width, and depth of filters

vector of three positive integers

Height, width, and depth of the filters, specified as a vector $[h \ w \ d]$ of three positive integers, where $h$ is the height, $w$ is the width, and $d$ is the depth. FilterSize defines the size of the local regions to which the neurons connect in the input.

When creating the layer, you can specify FilterSize as a scalar to use the same value for the height, width, and depth.

Example: $[5 \ 5 \ 5]$ specifies filters with a height, width, and depth of 5.

NumFilters — Number of filters

positive integer

Number of filters, specified as a positive integer. This number corresponds to the number of neurons in the convolutional layer that connect to the same region in the input. This
parameter determines the number of channels (feature maps) in the output of the convolutional layer.

Example: 96

**Stride — Step size for traversing input**

[1 1 1] (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector \([a \ b \ c]\) of three positive integers, where \(a\) is the vertical step size, \(b\) is the horizontal step size, and \(c\) is the step size along the depth. When creating the layer, you can specify **Stride** as a scalar to use the same value for step sizes in all three directions.

Example: \([2 \ 3 \ 1]\) specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

**CroppingMode — Method to determine cropping size**

'manual' (default) | 'same'

Method to determine cropping size, specified as 'manual' or 'same'.

The software automatically sets the value of **CroppingMode** based on the 'Cropping' value you specify when creating the layer.

- If you set the 'Cropping' option to a numeric value, then the software automatically sets the **CroppingMode** property of the layer to 'manual'.
- If you set the 'Cropping' option to 'same', then the software automatically sets the **CroppingMode** property of the layer to 'same' and set the cropping so that the output size equals \(\text{inputSize} \times \text{Stride}\), where \(\text{inputSize}\) is the height, width, and depth of the layer input.

To specify the cropping size, use the 'Cropping' option of **transposedConv3dLayer**.

**CroppingSize — Output size reduction**

[0 0 0; 0 0 0] (default) | matrix of nonnegative integers

Output size reduction, specified as a matrix of nonnegative integers \([t \ l \ f; b \ r \ bk]\), \(t, l, f, b, r, bk\) are the amounts to crop from the top, left, front, bottom, right, and back of the input, respectively.

To specify the cropping size manually, use the 'Cropping' option of **transposedConv2dLayer**.
Example: [0 1 0 1 0 1]

**NumChannels** — Number of channels for each filter

`'auto'` (default) | integer

Number of channels for each filter, specified `'auto'` or an integer.

This parameter must be equal to the number of channels of the input to this convolutional layer. For example, if the input is a color image, then the number of channels for the input must be 3. If the number of filters for the convolutional layer prior to the current layer is 16, then the number of channels for this layer must be 16.

### Parameters and Initialization

**WeightsInitializer** — Function to initialize weights

`'glorot'` (default) | `'he'` | `'narrow-normal'` | `'zeros'` | `'ones'` | function handle

Function to initialize the weights, specified as one of the following:

- `'glorot'` - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance $2/(\text{numIn} + \text{numOut})$, where $\text{numIn} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{FilterSize}(3)\times\text{NumChannels}$ and $\text{numOut} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{FilterSize}(3)\times\text{NumFilters}$.
- `'he'` - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance $2/\text{numIn}$, where $\text{numIn} = \text{FilterSize}(1)\times\text{FilterSize}(2)\times\text{FilterSize}(3)\times\text{NumChannels}$.
- `'narrow-normal'` - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- `'zeros'` - Initialize the weights with zeros.
- `'ones'` - Initialize the weights with ones.
- Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the `Weights` property is empty.

**Data Types:** char | string | function_handle
**BiasInitializer — Function to initialize bias**

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

- 'zeros' - Initialize the bias with zeros.
- 'ones' - Initialize the bias with ones.
- 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form `bias = func(sz)`, where `sz` is the size of the bias.

The layer only initializes the bias when the `Bias` property is empty.

Data Types: char | string | function_handle

**Weights — Layer weights**

[ ] (default) | numeric array

Layer weights for the transposed convolutional layer, specified as a numeric array.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the `Weights` property of the layer. When training a network, if the `Weights` property of the layer is nonempty, then `trainNetwork` uses the `Weights` property as the initial value. If the `Weights` property is empty, then `trainNetwork` uses the initializer specified by the `WeightsInitializer` property of the layer.

At training time, `Weights` is a `FilterSize(1)`-by-`FilterSize(2)`-by-`FilterSize(3)`-by-`NumFilters`-by-`NumChannels` array.

Data Types: single | double

**Bias — Layer biases**

[ ] (default) | numeric array

Layer biases for the transposed convolutional layer, specified as a numeric array.

The layer biases are learnable parameters. When training a network, if `Bias` is nonempty, then `trainNetwork` uses the `Bias` property as the initial value. If `Bias` is empty, then `trainNetwork` uses the initializer specified by `BiasInitializer`.

At training time, `Bias` is a 1-by-1-by-1-by-`NumFilters` array.
Learn Rate and Regularization

**WeightLearnRateFactor — Learning rate factor for weights**
1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**BiasLearnRateFactor — Learning rate factor for biases**
1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

Example: 2

**WeightL2Factor — L2 regularization factor for weights**
1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2
**BiasL2Factor — L2 regularization factor for biases**
0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

**Layer**

**Name — Layer name**
' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**
{"in"} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double
OutputNames — Output names

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

Examples

Create Transposed 3-D Convolutional Layer

Create a transposed 3-D convolutional layer with 32 filters, each with a height, width, and depth of 11. Use a stride of 4 in the horizontal and vertical directions and 2 along the depth.

layer = transposedConv3dLayer(11,32,'Stride',[4 4 2])

layer =
  TransposedConvolution3DLayer with properties:

    Name: ''

    Hyperparameters
      FilterSize: [11 11 11]
      NumChannels: 'auto'
      NumFilters: 32
      Stride: [4 4 2]
      CroppingMode: 'manual'
      CroppingSize: [2x3 double]

    Learnable Parameters
      Weights: []
      Bias: []

Show all properties
References


See Also
averagePooling3dLayer | convolution3dLayer | maxPooling3dLayer | transposedConv2dLayer | transposedConv3dLayer

Topics
“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a
**crop2dLayer**

2-D crop layer

**Description**

A 2-D crop layer applies 2-D cropping to the input.

There are two inputs to this layer:

- `'in'` — The feature map that will be cropped
- `'ref'` — A reference layer used to determine the size, `[height width]`, of the cropped output

Once you create this layer, you can add it to a `layerGraph` to make serial connections between layers. To connect the crop layer to other layers, call `connectLayers` and specify the input names. The `connectLayers` function returns a connected `LayerGraph` object ready to train a network.
Creation

Syntax

layer = crop2dLayer(Mode)
layer = crop2dLayer(Location)
layer = crop2dLayer(___,'Name',Name)
**Description**

`layer = crop2dLayer(Mode)` returns a layer that crops an input feature map, and sets the `Mode` property.

`layer = crop2dLayer(Location)` returns a layer that crops an input feature map using a rectangular window, and sets the `Location` property that indicates the position of the window.

`layer = crop2dLayer(___,'Name',Name)` creates a layer for cropping and sets the optional `Name` property.

**Properties**

**Mode — Cropping mode**

'*centercrop' (default) | 'custom'*

Cropping mode, specified as 'centercrop' or 'custom'.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'centercrop'</td>
<td>The location of the cropping window is the center of the input feature map.</td>
</tr>
<tr>
<td>'custom'</td>
<td>The location of the cropping window is based on the <code>Location</code> property. This value is automatically set when the <code>Location</code> property is specified as a 2-element row vector.</td>
</tr>
</tbody>
</table>

Data Types: `char`

**Location — Cropping window location**

'*auto' (default) | 2-element row vector*

Cropping window location, specified as 'auto' or a 2-element row vector.
Location Description
2-element row vector in the format \([x \ y]\) The upper-left corner of the cropping window is at the location \([x \ y]\) of the input feature map. \(x\) indicates the location in the horizontal direction and \(y\) is the vertical direction.

'auto' The cropping window is located at the center of the input feature map. This value is automatically set when the Mode property is specified as 'centercrop'.

Name — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

NumInputs — Number of inputs

2 (default)

Number of inputs of the layer. This layer has two inputs.

Data Types: double

InputNames — Input names

{'in' 'ref'} (default)

Input names of the layer. This layer has two inputs, named 'in' and 'ref'.

Data Types: cell

NumOutputs — Number of outputs

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

OutputNames — Output names

{'out'} (default)
Output names of the layer. This layer has a single output only.

Data Types: cell

Examples

Create 2-D Crop Layer

Create a 2-D crop layer and connect both of the inputs using a `layerGraph` object.

Create the layers.

```matlab
layers = [imageInputLayer([32 32 3],'Name','image')
crop2dLayer('centercrop','Name','crop')]
```

```matlab
layers = 2x1 Layer array with layers:

1   'image'   Image Input   32x32x3 images with 'zerocenter' normalization
2   'crop'    Crop 2D       center crop
```

Create a `layerGraph`. The first input of `crop2dLayer` is automatically connected to the first output of the image input layer.

```matlab
lgraph = layerGraph(layers)
lgraph =
    LayerGraph with properties:
        Layers: [2x1 nnet.cnn.layer.Layer]
            Connections: [1x2 table]
                InputNames: {'image'}
            OutputNames: {1x0 cell}
```

Connect the image input layer to the "ref" input of the 2-D crop layer.

```matlab
lgraph = connectLayers(lgraph,'image','crop/ref')
lgraph =
    LayerGraph with properties:
```
Layers: [2×1 nnet.cnn.layer.Layer]
Connections: [2×2 table]
InputNames: {'image'}
OutputNames: {1×0 cell}

Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

deeplabv3plusLayers | fcnLayers | layerGraph | pixelClassificationLayer |
segnetLayers | semanticseg | trainNetwork | unetLayers

Topics

“Getting Started With Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
“Deep Learning in MATLAB”

Introduced in R2017b
crop3dLayer

3-D crop layer

Description

A 3-D crop layer crops a 3-D volume to the size of the input feature map.

Specify the number of inputs to the layer when you create it. The inputs to the layer have the names 'in' and 'ref'. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers. All inputs to a 3-D crop layer must have the same number of dimensions.

Creation

Syntax

layer = crop3dLayer
layer = crop3dLayer([X Y Z])
layer = crop3dLayer(__,'Name',Name)

Description

layer = crop3dLayer creates a 3-D crop layer that crops an input feature map from the center of the feature map. The size of the cropped region is equal to the size of a second reference input feature map.

layer = crop3dLayer([X Y Z]) also sets the cropLocation property with the (X,Y,Z) coordinate of the crop window. X is the coordinate in the horizontal direction, Y is the coordinate in the vertical direction, and Z is the coordinate in the depth direction.

layer = crop3dLayer(__,'Name',Name) also sets the Name property. To create a network containing a 3-D crop layer, you must specify a layer name.
Properties

Crop

cropLocation — Crop location

'decentrocrop' (default) | three-element numeric vector

Crop location, specified as 'centercrop' or a three-element numeric vector representing the (x,y,z) coordinate of the crop window.

Layer

Name — Layer name

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

NumInputs — Number of inputs

2 (default)

Number of inputs of the layer. This layer accepts two inputs.

Data Types: double

InputNames — Input names

{'in','ref'} (default)

Input names of the layer, specified as {'in','ref'}. This layer accepts two inputs.

Data Types: cell

NumOutputs — Number of outputs

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double
OutputNames — Output names
{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

Examples

Create and Connect 3-D Crop Layer

Create a 3-D crop layer and connect both of its inputs using a layerGraph object.

```matlab
layers = [cell(32 32 32 3),'Name','image'
     convolution3dLayer(3,16,'Padding','same','Name','conv'
     crop3dLayer('Name','crop'
     concatenationLayer(4,2,'Name','concat')
]
layers =
4x1 Layer array with layers:
 1   'image'    3-D Image Input   32x32x32x3 images with 'zerocenter' normalization
 2   'conv'     Convolution       16 3x3x3 convolutions with stride [1 1 1] and padding 'same'
 3   'crop'     Crop 3D           center crop
 4   'concat'   Concatenation     Concatenation of 2 inputs along dimension 4

Create a layer graph. The first input of the 3-D crop layer is automatically connected to
the output of the 3-D convolutional layer.

lgraph = layerGraph(layers);

Add a max pooling layer to the layer graph.

maxPool = maxPooling3dLayer(2,'stride',2,'Name','pool');
lgraph = addLayers(lgraph,maxPool);
lgraph = connectLayers(lgraph,'image','pool');

Connect the second input of the crop layer to the output of the max pooling layer.

lgraph = connectLayers(lgraph,'pool','crop/ref');
Concatenate the crop layer output and the max pooling layer output.

\[
l_{\text{graph}} = \text{connectLayers}(l_{\text{graph}}, 'pool', 'concat/in2');
\]

Display the layer graph.

\[
\text{plot}(l_{\text{graph}})
\]

**See Also**
crop2dLayer | layerGraph | trainNetwork
Topics
“Deep Learning in MATLAB”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019b
CrossChannelNormalizationLayer

Channel-wise local response normalization layer

Description

A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.

Creation

Syntax

layer = crossChannelNormalizationLayer(windowChannelSize)
layer = crossChannelNormalizationLayer(windowChannelSize,Name,Value)

Description

layer = crossChannelNormalizationLayer(windowChannelSize) creates a channel-wise local response normalization layer and sets the WindowChannelSize property.

layer = crossChannelNormalizationLayer(windowChannelSize,Name,Value) sets the optional properties WindowChannelSize, Alpha, Beta, K, and Name using name-value pairs. For example, crossChannelNormalizationLayer(5,'K',1) creates a local response normalization layer for channel-wise normalization with a window size of 5 and K hyperparameter 1. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Properties

Cross-Channel Normalization

WindowChannelSize — Size of the channel window
positive integer

Size of the channel window, which controls the number of channels that are used for the normalization of each element, specified as a positive integer.

If WindowChannelSize is even, then the window is asymmetric. The software looks at the previous floor((w-1)/2) channels and the following floor(w/2) channels. For example, if WindowChannelSize is 4, then the layer normalizes each element by its neighbor in the previous channel and by its neighbors in the next two channels.

Example: 5

Alpha — $\alpha$ hyperparameter in normalization
0.0001 (default) | numeric scalar

$\alpha$ hyperparameter in the normalization (the multiplier term), specified as a numeric scalar.

Example: 0.0002

Beta — $\beta$ hyperparameter in normalization
0.75 (default) | numeric scalar

$\beta$ hyperparameter in the normalization, specified as a numeric scalar. The value of Beta must be greater than or equal to 0.01.

Example: 0.8

K — $K$ hyperparameter in the normalization
2 (default) | numeric scalar

$K$ hyperparameter in the normalization, specified as a numeric scalar. The value of K must be greater than or equal to $10^{-5}$.

Example: 2.5
Layer

**Name — Layer name**
' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and **Name** is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**
{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

Examples
Create Local Response Normalization Layer

Create a local response normalization layer for channel-wise normalization, where a window of five channels normalizes each element, and the additive constant for the normalizer \( K \) is 1.

\[
\text{layer} = \text{crossChannelNormalizationLayer}(5,'K',1)
\]

\[
\text{layer} = \text{CrossChannelNormalizationLayer with properties:}
\]

- Name: ''
- Hyperparameters
  - WindowChannelSize: 5
  - Alpha: 1.0000e-04
  - Beta: 0.7500
  - K: 1

Include a local response normalization layer in a Layer array.

\[
\text{layers} = [\ldots, \text{imageInputLayer}([28 28 1]), \text{convolution2dLayer}(5,20), \text{reluLayer}, \text{crossChannelNormalizationLayer}(3), \text{fullyConnectedLayer}(10), \text{softmaxLayer}, \text{classificationLayer}]
\]

\[
\text{layers} = \text{7x1 Layer array with layers:}
\]

- 1 '' Image Input
- 2 '' Convolution
- 3 '' ReLU
- 4 '' Cross Channel Normalization
- 5 '' Fully Connected
- 6 '' Softmax
- 7 '' Classification Output

28x28x1 images with 'zerocenter' normalization
20 5x5 convolutions with stride [1 1] and ReLU
Cross channel normalization with 3 channels
10 fully connected layer
softmax
crossentropyex
More About

Local Response Normalization

A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.

This layer performs a channel-wise local response normalization. It usually follows the ReLU activation layer. This layer replaces each element with a normalized value it obtains using the elements from a certain number of neighboring channels (elements in the normalization window). That is, for each element \( x \) in the input, \texttt{trainNetwork} computes a normalized value \( x' \) using

\[
    x' = \frac{x}{\left(K + \alpha \cdot ss \cdot \text{windowChannelSize} \right)^\beta}
\]

where \( K, \alpha, \) and \( \beta \) are the hyperparameters in the normalization, and \( ss \) is the sum of squares of the elements in the normalization window [1]. You must specify the size of the normalization window using the \texttt{windowChannelSize} argument of the \texttt{crossChannelNormalizationLayer} function. You can also specify the hyperparameters using the \texttt{Alpha}, \texttt{Beta}, and \texttt{K} name-value pair arguments.

The previous normalization formula is slightly different than what is presented in [1]. You can obtain the equivalent formula by multiplying the \texttt{alpha} value by the \texttt{windowChannelSize}.

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.
See Also
averagePooling2dLayer | convolution2dLayer | maxPooling2dLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a
dropoutLayer

Dropout layer

Description

A dropout layer randomly sets input elements to zero with a given probability.

Creation

Syntax

layer = dropoutLayer
layer = dropoutLayer(probability)
layer = dropoutLayer(___,'Name',Name)

Description

layer = dropoutLayer creates a dropout layer.

layer = dropoutLayer(probability) creates a dropout layer and sets the Probability property.

layer = dropoutLayer(___,'Name',Name) sets the optional Name property using a name-value pair and any of the arguments in the previous syntaxes. For example, dropoutLayer(0.4,'Name','drop1') creates a dropout layer with dropout probability 0.4 and name 'drop1'. Enclose the property name in single quotes.
Properties

Dropout

**Probability — Probability to drop out input elements**

0.5 (default) | numeric scalar in the range 0 to 1

Probability for dropping out input elements, specified as a numeric scalar in the range 0–1.

At training time, the layer randomly sets input elements to zero given by the dropout mask `rand(size(X))<Probability`, where `X` is the layer input and then scales the remaining elements by `1/(1-Probability)`. This operation effectively changes the underlying network architecture between iterations and helps prevent the network from overfitting [1], [2]. A higher number results in more elements being dropped during training. At prediction time, the output of the layer is equal to its input.

For image input, the layer applies a different mask for each channel of each image. For sequence input, the layer applies a different dropout mask for each time step of each sequence.

Example: `0.4`

Layer

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double
**InputNames — Input names**

{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**

**Create Dropout Layer**

Create a dropout layer with name 'drop1'.

```matlab
layer = dropoutLayer('Name','drop1')
```

```matlab
layer = DropoutLayer with properties:

    Name: 'drop1'

    Hyperparameters
    Probability: 0.5000
```

Include a dropout layer in a `Layer` array.

```matlab
layers = [ ... 
    imageInputLayer([28 28 1])
```
```plaintext
convolution2dLayer(5,20)
reluLayer
dropoutLayer
fullyConnectedLayer(10)
softmaxLayer
classificationLayer
```

```plaintext
layers =
7x1 Layer array with layers:
```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Image Input</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Convolution</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>ReLU</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Dropout</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Fully Connected</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Softmax</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Classification Output</td>
</tr>
</tbody>
</table>
```

More About

**Dropout Layer**

A dropout layer randomly sets input elements to zero with a given probability.

At training time, the layer randomly sets input elements to zero given by the dropout mask `rand(size(X))<Probability`, where `X` is the layer input and then scales the remaining elements by `1/(1-Probability)`. This operation effectively changes the underlying network architecture between iterations and helps prevent the network from overfitting [1], [2]. A higher number results in more elements being dropped during training. At prediction time, the output of the layer is equal to its input.

Similar to max or average pooling layers, no learning takes place in this layer.

For image input, the layer applies a different mask for each channel of each image. For sequence input, the layer applies a different dropout mask for each time step of each sequence.
References


Extended Capabilities

**GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

imageInputLayer | reluLayer

Topics

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a
fullyConnectedLayer

Fully connected layer

Description

A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.

Creation

Syntax

layer = fullyConnectedLayer(outputSize)
layer = fullyConnectedLayer(outputSize,Name,Value)

Description

layer = fullyConnectedLayer(outputSize) returns a fully connected layer and specifies the OutputSize property.

layer = fullyConnectedLayer(outputSize,Name,Value) sets the optional "Parameters and Initialization" on page 1-971, "Learn Rate and Regularization" on page 1-973, and Name properties using name-value pairs. For example, fullyConnectedLayer(10,'Name','fc1') creates a fully connected layer with an output size of 10 and the name 'fc1'. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Properties

Fully Connected

OutputSize — Output size
positive integer

Output size for the fully connected layer, specified as a positive integer.
Example: 10

InputSize — Input size
'auto' (default) | positive integer

Input size for the fully connected layer, specified as a positive integer or 'auto'. If InputSize is 'auto', then the software automatically determines the input size during training.

Parameters and Initialization

WeightsInitializer — Function to initialize weights
'glorot' (default) | 'he' | 'orthogonal' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the weights, specified as one of the following:

- 'glorot' - Initialize the weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance \(2/(\text{InputSize} + \text{OutputSize})\).
- 'he' - Initialize the weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance \(2/\text{InputSize}\).
- 'orthogonal' - Initialize the input weights with \(Q\), the orthogonal matrix given by the QR decomposition of \(Z = QR\) for a random matrix \(Z\) sampled from a unit normal distribution. [3]
- 'narrow-normal' - Initialize the weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- 'zeros' - Initialize the weights with zeros.
- 'ones' - Initialize the weights with ones.
• Function handle - Initialize the weights with a custom function. If you specify a function handle, then the function must be of the form weights = func(sz), where sz is the size of the weights. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the weights when the Weights property is empty.

Data Types: char | string | function_handle

BiasInitializer — Function to initialize bias

'zeros' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

• 'zeros' - Initialize the bias with zeros.
• 'ones' - Initialize the bias with ones.
• 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
• Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form bias = func(sz), where sz is the size of the bias.

The layer only initializes the bias when the Bias property is empty.

Data Types: char | string | function_handle

Weights — Layer weights

[] (default) | matrix

Layer weights, specified as a matrix.

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the Weights property of the layer. When training a network, if the Weights property of the layer is nonempty, then trainNetwork uses the Weights property as the initial value. If the Weights property is empty, then trainNetwork uses the initializer specified by the WeightsInitializer property of the layer.

At training time, Weights is an OutputSize-by-InputSize matrix.

Data Types: single | double

Bias — Layer biases

[] (default) | matrix
Layer biases, specified as a matrix.

The layer biases are learnable parameters. When training a network, if Bias is nonempty, then trainNetwork uses the Bias property as the initial value. If Bias is empty, then trainNetwork uses the initializer specified by BiasInitializer.

At training time, Bias is an OutputSize-by-1 matrix.

Data Types: single | double

**Learn Rate and Regularization**

**WeightLearnRateFactor — Learning rate factor for weights**

1 (default) | nonnegative scalar

Learning rate factor for the weights, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the weights in this layer. For example, if WeightLearnRateFactor is 2, then the learning rate for the weights in this layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

Example: 2

**BiasLearnRateFactor — Learning rate factor for biases**

1 (default) | nonnegative scalar

Learning rate factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if BiasLearnRateFactor is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

Example: 2

**WeightL2Factor — L2 regularization factor for weights**

1 (default) | nonnegative scalar

L2 regularization factor for the weights, specified as a nonnegative scalar.
The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the weights in this layer. For example, if \texttt{WeightL2Factor} is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the \texttt{trainingOptions} function.

Example: 2

**BiasL2Factor — L2 regularization factor for biases**

0 (default) | nonnegative scalar

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if \texttt{BiasL2Factor} is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the \texttt{trainingOptions} function.

Example: 2

**Layer**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and \texttt{Name} is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'}in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: `cell`

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

**OutputNames — Output names**
`{'out'}` (default)

Output names of the layer. This layer has a single output only.

Data Types: `cell`

**Examples**

**Create Fully Connected Layer**

Create a fully connected layer with an output size of 10 and the name `'fc1'`.

```matlab
layer = fullyConnectedLayer(10,'Name','fc1')
```

```matlab
layer = 
    FullyConnectedLayer with properties:
        Name: 'fc1'

    Hyperparameters
        InputSize: 'auto'
        OutputSize: 10

    Learnable Parameters
        Weights: []
        Bias: []
```

Include a fully connected layer in a Layer array.
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
];

layers =
    7x1 Layer array with layers:
    1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
    2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0  0  0]
    3   ''   ReLU                    ReLU
    4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
    5   ''   Fully Connected         10 fully connected layer
    6   ''   Softmax                 softmax
    7   ''   Classification Output   crossentropyex

**Specify Initial Weights and Biases in Fully Connected Layer**

To specify the weights and bias initializer functions, use the `WeightsInitializer` and `BiasInitializer` properties respectively. To specify the weights and biases directly, use the `Weights` and `Bias` properties respectively.

**Specify Initialization Function**

Create a fully connected layer with an output size of 10 and specify the weights initializer to be the He initializer.

```matlab
outputSize = 10;
layer = fullyConnectedLayer(outputSize,'WeightsInitializer','he')
```

layer =
    FullyConnectedLayer with properties:
        Name: ''
        Hyperparameters
            InputSize: 'auto'
            OutputSize: 10
Learnable Parameters
   Weights: []
   Bias: []

Show all properties

Note that the Weights and Bias properties are empty. At training time, the software initializes these properties using the specified initialization functions.

Specify Custom Initialization Function

To specify your own initialization function for the weights and biases, set the WeightsInitializer and BiasInitializer properties to a function handle. For these properties, specify function handles that take the size of the weights and biases as input and output the initialized value.

Create a fully connected layer with output size 10 and specify initializers that sample the weights and biases from a Gaussian distribution with a standard deviation of 0.0001.

```matlab
outputSize = 10;
weightsInitializationFcn = @(sz) rand(sz) * 0.0001;
biasInitializationFcn = @(sz) rand(sz) * 0.0001;

layer = fullyConnectedLayer(outputSize, ...  
    'WeightsInitializer',@(sz) rand(sz) * 0.0001, ...  
    'BiasInitializer',@(sz) rand(sz) * 0.0001)
```

Layer Properties:
- **Name**: ''
- **Hyperparameters**
  - **InputSize**: 'auto'
  - **OutputSize**: 10

Learnable Parameters
   **Weights**: []
   **Bias**: []

Show all properties
Again, the Weights and Bias properties are empty. At training time, the software initializes these properties using the specified initialization functions.

**Specify Weights and Bias Directly**

Create a fully connected layer with an output size of 10 and set the weights and bias to \( W \) and \( b \) in the MAT file FCWeights.mat respectively.

```matlab
outputSize = 10;
load FCWeights

layer = fullyConnectedLayer(outputSize, ...
    'Weights',W, ...
    'Bias',b)
```

Here, the Weights and Bias properties contain the specified values. At training time, if these properties are non-empty, then the software uses the specified values as the initial weights and biases. In this case, the software does not use the initializer functions.

**More About**

**Fully Connected Layer**

A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.
The convolutional (and down-sampling) layers are followed by one or more fully connected layers.

As the name suggests, all neurons in a fully connected layer connect to all the neurons in the previous layer. This layer combines all of the features (local information) learned by the previous layers across the image to identify the larger patterns. For classification problems, the last fully connected layer combines the features to classify the images. This is the reason that the outputSize argument of the last fully connected layer of the network is equal to the number of classes of the data set. For regression problems, the output size must be equal to the number of response variables.

You can also adjust the learning rate and the regularization parameters for this layer using the related name-value pair arguments when creating the fully connected layer. If you choose not to adjust them, then trainNetwork uses the global training parameters defined by the trainingOptions function. For details on global and layer training options, see “Set Up Parameters and Train Convolutional Neural Network”.

A fully connected layer multiplies the input by a weight matrix \( W \) and then adds a bias vector \( b \).

If the input to the layer is a sequence (for example, in an LSTM network), then the fully connected layer acts independently on each time step. For example, if the layer before the fully connected layer outputs an array \( X \) of size \( D \)-by-\( N \)-by-\( S \), then the fully connected layer outputs an array \( Z \) of size \( \text{outputSize} \)-by-\( N \)-by-\( S \). At time step \( t \), the corresponding entry of \( Z \) is \( WX_t + b \), where \( X_t \) denotes time step \( t \) of \( X \).

**Compatibility Considerations**

**Default weights initialization is Glorot**

*Behavior changed in R2019a*

Starting in R2019a, the software, by default, initializes the layer weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'WeightsInitializer' option of the layer to 'narrow-normal'.

1-979
References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
batchNormalizationLayer | convolution2dLayer | reluLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Compare Layer Weight Initializers”
“List of Deep Learning Layers”

Introduced in R2016a
**imageInputLayer**

__Image input layer__

**Description**

An image input layer inputs 2-D images to a network and applies data normalization.

For 3-D image input, use `image3dInputLayer`.

**Creation**

**Syntax**

```matlab
layer = imageInputLayer(inputSize)
layer = imageInputLayer(inputSize,Name,Value)
```

**Description**

`layer = imageInputLayer(inputSize)` returns an image input layer and specifies the `InputSize` property.

`layer = imageInputLayer(inputSize,Name,Value)` sets the optional properties on page 1-981 using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

**Properties**

**Image Input**

*InputSize — Size of the input*

row vector of integers
Size of the input data, specified as a row vector of integers \([h \ w \ c]\), where \(h\), \(w\), and \(c\) correspond to the height, width, and number of channels respectively.

- For grayscale images, specify a vector with \(c\) equal to 1.
- For RGB images, specify a vector with \(c\) equal to 3.
- For multispectral or hyperspectral images, specify a vector with \(c\) equal to the number of channels.

For 3-D image or volume input, use `image3dInputLayer`.

Example: \([224 \ 224 \ 3]\)

**Normalization — Data normalization**

'zerocenter' (default) | 'zscore' | 'rescale-symmetric' | 'rescale-zero-one' | 'none' | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- 'zerocenter' — Subtract the mean specified by Mean.
- 'zscore' — Subtract the mean specified by Mean and divide by StandardDeviation.
- 'rescale-symmetric' — Rescale the input to be in the range \([-1, 1]\) using the minimum and maximum values specified by Min and Max, respectively.
- 'rescale-zero-one' — Rescale the input to be in the range \([0, 1]\) using the minimum and maximum values specified by Min and Max, respectively.
- 'none' — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form \(Y = \text{func}(X)\), where \(X\) is the input data, and the output \(Y\) is the normalized data.

**Tip** The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the 'ResetInputNormalization' option in `trainingOptions` to false.

**NormalizationDimension — Normalization dimension**

'auto' (default) | 'channel' | 'element' | 'all'

Normalization dimension, specified as one of the following:
• 'auto' – If the training option is false and you specify any of the normalization statistics (Mean, StandardDeviation, Min, or Max), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.

• 'channel' – Channel-wise normalization.

• 'element' – Element-wise normalization.

• 'all' – Normalize all values using scalar statistics.

**Mean — Mean for zero-center and z-score normalization**

\[
[] \text{ (default) | 3-D array | numeric scalar}
\]

Mean for zero-center and z-score normalization, specified as a \( h \)-by-\( w \)-by-\( c \) array, a 1-by-1-by-\( c \) array of means per channel, a numeric scalar, or [], where \( h \), \( w \), and \( c \) correspond to the height, width, and the number of channels of the mean, respectively.

If you specify the Mean property, then Normalization must be 'zerocenter' or 'zscore'. If Mean is [], then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using **assembleNetwork**).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**StandardDeviation — Standard deviation for z-score normalization**

\[
[] \text{ (default) | 3-D array | numeric scalar}
\]

Standard deviation for z-score normalization, specified as a \( h \)-by-\( w \)-by-\( c \) array, a 1-by-1-by-\( c \) array of means per channel, a numeric scalar, or [], where \( h \), \( w \), and \( c \) correspond to the height, width, and the number of channels of the standard deviation, respectively.

If you specify the StandardDeviation property, then Normalization must be 'zscore'. If StandardDeviation is [], then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using **assembleNetwork**).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

1-983
Min — Minimum value for rescaling
[] (default) | 3-D array | numeric scalar

Minimum value for rescaling, specified as a h-by-w-by-c array, a 1-by-1-by-c array of minima per channel, a numeric scalar, or [], where h, w, and c correspond to the height, width, and the number of channels of the minima, respectively.

If you specify the Min property, then Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Min is [], then the software calculates the minimum at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Max — Maximum value for rescaling
[] (default) | 3-D array | numeric scalar

Maximum value for rescaling, specified as a h-by-w-by-c array, a 1-by-1-by-c array of maxima per channel, a numeric scalar, or [], where h, w, and c correspond to the height, width, and the number of channels of the maxima, respectively.

If you specify the Max property, then Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Max is [], then the software calculates the maximum at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

DataAugmentation — Data augmentation transforms
'none' (default) | 'randcrop' | 'randfliplr' | cell array of 'randcrop' and 'randfliplr'

Note The DataAugmentation property is not recommended. To preprocess images with cropping, reflection, and other geometric transformations, use augmentedImageDatastore instead.
Data augmentation transforms to use during training, specified as one of the following.

- 'none' — No data augmentation
- 'randcrop' — Take a random crop from the training image. The random crop has the same size as the input size.
- 'randfliplr' — Randomly flip the input images horizontally with a 50% chance.
- Cell array of 'randcrop' and 'randfliplr'. The software applies the augmentation in the order specified in the cell array.

Augmentation of image data is another way of reducing overfitting [1], [2].

Data Types: string | char | cell

Layer

Name — Layer name
' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

NumInputs — Number of inputs
0 (default)

Number of inputs of the layer. The layer has no inputs.

Data Types: double

InputNames — Input names
{} (default)

Input names of the layer. The layer has no inputs.

Data Types: cell

NumOutputs — Number of outputs
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**

{'}out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

## Examples

### Create Image Input Layer

Create an image input layer for 28-by-28 color images with name 'input'. By default, the layer performs data normalization by subtracting the mean image of the training set from every input image.

```matlab
inputlayer = imageInputLayer([28 28 3],'Name','input')
```

```matlab
inputlayer = 
    ImageInputLayer with properties:
     
        Name: 'input'
        InputSize: [28 28 3]

    Hyperparameters
        DataAugmentation: 'none'
        Normalization: 'zerocenter'
        NormalizationDimension: 'auto'
        Mean: []
```

Include an image input layer in a `Layer` array.

```matlab
layers = [ ... 
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```
layers =
7x1 Layer array with layers:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>''</td>
<td>Image Input 28x28x1 images with 'zerocenter' normalization</td>
</tr>
<tr>
<td>2</td>
<td>''</td>
<td>Convolution 20 5x5 convolutions with stride [1 1] and padding</td>
</tr>
<tr>
<td>3</td>
<td>''</td>
<td>ReLU ReLU</td>
</tr>
<tr>
<td>4</td>
<td>''</td>
<td>Max Pooling 2x2 max pooling with stride [2 2] and padding [0</td>
</tr>
<tr>
<td>5</td>
<td>''</td>
<td>Fully Connected 10 fully connected layer</td>
</tr>
<tr>
<td>6</td>
<td>''</td>
<td>Softmax softmax</td>
</tr>
<tr>
<td>7</td>
<td>''</td>
<td>Classification Output crossentropyex</td>
</tr>
</tbody>
</table>

**Compatibility Considerations**

**AverageImage property will be removed**

*Not recommended starting in R2019b*

AverageImage will be removed. Use Mean instead. To update your code, replace all instances of AverageImage with Mean. There are no differences between the properties that require additional updates to your code.

**imageInputLayer and image3dInputLayer, by default, use channel-wise normalization**

*Behavior change in future release*

Starting in R2019b, imageInputLayer and image3dInputLayer, by default, use channel-wise normalization. In previous versions, these layers use element-wise normalization. To reproduce this behavior, set the NormalizationDimension option of these layers to 'element'.

**References**


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
augmentedImageDatastore | convolution2dLayer | fullyConnectedLayer | image3dInputLayer | maxPooling2dLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a
image3dInputLayer

3-D image input layer

Description

A 3-D image input layer inputs 3-D images or volumes to a network and applies data normalization.

For 2-D image input, use imageInputLayer.

Creation

Syntax

layer = image3dInputLayer(inputSize)
layer = image3dInputLayer(inputSize,Name,Value)

Description

layer = image3dInputLayer(inputSize) returns a 3-D image input layer and specifies the InputSize property.

layer = image3dInputLayer(inputSize,Name,Value) sets the optional properties using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

3-D Image Input

InputSize — Size of the input
row vector of integers
Size of the input data, specified as a row vector of integers \([h \ w \ d \ c]\), where \(h\), \(w\), \(d\),
and \(c\) correspond to the height, width, depth, and number of channels respectively.

- For grayscale input, specify a vector with \(c\) equal to 1.
- For RGB input, specify a vector with \(c\) equal to 3.
- For multispectral or hyperspectral input, specify a vector with \(c\) equal to the number of channels.

For 2-D image input, use `imageInputLayer`.

Example: \([132 \ 132 \ 116 \ 3]\)

**Normalization — Data normalization**

- `'zerocenter'` (default) | `'zscore'` | `'rescale-symmetric'` | `'rescale-zero-one'` | `'none'` | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- `'zerocenter'` — Subtract the mean specified by `Mean`.
- `'zscore'` — Subtract the mean specified by `Mean` and divide by `StandardDeviation`.
- `'rescale-symmetric'` — Rescale the input to be in the range \([-1, 1]\) using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'rescale-zero-one'` — Rescale the input to be in the range \([0, 1]\) using the minimum and maximum values specified by `Min` and `Max`, respectively.
- `'none'` — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form \(Y = \text{func}(X)\), where \(X\) is the input data, and the output \(Y\) is the normalized data.

**Tip** The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

**NormalizationDimension — Normalization dimension**

- `'auto'` (default) | `'channel'` | `'element'` | `'all'`

Normalization dimension, specified as one of the following:
• 'auto' – If the training option is false and you specify any of the normalization statistics (Mean, StandardDeviation, Min, or Max), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.

• 'channel' – Channel-wise normalization.

• 'element' – Element-wise normalization.

• 'all' – Normalize all values using scalar statistics.

**Mean — Mean for zero-center and z-score normalization**

```
[ ] (default) | 4-D array | numeric scalar
```

Mean for zero-center and z-score normalization, specified as a h-by-w-by-d-by-c array, a 1-by-1-by-1-by-c array of means per channel, a numeric scalar, or [], where h, w, d, and c correspond to the height, width, depth, and the number of channels of the mean, respectively.

If you specify the Mean property, then Normalization must be 'zerocenter' or 'zscore'. If Mean is [], then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**StandardDeviation — Standard deviation for z-score normalization**

```
[ ] (default) | 4-D array | numeric scalar
```

Standard deviation for z-score normalization, specified as a h-by-w-by-d-by-c array, a 1-by-1-by-1-by-c array of means per channel, a numeric scalar, or [], where h, w, d, and c correspond to the height, width, depth, and the number of channels of the standard deviation, respectively.

If you specify the StandardDeviation property, then Normalization must be 'zscore'. If StandardDeviation is [], then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
**Min — Minimum value for rescaling**

[ ] (default) | 4-D array | numeric scalar

Minimum value for rescaling, specified as a $h$-by-$w$-by-$d$-by-$c$ array, a 1-by-1-by-1-by-$c$ array of minima per channel, a numeric scalar, or [], where $h$, $w$, $d$, and $c$ correspond to the height, width, depth, and the number of channels of the minima, respectively.

If you specify the Min property, then Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Min is [], then the software calculates the minimum at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Max — Maximum value for rescaling**

[ ] (default) | 4-D array | numeric scalar

Maximum value for rescaling, specified as a $h$-by-$w$-by-$d$-by-$c$ array, a 1-by-1-by-1-by-$c$ array of maxima per channel, a numeric scalar, or [], where $h$, $w$, $d$, and $c$ correspond to the height, width, depth, and the number of channels of the maxima, respectively.

If you specify the Min property, then Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Max is [], then the software calculates the maximum at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Layer**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.
Data Types: char | string

**NumInputs — Number of inputs**
0 (default)

Number of inputs of the layer. The layer has no inputs.
Data Types: double

**InputNames — Input names**
{} (default)

Input names of the layer. The layer has no inputs.
Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create 3-D Image Input Layer**

Create a 3-D image input layer for 132-by-132-by-116 color 3-D images with name 'input'. By default, the layer performs data normalization by subtracting the mean image of the training set from every input image.

```matlab
layer = image3dInputLayer([132 132 116],'Name','input')
```

layer =
   Image3DInputLayer with properties:
Include a 3-D image input layer in a `Layer` array.

```matlab
layers = [
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,16,'Stride',4)
    reluLayer
    maxPooling3dLayer(2,'Stride',4)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

**Compatibility Considerations**

**AverageImage property will be removed**

*Not recommended starting in R2019b*

AverageImage will be removed. Use Mean instead. To update your code, replace all instances of `AverageImage` with `Mean`. There are no differences between the properties that require additional updates to your code.
imageInputLayer and image3dInputLayer, by default, use channel-wise normalization

Behavior change in future release

Starting in R2019b, imageInputLayer and image3dInputLayer, by default, use channel-wise normalization. In previous versions, these layers use element-wise normalization. To reproduce this behavior, set the NormalizationDimension option of these layers to 'element'.

See Also
averagePooling3dLayer | convolution3dLayer | fullyConnectedLayer | imageInputLayer | maxPooling3dLayer | trainNetwork | transposedConv3dLayer

Topics
“3-D Brain Tumor Segmentation Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a
maxPooling2dLayer

Max pooling layer

Description

A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.

Creation

Syntax

layer = maxPooling2dLayer(poolSize)
layer = maxPooling2dLayer(poolSize,Name,Value)

Description

layer = maxPooling2dLayer(poolSize) creates a max pooling layer and sets the PoolSize property.

layer = maxPooling2dLayer(poolSize,Name,Value) sets the optional Stride, Name, and HasUnpoolingOutputs properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example, maxPooling2dLayer(2,'Stride',3) creates a max pooling layer with pool size [2 2] and stride [3 3]. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input and to set the Stride, Name, and HasUnpoolingOutputs properties. Enclose names in single quotes.

Example: maxPooling2dLayer(2,'Stride',3) creates a max pooling layer with pool size [2 2] and stride [3 3].

Padding — Input edge padding
[0 0 0 0] (default) | vector of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

• 'same' — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is ceil(inputSize/stride), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

• Nonnegative integer p — Add padding of size p to all the edges of the input.

• Vector [a b] of nonnegative integers — Add padding of size a to the top and bottom of the input and padding of size b to the left and right.

• Vector [t b l r] of nonnegative integers — Add padding of size t to the top, b to the bottom, l to the left, and r to the right of the input.

Example: 'Padding',1 adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.

Example: 'Padding','same' adds padding so that the output has the same size as the input (if the stride equals 1).
Properties

Max Pooling

**PoolSize — Dimensions of pooling regions**
vector of two positive integers

Dimensions of the pooling regions, specified as a vector of two positive integers \([h \ w]\), where \(h\) is the height and \(w\) is the width. When creating the layer, you can specify \(\text{PoolSize}\) as a scalar to use the same value for both dimensions.

If the stride dimensions \(\text{Stride}\) are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions \(\text{PaddingSize}\) must be less than the pooling region dimensions \(\text{PoolSize}\).

Example: \([2 \ 1]\) specifies pooling regions of height 2 and width 1.

**Stride — Step size for traversing input**
\([1 \ 1]\) (default) | vector of two positive integers

Step size for traversing the input vertically and horizontally, specified as a vector of two positive integers \([a \ b]\), where \(a\) is the vertical step size and \(b\) is the horizontal step size. When creating the layer, you can specify \(\text{Stride}\) as a scalar to use the same value for both dimensions.

If the stride dimensions \(\text{Stride}\) are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions \(\text{PaddingSize}\) must be less than the pooling region dimensions \(\text{PoolSize}\).

Example: \([2 \ 3]\) specifies a vertical step size of 2 and a horizontal step size of 3.

**PaddingSize — Size of padding**
\([0 \ 0 \ 0 \ 0]\) (default) | vector of four nonnegative integers

Size of padding to apply to input borders, specified as a vector \([t \ b \ l \ r]\) of four nonnegative integers, where \(t\) is the padding applied to the top, \(b\) is the padding applied to the bottom, \(l\) is the padding applied to the left, and \(r\) is the padding applied to the right.
When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: [1 1 2 2] adds one row of padding to the top and bottom, and two columns of padding to the left and right of the input.

**PaddingMode — Method to determine padding size**

'manual' (default) | 'same'

Method to determine padding size, specified as 'manual' or 'same'.

The software automatically sets the value of PaddingMode based on the 'Padding' value you specify when creating a layer.

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets PaddingMode to 'manual'.
- If you set the 'Padding' option to 'same', then the software automatically sets PaddingMode to 'same' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is ceil(inputSize/stride), where inputSize is the height or width of the input and stride is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, and to the left and right, if possible. If the padding that must be added vertically has an odd value, then the software adds extra padding to the bottom. If the padding that must be added horizontally has an odd value, then the software adds extra padding to the right.

**Padding — Size of padding**

[0 0] (default) | vector of two nonnegative integers

**Note** Padding property will be removed in a future release. Use PaddingSize instead.

When creating a layer, use the 'Padding' name-value pair argument to specify the padding size.

Size of padding to apply to input borders vertically and horizontally, specified as a vector [a b] of two nonnegative integers, where a is the padding applied to the top and bottom of the input data and b is the padding applied to the left and right.

Example: [1 1] adds one row of padding to the top and bottom, and one column of padding to the left and right of the input.
**HasUnpoolingOutputs — Flag for outputs to unpooling layer**

false (default) | true

Flag for outputs to unpooling layer, specified as true or false.

If the HasUnpoolingOutputs value equals false, then the max pooling layer has a single output with the name 'out'.

To use the output of a max pooling layer as the input to a max unpooling layer, set the HasUnpoolingOutputs value to true. In this case, the max pooling layer has two additional outputs that you can connect to a max unpooling layer:

- 'indices' — Indices of the maximum value in each pooled region.
- 'size' — Size of the input feature map.

To enable outputs to a max unpooling layer, the pooling regions of the max pooling layer must be nonoverlapping.

For more information on how to unpool the output of a max pooling layer, see `maxUnpooling2dLayer`.

**Layer**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{ 'in' } (default)
Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default) | 3

Number of outputs of the layer.

If the HasUnpoolingOutputs value equals false, then the max pooling layer has a single output with the name 'out'.

To use the output of a max pooling layer as the input to a max unpooling layer, set the HasUnpoolingOutputs value to true. In this case, the max pooling layer has two additional outputs that you can connect to a max unpooling layer:

- 'indices' — Indices of the maximum value in each pooled region.
- 'size' — Size of the input feature map.

To enable outputs to a max unpooling layer, the pooling regions of the max pooling layer must be nonoverlapping.

For more information on how to unpool the output of a max pooling layer, see `maxUnpooling2dLayer`.

Data Types: double

**OutputNames — Output names**

{'out'} (default) | {'out','indices','size'}

Output names of the layer.

If the HasUnpoolingOutputs value equals false, then the max pooling layer has a single output with the name 'out'.

To use the output of a max pooling layer as the input to a max unpooling layer, set the HasUnpoolingOutputs value to true. In this case, the max pooling layer has two additional outputs that you can connect to a max unpooling layer:

- 'indices' — Indices of the maximum value in each pooled region.
- 'size' — Size of the input feature map.

To enable outputs to a max unpooling layer, the pooling regions of the max pooling layer must be nonoverlapping.
For more information on how to unpool the output of a max pooling layer, see `maxUnpooling2dLayer`.

Data Types: `cell`

**Examples**

**Create Max Pooling Layer with Nonoverlapping Pooling Regions**

Create a max pooling layer with nonoverlapping pooling regions.

```matlab
layer = maxPooling2dLayer(2,'Stride',2)
```

```matlab
dl = maxPooling2dLayer(2,'Stride',2)
dl =
    MaxPooling2DLayer with properties:
      Name: ''
     HasUnpoolingOutputs: 0
      NumOutputs: 1
    OutputNames: {'out'}

    Hyperparameters
      PoolSize: [2 2]
      Stride: [2 2]
      PaddingMode: 'manual'
     PaddingSize: [0 0 0 0]
```

The height and the width of the rectangular regions (pool size) are both 2. The pooling regions do not overlap because the step size for traversing the images vertically and horizontally (stride) is also [2 2].

Include a max pooling layer with nonoverlapping regions in a `Layer` array.

```matlab
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```
Create Max Pooling Layer with Overlapping Pooling Regions

Create a max pooling layer with overlapping pooling regions.

```matlab
layer = maxPooling2dLayer([3 2], 'Stride', 2)
```

This layer creates pooling regions of size [3 2] and takes the maximum of the six elements in each region. The pooling regions overlap because there are stride dimensions `Stride` that are less than the respective pooling dimensions `PoolSize`.

Include a max pooling layer with overlapping pooling regions in a `Layer` array.

```matlab
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5, 20)
    reluLayer
    ... ];
```
maxPooling2dLayer([3 2], 'Stride', 2)
fullyConnectedLayer(10)
softmaxLayer
classificationLayer]

layers =
7x1 Layer array with layers:

1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0  0  0]
3   ''   ReLU                    ReLU
4   ''   Max Pooling             3x2 max pooling with stride [2  2] and padding [0  0  0  0]
5   ''   Fully Connected         10 fully connected layer
6   ''   Softmax                 softmax
7   ''   Classification Output   crossentropyex

More About

Max Pooling Layer

A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.

Pooling layers follow the convolutional layers for down-sampling, hence, reducing the number of connections to the following layers. They do not perform any learning themselves, but reduce the number of parameters to be learned in the following layers. They also help reduce overfitting.

A max pooling layer returns the maximum values of rectangular regions of its input. The size of the rectangular regions is determined by the poolSize argument of maxPoolingLayer. For example, if poolSize equals [2, 3], then the layer returns the maximum value in regions of height 2 and width 3.

Pooling layers scan through the input horizontally and vertically in step sizes you can specify using the 'Stride' name-value pair argument. If the pool size is smaller than or equal to the stride, then the pooling regions do not overlap.

For nonoverlapping regions (Pool Size and Stride are equal), if the input to the pooling layer is n-by-n, and the pooling region size is h-by-h, then the pooling layer down-samples the regions by h [1]. That is, the output of a max or average pooling layer for one channel...
of a convolutional layer is \( n/h \)-by-\( n/h \). For overlapping regions, the output of a pooling layer is \((\text{Input Size} - \text{Pool Size} + 2\times\text{Padding})/\text{Stride} + 1\).

**References**


**Extended Capabilities**

**GPU Code Generation**
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

**See Also**
averagePooling2dLayer | convolution2dLayer | globalAveragePooling2dLayer | maxUnpooling2dLayer

**Topics**
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

**Introduced in R2016a**
maxPooling3dLayer

3-D max pooling layer

Description

A 3-D max pooling layer performs down-sampling by dividing three-dimensional input into cuboidal pooling regions, and computing the maximum of each region.

Creation

Syntax

layer = maxPooling3dLayer(poolSize)
layer = maxPooling3dLayer(poolSize,Name,Value)

Description

layer = maxPooling3dLayer(poolSize) creates a 3-D max pooling layer and sets the PoolSize property.

layer = maxPooling3dLayer(poolSize,Name,Value) sets the optional Stride and Name properties using name-value pairs. To specify input padding, use the 'Padding' name-value pair argument. For example, 
maxPooling3dLayer(2,'Stride',3) creates a 3-D max pooling layer with pool size [2 2 2] and stride [3 3 3]. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Input Arguments

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to specify the size of the padding to add along the edges of the layer input and to set the Stride and Name properties. Enclose names in single quotes.

Example: `maxPooling3dLayer(2,'Stride',3)` creates a 3-D max pooling layer with pool size `[2 2 2]` and stride `[3 3 3]`.

Padding — Input edge padding

0 (default) | array of nonnegative integers | 'same'

Input edge padding, specified as the comma-separated pair consisting of 'Padding' and one of these values:

- `'same'` — Add padding of size calculated by the software at training or prediction time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is `ceil(inputSize/stride)`, where
- `inputSize` is the height, width, or depth of the input and `stride` is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

- Nonnegative integer `p` — Add padding of size `p` to all the edges of the input.
- Three-element vector `[a b c]` of nonnegative integers — Add padding of size `a` to the top and bottom, padding of size `b` to the left and right, and padding of size `c` to the front and back of the input.
- 2-by-3 matrix `[t l f; b r k]` of nonnegative integers — Add padding of size `t` to the top, `b` to the bottom, `l` to the left, `r` to the right, `f` to the front, and `k` to the back of the input. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

Example: `'Padding',1` adds one row of padding to the top and bottom, one column of padding to the left and right, and one plane of padding to the front and back of the input.

Example: `'Padding','same'` adds padding so that the output has the same size as the input (if the stride equals 1).
Properties

Max Pooling

PoolSize — Dimensions of pooling regions
vector of three positive integers

Dimensions of the pooling regions, specified as a vector of three positive integers \([h \ w \ d]\), where \(h\) is the height, \(w\) is the width, and \(d\) is the depth. When creating the layer, you can specify PoolSize as a scalar to use the same value for all three dimensions.

If the stride dimensions Stride are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions PaddingSize must be less than the pooling region dimensions PoolSize.

Example: \([2 \ 1 \ 1]\) specifies pooling regions of height 2, width 1, and depth 1.

Stride — Step size for traversing input
\([1 \ 1 \ 1]\) (default) | vector of three positive integers

Step size for traversing the input in three dimensions, specified as a vector \([a \ b \ c]\) of three positive integers, where \(a\) is the vertical step size, \(b\) is the horizontal step size, and \(c\) is the step size along the depth direction. When creating the layer, you can specify Stride as a scalar to use the same value for step sizes in all three directions.

If the stride dimensions Stride are less than the respective pooling dimensions, then the pooling regions overlap.

The padding dimensions PaddingSize must be less than the pooling region dimensions PoolSize.

Example: \([2 \ 3 \ 1]\) specifies a vertical step size of 2, a horizontal step size of 3, and a step size along the depth of 1.

PaddingSize — Size of padding
\([0 \ 0 \ 0;0 \ 0 \ 0]\) (default) | 2-by-3 matrix of nonnegative integers

Size of padding to apply to input borders, specified as 2-by-3 matrix \([t \ l \ f;b \ r \ k]\) of nonnegative integers, where \(t\) and \(b\) are the padding applied to the top and bottom in the vertical direction, \(l\) and \(r\) are the padding applied to the left and right in the horizontal
direction, and \(f\) and \(k\) are the padding applied to the front and back along the depth. In other words, the top row specifies the prepadding and the second row defines the postpadding in the three dimensions.

When you create a layer, use the 'Padding' name-value pair argument to specify the padding size.

Example: \([[1, 2, 4]; [1, 2, 4]]\) adds one row of padding to the top and bottom, two columns of padding to the left and right, and four planes of padding to the front and back of the input.

**PaddingMode — Method to determine padding size**

'\texttt{manual}' (default) | '\texttt{same}'

Method to determine padding size, specified as '\texttt{manual}' or '\texttt{same}'.

The software automatically sets the value of `PaddingMode` based on the 'Padding' value you specify when creating a layer:

- If you set the 'Padding' option to a scalar or a vector of nonnegative integers, then the software automatically sets `PaddingMode` to '\texttt{manual}'.
- If you set the 'Padding' option to '\texttt{same}', then the software automatically sets `PaddingMode` to '\texttt{same}' and calculates the size of the padding at training time so that the output has the same size as the input when the stride equals 1. If the stride is larger than 1, then the output size is \(\text{ceil}(\text{inputSize}/\text{stride})\), where \text{inputSize} is the height, width, or depth of the input and \text{stride} is the stride in the corresponding dimension. The software adds the same amount of padding to the top and bottom, to the left and right, and to the front and back, if possible. If the padding in a given dimension has an odd value, then the software adds the extra padding to the input as postpadding. In other words, the software adds extra vertical padding to the bottom, extra horizontal padding to the right, and extra depth padding to the back of the input.

**Layer**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.
Data Types: char | string

**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**
{'in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create Max Pooling 3-D Layer with Nonoverlapping Pooling Regions**

Create a max pooling 3-D layer with nonoverlapping pooling regions.

```matlab
layer = maxPooling3dLayer(2,'Stride',2)
layer =
    MaxPooling3DLayer with properties:
        Name: ''
```

NumOutputs: 1
OutputNames: {'out'}

Hyperparameters
  PoolSize: [2 2 2]
  Stride: [2 2 2]
  PaddingMode: 'manual'
  PaddingSize: [2x3 double]

The height, width, and depth of the cuboidal regions (pool size) are 2. The step size for traversing the images (stride) is 2 in all dimensions. The pooling regions do not overlap because the stride is greater than or equal to the corresponding pool size in all dimensions.

Include a max pooling layer with nonoverlapping regions in a Layer array.

```matlab
layers = [ ...
    image3dInputLayer([28 28 28 3])
    convolution3dLayer(5,20)
    reluLayer
    maxPooling3dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

Create Max Pooling 3-D Layer with Overlapping Pooling Regions

Create a max pooling 3-D layer with overlapping pooling regions and padding for the top and bottom of the input.
layer = maxPooling3dLayer([3 2 2],'Stride',2,'Padding',[1 0 0])

layer =
    MaxPooling3DLayer with properties:

    Name: ''
    NumOutputs: 1
    OutputNames: {'out'}

Hyperparameters
    PoolSize: [3 2 2]
    Stride: [2 2 2]
    PaddingMode: 'manual'
    PaddingSize: [2x3 double]

This layer creates pooling regions of size 3-by-2-by-2 and takes the maximum of the
twelve elements in each region. The stride is 2 in all dimensions. The pooling regions
overlap because there are stride dimensions Stride that are less than the respective
pooling dimensions PoolSize.

More About

3-D Max Pooling Layer

A 3-D max pooling layer extends the functionality of a max pooling layer to a third
dimension, depth. A max pooling layer performs down-sampling by dividing the input into
rectangular or cuboidal pooling regions, and computing the maximum of each region. To
learn more, see the definition of max pooling layer on page 1-1004 on the
maxPooling2dLayer reference page.

See Also

averagePooling3dLayer | convolution3dLayer |
globalAveragePooling3dLayer | maxPooling2dLayer

Topics

“3-D Brain Tumor Segmentation Using Deep Learning”


“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

**Introduced in R2019a**
maxUnpooling2dLayer

Max unpooling layer

Description

A max unpooling layer un pools the output of a max pooling layer.

Creation

Syntax

layer = maxUnpooling2dLayer
layer = maxUnpooling2dLayer('Name',name)

Description

layer = maxUnpooling2dLayer creates a max unpooling layer.

layer = maxUnpooling2dLayer('Name',name) sets the Name property. To create a network containing a max unpooling layer you must specify a layer name.

Properties

Name — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string
**NumInputs — Number of inputs**
3 (default)

Number of inputs of the layer.

There are three inputs to this layer:

- 'in' — Input feature map to unpool.
- 'indices' — Indices of the maximum value in each pooled region. This is output by the max pooling layer.
- 'size' — Output size of unpoled feature map. This is output by the max pooling layer.

Use the input names when connecting or disconnecting the max unpooling layer to other layers using `connectLayers` or `disconnectLayers`.

Data Types: `double`

**InputNames — Input names**
{'in','indices','size'} (default)

Input names of the layer.

There are three inputs to this layer:

- 'in' — Input feature map to unpool.
- 'indices' — Indices of the maximum value in each pooled region. This is output by the max pooling layer.
- 'size' — Output size of unpoled feature map. This is output by the max pooling layer.

Use the input names when connecting or disconnecting the max unpooling layer to other layers using `connectLayers` or `disconnectLayers`.

Data Types: `cell`

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`
OutputNames — Output names
{ 'out' } (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

Examples

Create Max Unpooling Layer

Create a max unpooling layer that unpool the output of a max pooling layer.

layer = maxUnpooling2dLayer

layer = 
    MaxUnpooling2DLayer with properties:
        Name: ''
        NumInputs: 3
        InputNames: {'in' 'indices' 'size'}

Unpool Max Pooling Layer

Create a max pooling layer, and set the 'HasUnpoolingOutputs' property as true. This property gives the max pooling layer two additional outputs, 'indices' and 'size', which enables unpooling the layer. Also create a max unpooling layer.

layers = [ 
    maxPooling2dLayer(2,'Stride',2,'Name','mpool','HasUnpoolingOutputs',true)
    maxUnpooling2dLayer('Name','unpool')
]

layers = 
    2x1 Layer array with layers:
    1    'mpool'    Max Pooling    2x2 max pooling with stride [2  2] and padding [0
    2    'unpool'   Max Unpooling  Max Unpooling

1-1016
Sequentially connect layers by adding them to a layerGraph. This step connects the 'out' output of the max pooling layer to the 'in' input of the max unpooling layer.

```matlab
lgraph = layerGraph(layers)

lgraph = 
    LayerGraph with properties:

    Layers: [2x1 nnet.cnn.layer.Layer]
    Connections: [1x2 table]
    InputNames: {1x0 cell}
    OutputNames: {1x0 cell}
```

Unpool the output of the max pooling layer, by connecting the max pooling layer outputs to the max unpooling layer inputs.

```matlab
lgraph = connectLayers(lgraph,'mpool/indices','unpool/indices');
lgraph = connectLayers(lgraph,'mpool/size','unpool/size');
```

### Extended Capabilities

#### GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

#### See Also
connectLayers | disconnectLayers | layerGraph | maxPooling2dLayer | trainNetwork

#### Topics
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Train Residual Network for Image Classification”
“List of Deep Learning Layers”
Introduced in R2017b
RegressionOutputLayer

Regression output layer

Description

A regression layer computes the half-mean-squared-error loss for regression problems.

Creation

Create a regression output layer using `regressionLayer`.

Properties

Regression Output

`ResponseNames` — Names of responses

{} (default) | cell array of character vectors | string array

Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is {}.

Data Types: `cell`

`LossFunction` — Loss function for training

`'mean-squared-error'`

Loss function the software uses for training, specified as `'mean-squared-error'`.

Layer

`Name` — Layer name

`' '` (default) | character vector | string scalar
Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

0 (default)

Number of outputs of the layer. The layer has no outputs.

Data Types: double

**OutputNames — Output names**

{} (default)

Output names of the layer. The layer has no outputs.

Data Types: cell

**Examples**

**Create Regression Output Layer**

Create a regression output layer with the name 'routput'.

```latex
layer = regressionLayer('Name','routput')
```
layer = RegressionOutputLayer with properties:

Name: 'routput'
ResponseNames: {}  

Hyperparameters
  LossFunction: 'mean-squared-error'

The default loss function for regression is mean-squared-error.

Include a regression output layer in a Layer array.

layers = [...
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer]

layers = 5x1 Layer array with layers:
1   ''   Image Input         28x28x1 images with 'zerocenter' normalization
2   ''   Convolution         25 12x12 convolutions with stride [1  1] and padding
3   ''   ReLU                ReLU
4   ''   Fully Connected     1 fully connected layer
5   ''   Regression Output   mean-squared-error

More About

Regression Output Layer

A regression layer computes the half-mean-squared-error loss for regression problems. For typical regression problems, a regression layer must follow the final fully connected layer.

For a single observation, the mean-squared-error is given by:
\[ \text{MSE} = \sum_{i=1}^{R} \frac{(t_i - y_i)^2}{R}, \]

where \( R \) is the number of responses, \( t_i \) is the target output, and \( y_i \) is the network’s prediction for response \( i \).

For image and sequence-to-one regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses, not normalized by \( R \):

\[ \text{loss} = \frac{1}{2} \sum_{i=1}^{R} (t_i - y_i)^2. \]

For image-to-image regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses for each pixel, not normalized by \( R \):

\[ \text{loss} = \frac{1}{2} \sum_{p=1}^{HWC} (t_p - y_p)^2, \]

where \( H, W, \) and \( C \) denote the height, width, and number of channels of the output respectively, and \( p \) indexes into each element (pixel) of \( t \) and \( y \) linearly.

For sequence-to-sequence regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses for each time step, not normalized by \( R \):

\[ \text{loss} = \frac{1}{2S} \sum_{i=1}^{S} \sum_{j=1}^{R} (t_{ij} - y_{ij})^2, \]

where \( S \) is the sequence length.

When training, the software calculates the mean loss over the observations in the mini-batch.
See Also
classificationLayer | fullyConnectedLayer | regressionLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2017a
reluLayer

Rectified Linear Unit (ReLU) layer

Description

A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.

This operation is equivalent to

\[ f(x) = \begin{cases} 
  x, & x \geq 0 \\
  0, & x < 0 .
\end{cases} \]

Creation

Syntax

layer = reluLayer
layer = reluLayer('Name',Name)

Description

layer = reluLayer creates a ReLU layer.

layer = reluLayer('Name',Name) creates a ReLU layer and sets the optional Name property using a name-value pair. For example, reluLayer('Name','relu1') creates a ReLU layer with the name 'relu1'. Enclose the property name in single quotes.

Properties

Name — Layer name

' ' (default) | character vector | string scalar
Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

## Examples

**Create ReLU Layer**

Create a ReLU layer with the name 'relu1'.

```
layer = reluLayer('Name','relu1')
```
layer =
    ReLULayer with properties:

    Name: 'relu1'

Include a ReLU layer in a Layer array.

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]

layers =
    7x1 Layer array with layers:

1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0  0  0]
3   ''   ReLU                    ReLU
4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
5   ''   Fully Connected         10 fully connected layer
6   ''   Softmax                 softmax
7   ''   Classification Output   crossentropyex

More About

ReLU Layer

A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.

Convolutional and batch normalization layers are usually followed by a nonlinear activation function such as a rectified linear unit (ReLU), specified by a ReLU layer. A ReLU layer performs a threshold operation to each element, where any input value less than zero is set to zero, that is,

\[
f(x) = \begin{cases} 
  x, & x \geq 0 \\
  0, & x < 0
\end{cases}
\]
The ReLU layer does not change the size of its input.

There are other nonlinear activation layers that perform different operations and can improve the network accuracy for some applications. For a list of activation layers, see “Activation Layers”.

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
batchNormalizationLayer | clippedReluLayer | leakyReluLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2016a
leakyReluLayer

Leaky Rectified Linear Unit (ReLU) layer

Description

A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.

This operation is equivalent to:

\[ f(x) = \begin{cases} 
  x, & x \geq 0 \\
  scale \times x, & x < 0 
\end{cases} \]

Creation

Syntax

layer = leakyReluLayer
layer = leakyReluLayer(scale)
layer = leakyReluLayer(___,'Name',Name)

Description

layer = leakyReluLayer returns a leaky ReLU layer.

layer = leakyReluLayer(scale) returns a leaky ReLU layer with a scalar multiplier for negative inputs equal to scale.

layer = leakyReluLayer(___,'Name',Name) returns a leaky ReLU layer and sets the optional Name property.
Properties

Leaky ReLU

Scale — Scalar multiplier for negative input values
0.01 (default) | numeric scalar

Scalar multiplier for negative input values, specified as a numeric scalar.
Example: 0.4

Layer

Name — Layer name
'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

NumInputs — Number of inputs
1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

InputNames — Input names
{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

NumOutputs — Number of outputs
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double
**OutputNames — Output names**

\{'out\}' (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**

**Create Leaky ReLU Layer**

Create a leaky ReLU layer with the name 'leaky1' and a scalar multiplier for negative inputs equal to 0.1.

```matlab
layer = leakyReluLayer(0.1,'Name','leaky1')
```

```
layer = 
LeakyReLULayer with properties:

    Name: 'leaky1'

Hyperparameters
Scale: 0.1000
```

Include a leaky ReLU layer in a Layer array.

```matlab
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16)
    batchNormalizationLayer
    leakyReluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32)
    batchNormalizationLayer
    leakyReluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
]```
layers =
11x1 Layer array with layers:

1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
2   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [0  0  0  0]
3   ''   Batch Normalization     Batch normalization
4   ''   Leaky ReLU              Leaky ReLU with scale 0.01
5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]
6   ''   Convolution             32 3x3 convolutions with stride [1  1] and padding [0  0  0  0]
7   ''   Batch Normalization     Batch normalization
8   ''   Leaky ReLU              Leaky ReLU with scale 0.01
9   ''   Fully Connected         10 fully connected layer
10  ''   Softmax                 softmax
11  ''   Classification Output   crossentropyex

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
clippedReluLayer | reluLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”
Introduced in R2017b
clippedReluLayer

Clipped Rectified Linear Unit (ReLU) layer

Description

A clipped ReLU layer performs a threshold operation, where any input value less than zero is set to zero and any value above the clipping ceiling is set to that clipping ceiling.

This operation is equivalent to:

\[ f(x) = \begin{cases} 
0, & x < 0 \\
x, & 0 \leq x < \text{ceiling} \\
\text{ceiling}, & x \geq \text{ceiling} 
\end{cases} \]

This clipping prevents the output from becoming too large.

Creation

Syntax

layer = clippedReluLayer(ceiling)
layer = clippedReluLayer(ceiling,'Name',Name)

Description

layer = clippedReluLayer(ceiling) returns a clipped ReLU layer with the clipping ceiling equal to ceiling.

layer = clippedReluLayer(ceiling,'Name',Name) sets the optional Name property.
Properties

Clipped ReLU

**Ceiling — Ceiling for input clipping**

positive scalar

Ceiling for input clipping, specified as a positive scalar.

Example: 10

Layer

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{ 'in' } (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double
OutputNames — Output names

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

Examples

Create Clipped ReLU Layer

Create a clipped ReLU layer with the name 'clip1' and the clipping ceiling equal to 10.

```matlab
layer = clippedReluLayer(10,'Name','clip1')
```

```matlab
dlloyer =
    ClippedReLULayer with properties:
        Name: 'clip1'

        Hyperparameters
            Ceiling: 10
```

Include a clipped ReLU layer in a Layer array.

```matlab
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    clippedReluLayer(10)
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```matlab
layers =
    7x1 Layer array with layers:
        1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
        2   ''   Convolution             20 5x5 convolutions with stride [1 1] and padding [0 0 0 0]
        3   ''   Clipped ReLU            Clipped ReLU with ceiling 10
        4   ''   Max Pooling             2x2 max pooling with stride [2 2] and padding [0 0 0 0]
```
Fully Connected       10 fully connected layer
Softmax                softmax
Classification Output  crossentropyex

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
leakyReluLayer | reluLayer

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2017b
eluLayer

Exponential linear unit (ELU) layer

Description

An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.

The layer performs the following operation:

\[
f(x) = \begin{cases} 
  x, & x \geq 0 \\
  \alpha(\exp(x) - 1), & x < 0 
\end{cases}
\]

The default value of \( \alpha \) is 1. Specify a value of \( \alpha \) for the layer by setting the \texttt{Alpha} property.

Creation

Syntax

layer = eluLayer
layer = eluLayer(alpha)
layer = eluLayer(___,'Name',Name)

Description

\texttt{layer = eluLayer} creates an ELU layer.

\texttt{layer = eluLayer(alpha)} creates an ELU layer and specifies the \texttt{Alpha} property.

\texttt{layer = eluLayer(___,'Name',Name)} additionally sets the optional \texttt{Name} property using any of the previous syntaxes. For example, \texttt{eluLayer('Name','elu1')} creates an ELU layer with the name 'elu1'.

1-1037
Properties

**ELU**

**Alpha — Nonlinearity parameter**

1 (default) | numeric scalar

Nonlinearity parameter $\alpha$, specified as a numeric scalar. The minimum value of the output of the ELU layer equals $-\alpha$ and the slope at negative inputs approaching 0 is $\alpha$.

**Layer**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'}in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double
**OutputNames — Output names**

`{'out'}` (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**

**Create ELU Layer**

Create an exponential linear unit (ELU) layer with the name 'elu1' and a default value of 1 for the nonlinearity parameter Alpha.

```matlab
layer = eluLayer('Name','elu1')
```

```
layer = 
  ELULayer with properties:
       Name: 'elu1'
      Alpha: 1

Show all properties
```

Include an ELU layer in a Layer array.

```matlab
layers = [ 
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16)
    batchNormalizationLayer
    eluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32)
    batchNormalizationLayer
    eluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
]
```
layers =
11x1 Layer array with layers:

1  ''  Image Input  28x28x1 images with 'zerocenter' normalization
2  ''  Convolution  16 3x3 convolutions with stride [1 1] and padding [0 0 0 0]
3  ''  Batch Normalization  Batch normalization
4  ''  ELU  ELU with Alpha 1
5  ''  Max Pooling  2x2 max pooling with stride [2 2] and padding [0 0 0 0]
6  ''  Convolution  32 3x3 convolutions with stride [1 1] and padding [0 0 0 0]
7  ''  Batch Normalization  Batch normalization
8  ''  ELU  ELU with Alpha 1
9  ''  Fully Connected  10 fully connected layer
10  ''  Softmax  softmax
11  ''  Classification Output  crossentropyex

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
batchNormalizationLayer|clippedReluLayer|leakyReluLayer|reluLayer|trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

**Introduced in R2019a**
tanhLayer

Hyperbolic tangent (tanh) layer

Description

A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.

Creation

Syntax

layer = tanhLayer
layer = tanhLayer('Name',Name)

Description

layer = tanhLayer creates a hyperbolic tangent layer.

layer = tanhLayer('Name',Name) additionally specifies the optional Name property. For example, tanhLayer('Name','tanh1') creates a tanh layer with the name 'tanh1'.

Properties

Name — Layer name

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string
**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**
{'in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

### Examples

**Create Hyperbolic Tangent Layer**

Create a hyperbolic tangent (tanh) layer with the name 'tanh1'.

```matlab
layer = tanhLayer('Name','tanh1')
```

```
layer = 
    TanhLayer with properties:
        Name: 'tanh1'
Show all properties
```
Include a tanh layer in a Layer array.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16)
    batchNormalizationLayer
    tanhLayer

    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32)
    batchNormalizationLayer
    tanhLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
]
```

**Extended Capabilities**

**GPU Code Generation**
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.
See Also
batchNormalizationLayer | clippedReluLayer | leakyReluLayer | reluLayer | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

Introduced in R2019a
batchNormalizationLayer

Batch normalization layer

Description

A batch normalization layer normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

The layer first normalizes the activations of each channel by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Then, the layer shifts the input by a learnable offset $\beta$ and scales it by a learnable scale factor $\gamma$.

Creation

Syntax

layer = batchNormalizationLayer
layer = batchNormalizationLayer('Name',Value)

Description

layer = batchNormalizationLayer creates a batch normalization layer.

layer = batchNormalizationLayer('Name',Value) creates a batch normalization layer and sets the optional “Batch Normalization” on page 1-1047, “Parameters and Initialization” on page 1-1048, “Learn Rate and Regularization” on page 1-1049, and Name properties using name-value pairs. For example, batchNormalizationLayer('Name','batchnorm') creates a batch normalization layer with the name 'batchnorm'. You can specify multiple name-value pairs. Enclose each property name in single quotes.
Properties

Batch Normalization

**TrainedMean — Input mean**
numeric array

Input mean of each channel, specified as a numeric array of size 1-by-1-by-NumChannels for 2-D input or 1-by-1-by-1-by-NumChannels for 3-D input.

After network training finishes, the software calculates the input mean over the entire training data set. The layer uses TrainedMean (in place of the mini-batch mean) to normalize the input during prediction.

**TrainedVariance — Input variance**
numeric array

Input variance of each channel, specified as a numeric array of size 1-by-1-by-NumChannels for 2-D input or 1-by-1-by-1-by-NumChannels for 3-D input.

After network training finishes, the software calculates the input variance over the entire training data set. The layer uses TrainedVariance (in place of the mini-batch variance) to normalize the input during prediction.

**Epsilon — Constant to add to mini-batch variances**
1e-5 (default) | numeric scalar

Constant to add to the mini-batch variances, specified as a numeric scalar equal to or larger than 1e-5.

The batch normalization layer adds this constant to the mini-batch variances before normalization to ensure numerical stability and avoid division by zero.

**NumChannels — Number of input channels**
'auto' (default) | positive integer

Number of input channels, specified as 'auto' or a positive integer.

This property is always equal to the number of channels of the input to the layer. If NumChannels equals 'auto', then the software infers the correct value for the number of channels at training time.
Parameters and Initialization

**ScaleInitializer — Function to initialize channel scale factors**

'ones' (default) | 'zeros' | 'narrow-normal' | function handle

Function to initialize the channel scale factors, specified as one of the following:

- 'ones' - Initialize the channel scale factors with ones.
- 'zeros' - Initialize the channel scale factors with zeros.
- 'narrow-normal' - Initialize the channel scale factors by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle - Initialize the channel scale factors with a custom function. If you specify a function handle, then the function must be of the form `scale = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel scale factors when the `Scale` property is empty.

Data Types: `char` | `string` | `function_handle`

**OffsetInitializer — Function to initialize channel offsets**

'zeros' (default) | 'ones' | 'narrow-normal' | function handle

Function to initialize the channel offsets, specified as one of the following:

- 'zeros' - Initialize the channel offsets with zeros.
- 'ones' - Initialize the channel offsets with ones.
- 'narrow-normal' - Initialize the channel offsets by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- Function handle - Initialize the channel offsets with a custom function. If you specify a function handle, then the function must be of the form `offset = func(sz)`, where `sz` is the size of the scale. For an example, see “Specify Custom Weight Initialization Function”.

The layer only initializes the channel offsets when the `Offset` property is empty.

Data Types: `char` | `string` | `function_handle`

**Scale — Channel scale factors**

[] (default) | numeric array
Channel scale factors $\gamma$, specified as a numeric array.

The channel scale factors are learnable parameters. When training a network, if Scale is nonempty, then trainNetwork uses the Scale property as the initial value. If Scale is empty, then trainNetwork uses the initializer specified by ScaleInitializer.

At training time:
- For 2-D input, Scale is a 1-by-1-by-NumChannels array of positive numbers.
- For 3-D input, Scale is a 1-by-1-by-1-by-NumChannels array of positive numbers.

**Offset — Channel offsets**

[] (default) | numeric array

Channel offsets $\beta$, specified as a numeric array.

The channel offsets are learnable parameters. When training a network, if Offset is nonempty, then trainNetwork uses the Offset property as the initial value. If Offset is empty, then trainNetwork uses the initializer specified by OffsetInitializer.

At training time:
- For 2-D input, Offset is a 1-by-1-by-NumChannels array.
- For 3-D input, Offset is a 1-by-1-by-1-by-NumChannels array.

**Learn Rate and Regularization**

**ScaleLearnRateFactor — Learning rate factor for scale factors**

1 (default) | nonnegative scalar

Learning rate factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global learning rate to determine the learning rate for the scale factors in a layer. For example, if ScaleLearnRateFactor is 2, then the learning rate for the scale factors in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

**OffsetLearnRateFactor — Learning rate factor for offsets**

1 (default) | nonnegative scalar

Learning rate factor for the offsets, specified as a nonnegative scalar.
The software multiplies this factor by the global learning rate to determine the learning rate for the offsets in a layer. For example, if OffsetLearnRateFactor equals 2, then the learning rate for the offsets in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

**ScaleL2Factor** — L\(_2\) regularization factor for scale factors

1 (default) | nonnegative scalar

L\(_2\) regularization factor for the scale factors, specified as a nonnegative scalar.

The software multiplies this factor by the global L\(_2\) regularization factor to determine the learning rate for the scale factors in a layer. For example, if ScaleL2Factor is 2, then the L\(_2\) regularization for the offsets in the layer is twice the global L\(_2\) regularization factor. You can specify the global L\(_2\) regularization factor using the trainingOptions function.

**OffsetL2Factor** — L\(_2\) regularization factor for offsets

1 (default) | nonnegative scalar

L\(_2\) regularization factor for the offsets, specified as a nonnegative scalar.

The software multiplies this factor by the global L\(_2\) regularization factor to determine the learning rate for the offsets in a layer. For example, if OffsetL2Factor is 2, then the L\(_2\) regularization for the offsets in the layer is twice the global L\(_2\) regularization factor. You can specify the global L\(_2\) regularization factor using the trainingOptions function.

**Layer**

**Name** — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to ' ', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs** — Number of inputs

1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**

`{'in'}` (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

`{'out'}` (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

### Examples

**Create Batch Normalization Layer**

Create a batch normalization layer with the name 'BN1'.

```matlab
layer = batchNormalizationLayer('Name','BN1')
```

```matlab
layer = BatchNormalizationLayer with properties:

    Name: 'BN1'
    NumChannels: 'auto'
    TrainedMean: []
    TrainedVariance: []

    Hyperparameters
        Epsilon: 1.0000e-05
```
Include batch normalization layers in a Layer array.

```matlab
layers = [
    imageInputLayer([32 32 3])
    convolution2dLayer(3,16,'Padding',1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding',1)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
];
```

layers =
11x1 Layer array with layers:

1   'Image Input' 32x32x3 images with 'zerocenter' normalization
2   'Convolution' 16 3x3 convolutions with stride [1 1] and padding [1 1 1 1]
3   'Batch Normalization' Batch normalization
4   'ReLU' ReLU
5   'Max Pooling' 2x2 max pooling with stride [2 2] and padding [0 0 0 0]
6   'Convolution' 32 3x3 convolutions with stride [1 1] and padding [1 1 1 1]
7   'Batch Normalization' Batch normalization
8   'ReLU' ReLU
9   'Fully Connected' 10 fully connected layer
10  'Softmax' softmax
11  'Classification Output' crossentropyex
More About

Batch Normalization Layer

A batch normalization layer normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

The layer first normalizes the activations of each channel by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Then, the layer shifts the input by a learnable offset $\beta$ and scales it by a learnable scale factor $\gamma$. $\beta$ and $\gamma$ are themselves learnable parameters that are updated during network training.

Batch normalization layers normalize the activations and gradients propagating through a neural network, making network training an easier optimization problem. To take full advantage of this fact, you can try increasing the learning rate. Since the optimization problem is easier, the parameter updates can be larger and the network can learn faster. You can also try reducing the $L_2$ and dropout regularization. With batch normalization layers, the activations of a specific image during training depend on which images happen to appear in the same mini-batch. To take full advantage of this regularizing effect, try shuffling the training data before every training epoch. To specify how often to shuffle the data during training, use the 'Shuffle' name-value pair argument of trainingOptions.

Algorithms

A batch normalization normalizes its inputs $x_i$ by first calculating the mean $\mu_B$ and variance $\sigma_B^2$ over a mini-batch and over each input channel. Then, it calculates the normalized activations as

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}.$$ 

Here, $\epsilon$ (the property Epsilon) improves numerical stability when the mini-batch variance is very small. To allow for the possibility that inputs with zero mean and unit variance are not optimal for the layer that follows the batch normalization layer, the batch normalization layer further shifts and scales the activations as
\[ y_i = \gamma \bar{x}_i + \beta. \]

Here, the offset \( \beta \) and scale factor \( \gamma \) (Offset and Scale properties) are learnable parameters that are updated during network training.

When network training finishes, the batch normalization layer calculates the mean and variance over the full training set and stores them in the \texttt{TrainedMean} and \texttt{TrainedVariance} properties. When you use a trained network to make predictions on new images, the layer uses the trained mean and variance instead of the mini-batch mean and variance to normalize the activations.

**References**


**Extended Capabilities**

**GPU Code Generation**
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

**See Also**

\texttt{convolution2dLayer} | \texttt{fullyConnectedLayer} | \texttt{reluLayer} | \texttt{trainNetwork} | \texttt{trainingOptions}

**Topics**

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

**Introduced in R2017b**
softmaxLayer

Softmax layer

**Description**

A softmax layer applies a softmax function to the input.

**Creation**

**Syntax**

```matlab
layer = softmaxLayer
layer = softmaxLayer('Name',Name)
```

**Description**

`layer = softmaxLayer` creates a softmax layer.

`layer = softmaxLayer('Name',Name)` creates a softmax layer and sets the optional `Name` property using a name-value pair. For example, `softmaxLayer('Name','sm1')` creates a softmax layer with the name 'sm1'. Enclose the property name in single quotes.

**Properties**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to ' ', then the software automatically assigns a name to the layer at training time.
Data Types: char | string

**NumInputs — Number of inputs**
1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

**InputNames — Input names**
{'in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

**NumOutputs — Number of outputs**
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create Softmax Layer**

Create a softmax layer with the name 'sm1'.

```matlab
layer = softmaxLayer('Name','sm1')
```

```matlab
layer = SoftmaxLayer with properties:
    Name: 'sm1'
```
Include a softmax layer in a Layer array.

```
layers = [ ...  
    imageInputLayer([28 28 1])  
    convolution2dLayer(5,20)  
    reluLayer  
    maxPooling2dLayer(2,'Stride',2)  
    fullyConnectedLayer(10)  
    softmaxLayer  
    classificationLayer]  
```

```
layers =  
7x1 Layer array with layers:  
1   ''   Image Input             28x28x1 images with 'zerocenter' normalization  
2   ''   Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0  0  0]  
3   ''   ReLU                    ReLU  
4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0  0  0]  
5   ''   Fully Connected         10 fully connected layer  
6   ''   Softmax                 softmax  
7   ''   Classification Output   crossentropyex  
```

**More About**

**Softmax Layer**

A softmax layer applies a softmax function to the input.

For classification problems, a softmax layer and then a classification layer must follow the final fully connected layer.

The output unit activation function is the softmax function:

\[ y_r(x) = \frac{\exp(a_r(x))}{\sum_{j=1}^{k} \exp(a_j(x))}, \]

where \( 0 \leq y_r \leq 1 \) and \( \sum_{j=1}^{k} y_j = 1 \).
The softmax function is the output unit activation function after the last fully connected layer for multi-class classification problems:

\[
P(c_r|x, \theta) = \frac{P(x, \theta|c_r)P(c_r)}{\sum_{j=1}^{k} P(x, \theta|c_j)P(c_j)} = \frac{\exp(a_r(x, \theta))}{\sum_{j=1}^{k} \exp(a_j(x, \theta))},
\]

where \(0 \leq P(c_r|x, \theta) \leq 1\) and \(\sum_{j=1}^{k} P(c_j|x, \theta) = 1\). Moreover, \(a_r = \ln(P(x, \theta|c_r)P(c_r)), P(x, \theta|c_r)\) is the conditional probability of the sample given class \(r\), and \(P(c_r)\) is the class prior probability.

The softmax function is also known as the normalized exponential and can be considered the multi-class generalization of the logistic sigmoid function [1].

References


Extended Capabilities

**GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

**See Also**

classificationLayer | convolution2dLayer | fullyConnectedLayer | trainNetwork

**Topics**

“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”

**Introduced in R2016a**
bilstmLayer

Bidirectional long short-term memory (BiLSTM) layer

Description

A bidirectional LSTM (BiLSTM) layer learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the network to learn from the complete time series at each time step.

Creation

Syntax

layer = bilstmLayer(numHiddenUnits)
layer = bilstmLayer(numHiddenUnits,Name,Value)

Description

layer = bilstmLayer(numHiddenUnits) creates a bidirectional LSTM layer and sets the NumHiddenUnits property.

layer = bilstmLayer(numHiddenUnits,Name,Value) sets additional OutputMode, “Activations” on page 1-1062, “Parameters and Initialization” on page 1-1063, “Learn Rate and Regularization” on page 1-1067, and Name properties using one or more name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in quotes.
Properties

**BiLSTM**

**NumHiddenUnits — Number of hidden units**

positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

The number of hidden units corresponds to the amount of information remembered between time steps (the hidden state). The hidden state can contain information from all previous time steps, regardless of the sequence length. If the number of hidden units is too large, then the layer might overfit to the training data. This value can vary from a few dozen to a few thousand.

The hidden state does not limit the number of time steps are processed in an iteration. To split your sequences into smaller sequences for training, use the 'SequenceLength' option in trainingOptions.

Example: 200

**OutputMode — Format of output**

'sequence' (default) | 'last'

Format of output, specified as one of the following:

- 'sequence' - Output the complete sequence.
- 'last' - Output the last time step of the sequence.

**InputSize — Input size**

'auto' (default) | positive integer

Input size, specified as a positive integer or 'auto'. If InputSize is 'auto', then the software automatically assigns the input size at training time.

Example: 100
Activations

StateActivationFunction — Activation function to update the cell and hidden state
'tanh' (default) | 'softsign'

Activation function to update the cell and hidden state, specified as one of the following:

- 'tanh' - Use the hyperbolic tangent function (tanh).
- 'softsign' - Use the softsign function softsign(x) = \( \frac{x}{1 + |x|} \).

The layer uses this option as the function \( \sigma_c \) in the calculations to update the cell and hidden state. For more information on how activation functions are used in an LSTM layer, see “Long Short-Term Memory Layer” on page 1-1096.

GateActivationFunction — Activation function to apply to the gates
'sigmoid' (default) | 'hard-sigmoid'

Activation function to apply to the gates, specified as one of the following:

- 'sigmoid' - Use the sigmoid function \( \sigma(x) = (1 + e^{-x})^{-1} \).
- 'hard-sigmoid' - Use the hard sigmoid function

\[
\sigma(x) = \begin{cases} 
0 & \text{if } x < -2.5 \\
0.2x + 0.5 & \text{if } -2.5 \leq x \leq 2.5 \\
1 & \text{if } x > 2.5 
\end{cases}
\]

The layer uses this option as the function \( \sigma_g \) in the calculations for the input, output, and forget gate. For more information on how activation functions are used in an LSTM layer, see “Long Short-Term Memory Layer” on page 1-1096.

State

CellState — Initial value of cell state
numeric vector

Initial value of the cell state, specified as a 2*NumHiddenUnits-by-1 numeric vector. This value corresponds to the cell state at time step 0.
After setting this property, calls to the `resetState` function set the cell state to this value.

**HiddenState — Initial value of hidden state**

numeric vector

Initial value of the hidden state, specified as a 2*NumHiddenUnits-by-1 numeric vector. This value corresponds to the hidden state at time step 0.

After setting this property, calls to the `resetState` function set the hidden state to this value.

**Parameters and Initialization**

**InputWeightsInitializer — Function to initialize input weights**

'glorot' (default) | 'he' | 'orthogonal' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the input weights, specified as one of the following:

- 'glorot' - Initialize the input weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance \(2/(\text{InputSize} + \text{numOut})\), where \(\text{numOut} = 8\times\text{NumHiddenUnits}\).
- 'he' - Initialize the input weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance \(2/\text{InputSize}\).
- 'orthogonal' - Initialize the input weights with \(Q\), the orthogonal matrix given by the QR decomposition of \(Z = QR\) for a random matrix \(Z\) sampled from a unit normal distribution. [3]
- 'narrow-normal' - Initialize the input weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- 'zeros' - Initialize the input weights with zeros.
- 'ones' - Initialize the input weights with ones.
- Function handle - Initialize the input weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the input weights.

The layer only initializes the input weights when the `InputWeights` property is empty.

Data Types: char | string | function_handle
RecurrentWeightsInitializer — Function to initialize recurrent weights
'vectorial' (default) | 'glorot' | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the recurrent weights, specified as one of the following:

- 'orthogonal' - Initialize the input weights with $Q$, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix $Z$ sampled from a unit normal distribution. [3]
- 'glorot' - Initialize the recurrent weights with the Glorot initializer [1] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance $2/(numIn + numOut)$, where $numIn = \text{NumHiddenUnits}$ and $numOut = 8*\text{NumHiddenUnits}$.
- 'he' - Initialize the recurrent weights with the He initializer [2]. The He initializer samples from a normal distribution with zero mean and variance $2/\text{NumHiddenUnits}$.
- 'narrow-normal' - Initialize the recurrent weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- 'zeros' - Initialize the recurrent weights with zeros.
- 'ones' - Initialize the recurrent weights with ones.
- Function handle - Initialize the recurrent weights with a custom function. If you specify a function handle, then the function must be of the form $weights = func(sz)$, where $sz$ is the size of the recurrent weights.

The layer only initializes the recurrent weights when the RecurrentWeights property is empty.

Data Types: char | string | function_handle

BiasInitializer — Function to initialize bias
'vectorial' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

- 'unit-forget-gate' - Initialize the forget gate bias with ones and the remaining biases with zeros.
- 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- 'ones' - Initialize the bias with ones.
• Function handle – Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form bias = func(sz), where sz is the size of the bias.

The layer only initializes the bias when the Bias property is empty.

Data Types: char | string | function_handle

**InputWeights — Input weights**

This property lets you specify the input weights as a matrix.

Input weights, specified as a matrix.

The input weight matrix is a concatenation of the eight input weight matrices for the components (gates) in the bidirectional LSTM layer. The eight matrices are concatenated vertically in the following order:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

The input weights are learnable parameters. When training a network, if InputWeights is nonempty, then trainNetwork uses the InputWeights property as the initial value. If InputWeights is empty, then trainNetwork uses the initializer specified by InputWeightsInitializer.

At training time, InputWeights is an 8*NumHiddenUnits-by-InputSize matrix.

**RecurrentWeights — Recurrent weights**

This property lets you specify the recurrent weights as a matrix.

Recurrent weights, specified as a matrix.

The recurrent weight matrix is a concatenation of the eight recurrent weight matrices for the components (gates) in the bidirectional LSTM layer. The eight matrices are concatenated vertically in the following order:
The recurrent weights are learnable parameters. When training a network, if `RecurrentWeights` is nonempty, then `trainNetwork` uses the `RecurrentWeights` property as the initial value. If `RecurrentWeights` is empty, then `trainNetwork` uses the initializer specified by `RecurrentWeightsInitializer`.

At training time, `RecurrentWeights` is an $8 \times \text{NumHiddenUnits}$-by-$\text{NumHiddenUnits}$ matrix.

**Bias — Layer biases**

[] (default) | numeric vector

Layer biases, specified as a numeric vector.

The bias vector is a concatenation of the eight bias vectors for the components (gates) in the bidirectional LSTM layer. The eight vectors are concatenated vertically in the following order:

1 Input gate (Forward)
2 Forget gate (Forward)
3 Cell candidate (Forward)
4 Output gate (Forward)
5 Input gate (Backward)
6 Forget gate (Backward)
7 Cell candidate (Backward)
8 Output gate (Backward)
The layer biases are learnable parameters. When training a network, if **Bias** is nonempty, then **trainNetwork** uses the **Bias** property as the initial value. If **Bias** is empty, then **trainNetwork** uses the initializer specified by **BiasInitializer**.

At training time, **Bias** is an `8*NumHiddenUnits-by-1` numeric vector.

**Learn Rate and Regularization**

**InputWeightsLearnRateFactor** — Learning rate factor for input weights

```
1 (default) | numeric scalar | 1-by-8 numeric vector
```

Learning rate factor for the input weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the input weights of the layer. For example, if **InputWeightsLearnRateFactor** is 2, then the learning rate factor for the input weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the **trainingOptions** function.

To control the value of the learning rate factor for the four individual matrices in **InputWeights**, assign a 1-by-8 vector, where the entries correspond to the learning rate factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 0.1

**RecurrentWeightsLearnRateFactor** — Learning rate factor for recurrent weights

```
1 (default) | numeric scalar | 1-by-8 numeric vector
```

The software multiplies this factor by the global learning rate to compute the learning rate factor for the recurrent weights of the layer.
Learning rate factor for the recurrent weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the recurrent weights of the layer. For example, if `RecurrentWeightsLearnRateFactor` is 2, then the learning rate for the recurrent weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learn rate for the four individual matrices in `RecurrentWeights`, assign a 1-by-8 vector, where the entries correspond to the learning rate factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 0.1
Example: [1 2 1 1 1 2 1 1]

**BiasLearnRateFactor — Learning rate factor for biases**

1 (default) | nonnegative scalar | 1-by-8 numeric vector

Learning rate factor for the biases, specified as a nonnegative scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.
To control the value of the learning rate factor for the four individual matrices in **Bias**, assign a 1-by-8 vector, where the entries correspond to the learning rate factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
Example: [1 2 1 1 2 1 1]

**InputWeightsL2Factor — L2 regularization factor for input weights**

1 (default) | numeric scalar | 1-by-8 numeric vector

L2 regularization factor for the input weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the input weights of the layer. For example, if **InputWeightsL2Factor** is 2, then the L2 regularization factor for the input weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the **trainingOptions** function.

To control the value of the L2 regularization factor for the four individual matrices in **InputWeights**, assign a 1-by-8 vector, where the entries correspond to the L2 regularization factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4 Output gate (Forward)
5 Input gate (Backward)
6 Forget gate (Backward)
7 Cell candidate (Backward)
8 Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 0.1
Example: [1 2 1 1 1 2 1 1]

**RecurrentWeightsL2Factor — L2 regularization factor for recurrent weights**

1 (default) | numeric scalar | 1-by-8 numeric vector

L2 regularization factor for the recurrent weights, specified as a numeric scalar or a 1-by-8 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the recurrent weights of the layer. For example, if `RecurrentWeightsL2Factor` is 2, then the L2 regularization factor for the recurrent weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `RecurrentWeights`, assign a 1-by-8 vector, where the entries correspond to the L2 regularization factor of the following:

1 Input gate (Forward)
2 Forget gate (Forward)
3 Cell candidate (Forward)
4 Output gate (Forward)
5 Input gate (Backward)
6 Forget gate (Backward)
7 Cell candidate (Backward)
8 Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.
Example: 0.1

Example: [1 2 1 1 1 2 1 1]

**BiasL2Factor — L2 regularization factor for biases**

0 (default) | nonnegative scalar | 1-by-8 numeric vector

L2 regularization factor for the biases, specified as a nonnegative scalar.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

To control the value of the L2 regularization factor for the four individual matrices in `Bias`, assign a 1-by-8 vector, where the entries correspond to the L2 regularization factor of the following:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Cell candidate (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Cell candidate (Backward)
8. Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: [1 2 1 1 1 2 1 1]

**Layer**

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. If `Name` is set to ' ', then the software automatically assigns a name at training time.
Data Types: char | string

NumInputs — Number of inputs
1 (default)

Number of inputs of the layer. This layer accepts a single input only.
Data Types: double

InputNames — Input names
{'in'} (default)

Input names of the layer. This layer accepts a single input only.
Data Types: cell

NumOutputs — Number of outputs
1 (default)

Number of outputs of the layer. This layer has a single output only.
Data Types: double

OutputNames — Output names
{'out'} (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

Examples

Create Bidirectional LSTM Layer

Create a bidirectional LSTM layer with the name 'bilstm1' and 100 hidden units.

layer = bilstmLayer(100,'Name','bilstm1')

layer = 
    BiLSTMLayer with properties:

        Name: 'bilstm1'
Hyperparameters
  InputSize: 'auto'
  NumHiddenUnits: 100
  OutputMode: 'sequence'
  StateActivationFunction: 'tanh'
  GateActivationFunction: 'sigmoid'

Learnable Parameters
  InputWeights: []
  RecurrentWeights: []
  Bias: []

State Parameters
  HiddenState: []
  CellState: []

Show all properties

Include a bidirectional LSTM layer in a Layer array.

inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
  sequenceInputLayer(inputSize)
  bilstmLayer(numHiddenUnits)
  fullyConnectedLayer(numClasses)
  softmaxLayer
  classificationLayer]

layers =
  5x1 Layer array with layers:

  1   ''   Sequence Input   Sequence input with 12 dimensions
  2   ''   BiLSTM          BiLSTM with 100 hidden units
  3   ''   Fully Connected 9 fully connected layer
  4   ''   Softmax         softmax
  5   ''   Classification Output crossentropyex
 Compatibility Considerations

**Default input weights initialization is Glorot**

*Behavior changed in R2019a*

Starting in R2019a, the software, by default, initializes the layer input weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer input weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'InputWeightsInitializer' option of the layer to 'narrow-normal'.

**Default recurrent weights initialization is orthogonal**

*Behavior changed in R2019a*

Starting in R2019a, the software, by default, initializes the layer recurrent weights of this layer with $Q$, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix $Z$ sampled from a unit normal distribution. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer recurrent weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'RecurrentWeightsInitializer' option of the layer to 'narrow-normal'.

**References**


**See Also**
classifyAndUpdateState | flattenLayer | lstmLayer | predictAndUpdateState | resetState | sequenceFoldingLayer | sequenceInputLayer | sequenceUnfoldingLayer

**Topics**
“Sequence Classification Using Deep Learning”
“Classify Videos Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Compare Layer Weight Initializers”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

**Introduced in R2018a**
**lstmLayer**

Long short-term memory (LSTM) layer

**Description**

An LSTM layer learns long-term dependencies between time steps in time series and sequence data.

The layer performs additive interactions, which can help improve gradient flow over long sequences during training.

**Creation**

**Syntax**

```matlab
text = lstmLayer(numHiddenUnits)
text = lstmLayer(numHiddenUnits,Name,Value)
```

**Description**

`text = lstmLayer(numHiddenUnits)` creates an LSTM layer and sets the `NumHiddenUnits` property.

`text = lstmLayer(numHiddenUnits,Name,Value)` sets additional `OutputMode`, “Activations” on page 1-1078, “State” on page 1-1078, “Parameters and Initialization” on page 1-1079, “Learn Rate and Regularization” on page 1-1082, and `Name` properties using one or more name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in quotes.
Properties

LSTM

NumHiddenUnits — Number of hidden units

positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

The number of hidden units corresponds to the amount of information remembered between time steps (the hidden state). The hidden state can contain information from all previous time steps, regardless of the sequence length. If the number of hidden units is too large, then the layer might overfit to the training data. This value can vary from a few dozen to a few thousand.

The hidden state does not limit the number of time steps are processed in an iteration. To split your sequences into smaller sequences for training, use the 'SequenceLength' option in trainingOptions.

Example: 200

OutputMode — Format of output

'sequence' (default) | 'last'

Format of output, specified as one of the following:

- 'sequence' - Output the complete sequence.
- 'last' - Output the last time step of the sequence.

InputSize — Input size

'auto' (default) | positive integer

Input size, specified as a positive integer or 'auto'. If InputSize is 'auto', then the software automatically assigns the input size at training time.

Example: 100
Activations

**StateActivationFunction — Activation function to update the cell and hidden state**
'tanh' (default) | 'softsign'

Activation function to update the cell and hidden state, specified as one of the following:

- 'tanh' - Use the hyperbolic tangent function ($\tanh$).
- 'softsign' - Use the softsign function $\text{softsign}(x) = \frac{x}{1 + |x|}$.

The layer uses this option as the function $\sigma_c$ in the calculations to update the cell and hidden state. For more information on how activation functions are used in an LSTM layer, see “Long Short-Term Memory Layer” on page 1-1096.

**GateActivationFunction — Activation function to apply to the gates**
'sigmoid' (default) | 'hard-sigmoid'

Activation function to apply to the gates, specified as one of the following:

- 'sigmoid' - Use the sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$.
- 'hard-sigmoid' - Use the hard sigmoid function

$$
\sigma(x) = \begin{cases} 
0 & \text{if } x < -2.5 \\
0.2x + 0.5 & \text{if } -2.5 \leq x \leq 2.5 \\
1 & \text{if } x > 2.5 
\end{cases}
$$

The layer uses this option as the function $\sigma_g$ in the calculations for the input, output, and forget gate. For more information on how activation functions are used in an LSTM layer, see “Long Short-Term Memory Layer” on page 1-1096.

State

**CellState — Initial value of cell state**
numeric vector

Initial value of the cell state, specified as a NumHiddenUnits-by-1 numeric vector. This value corresponds to the cell state at time step 0.
After setting this property, calls to the resetState function set the cell state to this value.

**HiddenState — Initial value of the hidden state**

numeric vector

Initial value of the hidden state, specified as a NumHiddenUnits-by-1 numeric vector. This value corresponds to the hidden state at time step 0.

After setting this property, calls to the resetState function set the hidden state to this value.

**Parameters and Initialization**

**InputWeightsInitializer — Function to initialize input weights**

'glorot' (default) | 'he' | 'orthogonal' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the input weights, specified as one of the following:

- 'glorot' - Initialize the input weights with the Glorot initializer [4] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance $2/(\text{InputSize} + \text{numOut})$, where $\text{numOut} = 4*\text{NumHiddenUnits}$.
- 'he' - Initialize the input weights with the He initializer [5]. The He initializer samples from a normal distribution with zero mean and variance $2/\text{InputSize}$.
- 'orthogonal' - Initialize the input weights with $Q$, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix $Z$ sampled from a unit normal distribution. [6]
- 'narrow-normal' - Initialize the input weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.
- 'zeros' - Initialize the input weights with zeros.
- 'ones' - Initialize the input weights with ones.
- Function handle - Initialize the input weights with a custom function. If you specify a function handle, then the function must be of the form `weights = func(sz)`, where `sz` is the size of the input weights.

The layer only initializes the input weights when the InputWeights property is empty.

Data Types: char | string | function_handle
**RecurrentWeightsInitializer — Function to initialize recurrent weights**

'orthogonal' (default) | 'glorot' | 'he' | 'narrow-normal' | 'zeros' | 'ones' | function handle

Function to initialize the recurrent weights, specified as one of the following:

- 'orthogonal' - Initialize the recurrent weights with $Q$, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix $Z$ sampled from a unit normal distribution. [6]

- 'glorot' - Initialize the recurrent weights with the Glorot initializer [4] (also known as Xavier initializer). The Glorot initializer independently samples from a uniform distribution with zero mean and variance $2/(\text{numIn} + \text{numOut})$, where $\text{numIn} = \text{NumHiddenUnits}$ and $\text{numOut} = 4\times\text{NumHiddenUnits}$.

- 'he' - Initialize the recurrent weights with the He initializer [5]. The He initializer samples from a normal distribution with zero mean and variance $2/\text{NumHiddenUnits}$.

- 'narrow-normal' - Initialize the recurrent weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

- 'zeros' - Initialize the recurrent weights with zeros.

- 'ones' - Initialize the recurrent weights with ones.

- Function handle - Initialize the recurrent weights with a custom function. If you specify a function handle, then the function must be of the form $\text{weights} = \text{func}(\text{sz})$, where $\text{sz}$ is the size of the recurrent weights.

The layer only initializes the recurrent weights when the RecurrentWeights property is empty.

Data Types: char | string | function_handle

**BiasInitializer — Function to initialize bias**

'unit-forget-gate' (default) | 'narrow-normal' | 'ones' | function handle

Function to initialize the bias, specified as one of the following:

- 'unit-forget-gate' - Initialize the forget gate bias with ones and the remaining biases with zeros.

- 'narrow-normal' - Initialize the bias by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

- 'ones' - Initialize the bias with ones.
Function handle - Initialize the bias with a custom function. If you specify a function handle, then the function must be of the form \( \text{bias} = \text{func}(sz) \), where \( sz \) is the size of the bias.

The layer only initializes the bias when the Bias property is empty.

Data Types: char | string | function_handle

**InputWeights — Input weights**

| [] (default) | matrix |

Input weights, specified as a matrix.

The input weight matrix is a concatenation of the four input weight matrices for the components (gates) in the LSTM layer. The four matrices are concatenated vertically in the following order:

1. Input gate
2. Forget gate
3. Cell candidate
4. Output gate

The input weights are learnable parameters. When training a network, if InputWeights is nonempty, then trainNetwork uses the InputWeights property as the initial value. If InputWeights is empty, then trainNetwork uses the initializer specified by InputWeightsInitializer.

At training time, InputWeights is a \( 4 \times \text{NumHiddenUnits} \)-by-\( \text{InputSize} \) matrix.

**RecurrentWeights — Recurrent weights**

| [] (default) | matrix |

Recurrent weights, specified as a matrix.

The recurrent weight matrix is a concatenation of the four recurrent weight matrices for the components (gates) in the LSTM layer. The four matrices are vertically concatenated in the following order:

1. Input gate
2. Forget gate
3. Cell candidate
Output gate

The recurrent weights are learnable parameters. When training a network, if RecurrentWeights is nonempty, then trainNetwork uses the RecurrentWeights property as the initial value. If RecurrentWeights is empty, then trainNetwork uses the initializer specified by RecurrentWeightsInitializer.

At training time RecurrentWeights is a 4*NumHiddenUnits-by-NumHiddenUnits matrix.

Bias — Layer biases
[] (default) | numeric vector

Layer biases for the LSTM layer, specified as a numeric vector.

The bias vector is a concatenation of the four bias vectors for the components (gates) in the LSTM layer. The four vectors are concatenated vertically in the following order:

1 Input gate
2 Forget gate
3 Cell candidate
4 Output gate

The layer biases are learnable parameters. When training a network, if Bias is nonempty, then trainNetwork uses the Bias property as the initial value. If Bias is empty, then trainNetwork uses the initializer specified by BiasInitializer.

At training time, Bias is a 4*NumHiddenUnits-by-1 numeric vector.

Learn Rate and Regularization

InputWeightsLearnRateFactor — Learning rate factor for input weights
1 (default) | numeric scalar | 1-by-4 numeric vector

Learning rate factor for the input weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate factor for the input weights of the layer. For example, if InputWeightsLearnRateFactor is 2, then the learning rate factor for the input...
weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

To control the value of the learning rate factor for the four individual matrices in InputWeights, specify a 1-by-4 vector. The entries of InputWeightsLearnRateFactor correspond to the learning rate factor of the following:

1  Input gate
2  Forget gate
3  Cell candidate
4  Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
Example: [1 2 1 1]

RecurrentWeightsLearnRateFactor — Learning rate factor for recurrent weights
1 (default) | numeric scalar | 1-by-4 numeric vector

Learning rate factor for the recurrent weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the recurrent weights of the layer. For example, if RecurrentWeightsLearnRateFactor is 2, then the learning rate for the recurrent weights of the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the trainingOptions function.

To control the value of the learning rate factor for the four individual matrices in RecurrentWeights, specify a 1-by-4 vector. The entries of RecurrentWeightsLearnRateFactor correspond to the learning rate factor of the following:

1  Input gate
2  Forget gate
3  Cell candidate
Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
     Example: [1 2 1 1]

**BiasLearnRateFactor — Learning rate factor for biases**

1 (default) | nonnegative scalar | 1-by-4 numeric vector

Learning rate factor for the biases, specified as a nonnegative scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global learning rate to determine the learning rate for the biases in this layer. For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in the layer is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

To control the value of the learning rate factor for the four individual matrices in `Bias`, specify a 1-by-4 vector. The entries of `BiasLearnRateFactor` correspond to the learning rate factor of the following:

1 Input gate
2 Forget gate
3 Cell candidate
4 Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
     Example: [1 2 1 1]

**InputWeightsL2Factor — L2 regularization factor for input weights**

1 (default) | numeric scalar | 1-by-4 numeric vector

L2 regularization factor for the input weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the input weights of the layer. For example, if
InputWeightsL2Factor is 2, then the L2 regularization factor for the input weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the trainingOptions function.

To control the value of the L2 regularization factor for the four individual matrices in InputWeights, specify a 1-by-4 vector. The entries of InputWeightsL2Factor correspond to the L2 regularization factor of the following:

1 Input gate
2 Forget gate
3 Cell candidate
4 Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
Example: [1 2 1 1]

RecurrentWeightsL2Factor — L2 regularization factor for recurrent weights
1 (default) | numeric scalar | 1-by-4 numeric vector

L2 regularization factor for the recurrent weights, specified as a numeric scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization factor for the recurrent weights of the layer. For example, if RecurrentWeightsL2Factor is 2, then the L2 regularization factor for the recurrent weights of the layer is twice the current global L2 regularization factor. The software determines the L2 regularization factor based on the settings specified with the trainingOptions function.

To control the value of the L2 regularization factor for the four individual matrices in RecurrentWeights, specify a 1-by-4 vector. The entries of RecurrentWeightsL2Factor correspond to the L2 regularization factor of the following:

1 Input gate
2 Forget gate
3 Cell candidate
Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
Example: [1 2 1 1]

BiasL2Factor — L2 regularization factor for biases

0 (default) | nonnegative scalar | 1-by-4 numeric vector

L2 regularization factor for the biases, specified as a nonnegative scalar or a 1-by-4 numeric vector.

The software multiplies this factor by the global L2 regularization factor to determine the L2 regularization for the biases in this layer. For example, if BiasL2Factor is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor.

You can specify the global L2 regularization factor using the trainingOptions function.

To control the value of the L2 regularization factor for the four individual matrices in Bias, specify a 1-by-4 vector. The entries of BiasL2Factor correspond to the L2 regularization factor of the following:

1 Input gate
2 Forget gate
3 Cell candidate
4 Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2
Example: [1 2 1 1]

Layer

Name — Layer name

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. If Name is set to ' ', then the software automatically assigns a name at training time.

Data Types: char | string
**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{''in''} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{''out''} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

## Examples

**Create LSTM Layer**

Create an LSTM layer with the name 'lstm1' and 100 hidden units.

```
layer = lstmLayer(100,'Name','lstm1')
```

```
layer =  
  LSTMLayer with properties: 
    Name: 'lstm1'

  Hyperparameters
```
Include an LSTM layer in a Layer array.

```matlab
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

Show all properties

Train Network for Sequence Classification

Train a deep learning LSTM network for sequence-to-label classification.
Load the Japanese Vowels data set as described in [1] and [2]. XTrain is a cell array containing 270 sequences of varying length with a feature dimension of 12. Y is a categorical vector of labels 1,2,...,9. The entries in XTrain are matrices with 12 rows (one row for each feature) and a varying number of columns (one column for each time step).

[XTrain,YTrain] = japaneseVowelsTrainData;

Visualize the first time series in a plot. Each line corresponds to a feature.

```matlab
figure
plot(XTrain{1}')
title("Training Observation 1")
numFeatures = size(XTrain{1},1);
legend("Feature " + string(1:numFeatures), 'Location','northeastoutside')
```
Define the LSTM network architecture. Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer to have 100 hidden units and to output the last element of the sequence. Finally, specify nine classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer:

```matlab
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
];
```
softmaxLayer
classificationLayer]
layers =
  5x1 Layer array with layers:
  1   ''   Sequence Input          Sequence input with 12 dimensions
  2   ''   LSTM                    LSTM with 100 hidden units
  3   ''   Fully Connected         9 fully connected layer
  4   ''   Softmax                 softmax
  5   ''   Classification Output   crossentropyex

Specify the training options. Specify the solver as 'adam' and 'GradientThreshold' as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 100.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set 'ExecutionEnvironment' to 'cpu'. To train on a GPU, if available, set 'ExecutionEnvironment' to 'auto' (the default value).

maxEpochs = 100;
miniBatchSize = 27;

options = trainingOptions('adam', ...
  'ExecutionEnvironment','cpu', ...
  'MaxEpochs',maxEpochs, ...  
  'MiniBatchSize',miniBatchSize, ... 
  'GradientThreshold',1, ... 
  'Verbose',false, ... 
  'Plots','training-progress');

Train the LSTM network with the specified training options.

net = trainNetwork(XTrain,YTrain,layers,options);

Load the test set and classify the sequences into speakers.

\[ \text{[XTest, YTest]} = \text{japaneseVowelsTestData}; \]

Classify the test data. Specify the same mini-batch size used for training.

\[ \text{YPred} = \text{classify(net, XTest, 'MiniBatchSize', miniBatchSize)}; \]

Calculate the classification accuracy of the predictions.
acc = sum(YPred == YTest)./numel(YTest)

acc = 0.9486

**Classification LSTM Networks**

To create an LSTM network for sequence-to-label classification, create a layer array containing a sequence input layer, an LSTM layer, a fully connected layer, a softmax layer, and a classification output layer.

Set the size of the sequence input layer to the number of features of the input data. Set the size of the fully connected layer to the number of classes. You do not need to specify the sequence length.

For the LSTM layer, specify the number of hidden units and the output mode 'last'.

```matlab
define layers
    numFeatures = 12;
    numHiddenUnits = 100;
    numClasses = 9;
    layers = [ ...
        sequenceInputLayer(numFeatures)
        lstmLayer(numHiddenUnits,'OutputMode','last')
        fullyConnectedLayer(numClasses)
        softmaxLayer
        classificationLayer];
```

For an example showing how to train an LSTM network for sequence-to-label classification and classify new data, see “Sequence Classification Using Deep Learning”.

To create an LSTM network for sequence-to-sequence classification, use the same architecture as for sequence-to-label classification, but set the output mode of the LSTM layer to 'sequence'.

```matlab
define layers
    numFeatures = 12;
    numHiddenUnits = 100;
    numClasses = 9;
    layers = [ ...
        sequenceInputLayer(numFeatures)
        lstmLayer(numHiddenUnits,'OutputMode','sequence')
        fullyConnectedLayer(numClasses)
        softmaxLayer
        classificationLayer];
```
Regression LSTM Networks

To create an LSTM network for sequence-to-one regression, create a layer array containing a sequence input layer, an LSTM layer, a fully connected layer, and a regression output layer.

Set the size of the sequence input layer to the number of features of the input data. Set the size of the fully connected layer to the number of responses. You do not need to specify the sequence length.

For the LSTM layer, specify the number of hidden units and the output mode 'last'.

```matlab
numFeatures = 12;
numHiddenUnits = 125;
numResponses = 1;

layers = [...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

To create an LSTM network for sequence-to-sequence regression, use the same architecture as for sequence-to-one regression, but set the output mode of the LSTM layer to 'sequence'.

```matlab
numFeatures = 12;
numHiddenUnits = 125;
numResponses = 1;

layers = [...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

For an example showing how to train an LSTM network for sequence-to-sequence regression and predict on new data, see “Sequence-to-Sequence Regression Using Deep Learning”. 

Deeper LSTM Networks

You can make LSTM networks deeper by inserting extra LSTM layers with the output mode 'sequence' before the LSTM layer. To prevent overfitting, you can insert dropout layers after the LSTM layers.

For sequence-to-label classification networks, the output mode of the last LSTM layer must be 'last'.

```
numFeatures = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits2,'OutputMode','last')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For sequence-to-sequence classification networks, the output mode of the last LSTM layer must be 'sequence'.

```
numFeatures = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```
More About

Long Short-Term Memory Layer

An LSTM layer learns long-term dependencies between time steps in time series and sequence data.

The state of the layer consists of the hidden state (also known as the output state) and the cell state. The hidden state at time step $t$ contains the output of the LSTM layer for this time step. The cell state contains information learned from the previous time steps. At each time step, the layer adds information to or removes information from the cell state. The layer controls these updates using gates.

The following components control the cell state and hidden state of the layer.

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input gate ($i$)</td>
<td>Control level of cell state update</td>
</tr>
<tr>
<td>Forget gate ($f$)</td>
<td>Control level of cell state reset (forget)</td>
</tr>
<tr>
<td>Cell candidate ($g$)</td>
<td>Add information to cell state</td>
</tr>
<tr>
<td>Output gate ($o$)</td>
<td>Control level of cell state added to hidden state</td>
</tr>
</tbody>
</table>

This diagram illustrates the flow of data at time step $t$. The diagram highlights how the gates forget, update, and output the cell and hidden states.
The learnable weights of an LSTM layer are the input weights $W$ (InputWeights), the recurrent weights $R$ (RecurrentWeights), and the bias $b$ (Bias). The matrices $W$, $R$, and $b$ are concatenations of the input weights, the recurrent weights, and the bias of each component, respectively. These matrices are concatenated as follows:

$$W = \begin{bmatrix} W_i \\ W_f \\ W_g \\ W_o \end{bmatrix}, R = \begin{bmatrix} R_i \\ R_f \\ R_g \\ R_o \end{bmatrix}, b = \begin{bmatrix} b_i \\ b_f \\ b_g \\ b_o \end{bmatrix}$$

where $i$, $f$, $g$, and $o$ denote the input gate, forget gate, cell candidate, and output gate, respectively.

The cell state at time step $t$ is given by

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t,$$

where $\odot$ denotes the Hadamard product (element-wise multiplication of vectors).

The hidden state at time step $t$ is given by

$$h_t = o_t \odot \sigma_c(c_t),$$
where $\sigma_c$ denotes the state activation function. The `lstmLayer` function, by default, uses the hyperbolic tangent function (tanh) to compute the state activation function.

The following formulas describe the components at time step $t$.

<table>
<thead>
<tr>
<th>Component</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input gate</td>
<td>$i_t = \sigma_g(W_i x_t + R_i h_{t-1} + b_i)$</td>
</tr>
<tr>
<td>Forget gate</td>
<td>$f_t = \sigma_g(W_f x_t + R_f h_{t-1} + b_f)$</td>
</tr>
<tr>
<td>Cell candidate</td>
<td>$g_t = \sigma_c(W_g x_t + R_g h_{t-1} + b_g)$</td>
</tr>
<tr>
<td>Output gate</td>
<td>$o_t = \sigma_g(W_o x_t + R_o h_{t-1} + b_o)$</td>
</tr>
</tbody>
</table>

In these calculations, $\sigma_g$ denotes the gate activation function. The `lstmLayer` function, by default, uses the sigmoid function given by $\sigma(x) = (1 + e^{-x})^{-1}$ to compute the gate activation function.

**Compatibility Considerations**

**Default input weights initialization is Glorot**

*Behavior changed in R2019a*

Starting in R2019a, the software, by default, initializes the layer input weights of this layer using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer input weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'InputWeightsInitializer' option of the layer to 'narrow-normal'.

**Default recurrent weights initialization is orthogonal**

*Behavior changed in R2019a*

Starting in R2019a, the software, by default, initializes the layer recurrent weights of this layer with $Q$, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix $Z$ sampled from a unit normal distribution. This behavior helps stabilize training and usually reduces the training time of deep networks.
In previous releases, the software, by default, initializes the layer recurrent weights using the by sampling from a normal distribution with zero mean and variance 0.01. To reproduce this behavior, set the 'RecurrentWeightsInitializer' option of the layer to 'narrow-normal'.

References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:
• For code generation, the `StateActivationFunction` property must be set to 'tanh'.
• For code generation, the `GateActivationFunction` property must be set to 'sigmoid'.

See Also
`bilstmLayer` | `classifyAndUpdateState` | `flattenLayer` | `predictAndUpdateState` | `resetState` | `sequenceFoldingLayer` | `sequenceInputLayer` | `sequenceUnfoldingLayer`

Topics
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Classify Videos Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Compare Layer Weight Initializers”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2017b
sequenceInputLayer

Sequence input layer

Description

A sequence input layer inputs sequence data to a network.

Creation

Syntax

layer = sequenceInputLayer(inputSize)
layer = sequenceInputLayer(inputSize,Name,Value)

Description

layer = sequenceInputLayer(inputSize) creates a sequence input layer and sets the InputSize property.

layer = sequenceInputLayer(inputSize,Name,Value) sets the optional Normalization, Mean, and Name properties using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

Image Input

InputSize — Size of input
positive integer | vector of positive integers

Size of the input, specified as a positive integer or a vector of positive integers.
For vector sequence input, `InputSize` is a scalar corresponding to the number of features.

For 2-D image sequence input, `InputSize` is vector of three elements \([h \ w \ c]\), where \(h\) is the image height, \(w\) is the image width, and \(c\) is the number of channels of the image.

For 3-D image sequence input, `InputSize` is vector of four elements \([h \ w \ d \ c]\), where \(h\) is the image height, \(w\) is the image width, \(d\) is the image depth, and \(c\) is the number of channels of the image.

Example: 100

**Normalization — Data normalization**

'zerocenter' (default) | 'zscore' | 'rescale-symmetric' | 'rescale-zero-one' | 'none' | function handle

Data normalization to apply every time data is forward propagated through the input layer, specified as one of the following:

- 'zerocenter' — Subtract the mean specified by `Mean`.
- 'zscore' — Subtract the mean specified by `Mean` and divide by `StandardDeviation`.
- 'rescale-symmetric' — Rescale the input to be in the range \([-1, 1]\] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- 'rescale-zero-one' — Rescale the input to be in the range \([0, 1]\] using the minimum and maximum values specified by `Min` and `Max`, respectively.
- 'none' — Do not normalize the input data.
- function handle — Normalize the data using the specified function. The function must be of the form \(Y = \text{func}(X)\), where \(X\) is the input data, and the output \(Y\) is the normalized data.

**Tip** The software, by default, automatically calculates the normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the 'ResetInputNormalization' option in `trainingOptions` to false.

**NormalizationDimension — Normalization dimension**

'auto' (default) | 'channel' | 'element' | 'all'

Normalization dimension, specified as one of the following:
- 'auto' - If the training option is false and you specify any of the normalization statistics (Mean, StandardDeviation, Min, or Max), then normalize over the dimensions matching the statistics. Otherwise, recalculate the statistics at training time and apply channel-wise normalization.
- 'channel' - Channel-wise normalization.
- 'element' - Element-wise normalization.
- 'all' - Normalize all values using scalar statistics.

**Mean — Mean for zero-center and z-score normalization**

[] (default) | numeric array | numeric scalar

Mean for zero-center and z-score normalization, specified as a numeric array, or empty.

- For vector sequence input, **Mean** must be a `InputSize`-by-1 vector of means per channel or a numeric scalar.
- For 2-D image sequence input, **Mean** must be a numeric array of the same size as `InputSize`, a 1-by-1-by-
  `C` array of means per channel, or a numeric scalar, where `C` is the number of channels of the input. For 2-D image sequences, the number of channels is `InputSize(3)`.
- For 3-D image sequence input, **Mean** must be a numeric array of the same size as `InputSize`, a 1-by-1-by-
  1-by-`C` array of means per channel, or a numeric scalar, where `C` is the number of channels of the input. For 3-D image sequences, the number of channels is `InputSize(4)`.

If you specify the **Mean** property, then **Normalization** must be 'zerocenter' or 'zscore'. If Mean is [], then the software calculates the mean at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**StandardDeviation — Standard deviation**

[] (default) | numeric array | numeric scalar

Standard deviation used for z-score normalization, specified as a numeric array, a numeric scalar, or empty.

- For vector sequence input, **StandardDeviation** must be a `InputSize`-by-1 vector of standard deviations per channel or a numeric scalar.
• For 2-D image sequence input, StandardDeviation must be a numeric array of the same size as InputSize, a 1-by-1-by-\(C\) array of standard deviations per channel, or a numeric scalar, where \(C\) is the number of channels of the input. For 2-D image sequences, the number of channels is InputSize(3).

• For 3-D image sequence input, StandardDeviation must be a numeric array of the same size as InputSize, a 1-by-1-by-1-by-\(C\) array of standard deviations per channel, or a numeric scalar, where \(C\) is the number of channels of the input. For 3-D image sequences, the number of channels is InputSize(4).

If you specify the StandardDeviation property, then Normalization must be 'zscore'. If StandardDeviation is [], then the software calculates the standard deviation at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Min — Minimum value for rescaling**

[
] (default) | numeric array | numeric scalar

Minimum value for rescaling, specified as a numeric array, or empty.

• For vector sequence input, Min must be a InputSize-by-1 vector of means per channel or a numeric scalar.

• For 2-D image sequence input, Min must be a numeric array of the same size as InputSize, a 1-by-1-by-\(C\) array of minima per channel, or a numeric scalar, where \(C\) is the number of channels of the input. For 2-D image sequences, the number of channels is InputSize(3).

• For 3-D image sequence input, Min must be a numeric array of the same size as InputSize, a 1-by-1-by-1-by-\(C\) array of minima per channel, or a numeric scalar, where \(C\) is the number of channels of the input. For 3-D image sequences, the number of channels is InputSize(4).

If you specify the Min property, then Normalization must be 'rescale-symmetric' or 'rescale-zero-one'. If Min is [], then the software calculates the minima at training time.

You can set this property when creating networks without training (for example, when assembling networks using assembleNetwork).
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Max — Maximum value for rescaling

[] (default) | numeric array | numeric scalar

Maximum value for rescaling, specified as a numeric array, or empty.

- For vector sequence input, `Max` must be a `InputSize`-by-1 vector of means per channel or a numeric scalar.
- For 2-D image sequence input, `Max` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-`C` array of maxima per channel, or a numeric scalar, where `C` is the number of channels of the input. For 2-D image sequences, the number of channels is `InputSize(3)`.
- For 3-D image sequence input, `Max` must be a numeric array of the same size as `InputSize`, a 1-by-1-by-1-by-`C` array of maxima per channel, or a numeric scalar, where `C` is the number of channels of the input. For 3-D image sequences, the number of channels is `InputSize(4)`.

If you specify the `Max` property, then `Normalization` must be 'rescale-symmetric' or 'rescale-zero-one'. If `Max` is [], then the software calculates the maxima at training time.

You can set this property when creating networks without training (for example, when assembling networks using `assembleNetwork`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layer

Name — Layer name

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: `char` | `string`
**NumInputs — Number of inputs**

0 (default)

Number of inputs of the layer. The layer has no inputs.

Data Types: double

**InputNames — Input names**

{} (default)

Input names of the layer. The layer has no inputs.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

## Examples

### Create Sequence Input Layer

Create a sequence input layer with the name `'seq1'` and an input size of 12.

```matlab
layer = sequenceInputLayer(12,'Name','seq1')
```

layer =
```
SequenceInputLayer with properties:

   Name: 'seq1'
   InputSize: 12
```
Hyperparameters
  Normalization: 'none'
  NormalizationDimension: 'auto'

Include an LSTM layer in a Layer array.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ... 
  sequenceInputLayer(inputSize)
  lstmLayer(numHiddenUnits,'OutputMode','last')
  fullyConnectedLayer(numClasses)
  softmaxLayer
  classificationLayer]
```

Create Sequence Input Layer for Image Sequences

Create a sequence input layer for sequences of 224-224 RGB images with the name 'seq1'.

```
layer = sequenceInputLayer([224 224 3], 'Name', 'seq1')
```

```
layer = SequenceInputLayer with properties:
  Name: 'seq1'
  InputSize: [224 224 3]

Hyperparameters
  Normalization: 'none'
```
Train Network for Sequence Classification

Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. \texttt{XTrain} is a cell array containing 270 sequences of varying length with a feature dimension of 12. \texttt{Y} is a categorical vector of labels 1,2,...,9. The entries in \texttt{XTrain} are matrices with 12 rows (one row for each feature) and a varying number of columns (one column for each time step).

\begin{verbatim}
[XTrain,YTrain] = japaneseVowelsTrainData;
\end{verbatim}

Visualize the first time series in a plot. Each line corresponds to a feature.

\begin{verbatim}
figure
plot(XTrain{1}')
title("Training Observation 1")
numFeatures = size(XTrain{1},1);
legend("Feature " + string(1:numFeatures), 'Location', 'northeastoutside')
\end{verbatim}
Define the LSTM network architecture. Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer to have 100 hidden units and to output the last element of the sequence. Finally, specify nine classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```matlab
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
];
```
softmaxLayer
classificationLayer]
layers =
5x1 Layer array with layers:

1   ''   Sequence Input          Sequence input with 12 dimensions
2   ''   LSTM                    LSTM with 100 hidden units
3   ''   Fully Connected         9 fully connected layer
4   ''   Softmax                 softmax
5   ''   Classification Output   crossentropyex

Specify the training options. Specify the solver as 'adam' and 'GradientThreshold' as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 100.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set 'ExecutionEnvironment' to 'cpu'. To train on a GPU, if available, set 'ExecutionEnvironment' to 'auto' (the default value).

maxEpochs = 100;
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',1, ...
    'Verbose',false, ...
    'Plots','training-progress');

Train the LSTM network with the specified training options.

net = trainNetwork(XTrain,YTrain,layers,options);
Load the test set and classify the sequences into speakers.

[XTest,YTest] = japaneseVowelsTestData;

Classify the test data. Specify the same mini-batch size used for training.

YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);

Calculate the classification accuracy of the predictions.
acc = sum(YPred == YTest)./numel(YTest)

acc = 0.9486

**Classification LSTM Networks**

To create an LSTM network for sequence-to-label classification, create a layer array containing a sequence input layer, an LSTM layer, a fully connected layer, a softmax layer, and a classification output layer.

Set the size of the sequence input layer to the number of features of the input data. Set the size of the fully connected layer to the number of classes. You do not need to specify the sequence length.

For the LSTM layer, specify the number of hidden units and the output mode 'last'.

```matlab
numFeatures = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For an example showing how to train an LSTM network for sequence-to-label classification and classify new data, see “Sequence Classification Using Deep Learning”.

To create an LSTM network for sequence-to-sequence classification, use the same architecture as for sequence-to-label classification, but set the output mode of the LSTM layer to 'sequence'.

```matlab
numFeatures = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```
Regression LSTM Networks

To create an LSTM network for sequence-to-one regression, create a layer array containing a sequence input layer, an LSTM layer, a fully connected layer, and a regression output layer.

Set the size of the sequence input layer to the number of features of the input data. Set the size of the fully connected layer to the number of responses. You do not need to specify the sequence length.

For the LSTM layer, specify the number of hidden units and the output mode 'last'.

```matlab
numFeatures = 12;
numHiddenUnits = 125;
numResponses = 1;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

To create an LSTM network for sequence-to-sequence regression, use the same architecture as for sequence-to-one regression, but set the output mode of the LSTM layer to 'sequence'.

```matlab
numFeatures = 12;
numHiddenUnits = 125;
numResponses = 1;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

For an example showing how to train an LSTM network for sequence-to-sequence regression and predict on new data, see “Sequence-to-Sequence Regression Using Deep Learning”.

sequenceInputLayer
**Deeper LSTM Networks**

You can make LSTM networks deeper by inserting extra LSTM layers with the output mode 'sequence' before the LSTM layer. To prevent overfitting, you can insert dropout layers after the LSTM layers.

For sequence-to-label classification networks, the output mode of the last LSTM layer must be 'last'.

```matlab
numFeatures = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits2,'OutputMode','last')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For sequence-to-sequence classification networks, the output mode of the last LSTM layer must be 'sequence'.

```matlab
numFeatures = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```
Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

• To input sequences of images into a network, use a sequence input layer.
• To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
• To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
• To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

• A sequence input layer with an input size of [28 28 1].
• A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
• An LSTM layer with 200 hidden units that outputs the last time step only.
• A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

```matlab
inputSize = [28 28 1];
filterSize = 5;
numFilters = 20;
numHiddenUnits = 200;
numClasses = 10;

layers = [ ...
    sequenceInputLayer(inputSize,'Name','input')
];```
sequenceFoldingLayer('Name','fold')

convolution2dLayer(filterSize,numFilters,'Name','conv')
batchNormalizationLayer('Name','bn')
reluLayer('Name','relu')

sequenceUnfoldingLayer('Name','unfold')
flattenLayer('Name','flatten')
lstmLayer(numHiddenUnits,'OutputMode','last','Name','lstm')

fullyConnectedLayer(numClasses, 'Name','fc')
softmaxLayer('Name','softmax')
classificationLayer('Name','classification')];

Convert the layers to a layer graph and connect the miniBatchSize output of the sequence folding layer to the corresponding input of the sequence unfolding layer.

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'fold/miniBatchSize','unfold/miniBatchSize');

View the final network architecture using the plot function.

figure
plot(lgraph)
Compatibility Considerations

sequenceInputLayer, by default, uses channel-wise normalization for zero-center normalization

Behavior change in future release

Starting in R2019b, sequenceInputLayer, by default, uses channel-wise normalization for zero-center normalization. In previous versions, this layer uses element-wise normalization. To reproduce this behavior, set the NormalizationDimension option of this layer to 'element'.
References


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
bilstmLayer | classifyAndUpdateState | flattenLayer | lstmLayer | predictAndUpdateState | resetState | sequenceFoldingLayer | sequenceUnfoldingLayer

Topics
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Classify Videos Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2017b
sequenceFoldingLayer

Sequence folding layer

Description

A sequence folding layer converts a batch of image sequences to a batch of images. Use a sequence folding layer to perform convolution operations on time steps of image sequences independently.

To use a sequence folding layer, you must connect the miniBatchSize output to the miniBatchSize input of the corresponding sequence unfolding layer. For an example, see “Create Network for Video Classification” on page 1-1121.

Creation

Syntax

layer = sequenceFoldingLayer
layer = sequenceFoldingLayer('Name',Name)

Description

layer = sequenceFoldingLayer creates a sequence folding layer.

layer = sequenceFoldingLayer('Name',Name) creates a sequence folding layer and sets the optional Name property using a name-value pair. For example, sequenceFoldingLayer('Name','fold1') creates a sequence folding layer with the name 'fold1'. Enclose the property name in single quotes.

Properties

Name — Layer name

' ' (default) | character vector | string scalar
Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{"in"} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

2 (default)

Number of outputs of the layer.

The layer has two outputs:

- 'out' – Output feature map corresponding to reshaped input.
- 'miniBatchSize' – Size of the mini-batch passed into the layer. This output must be connected to the 'miniBatchSize' input of the corresponding sequence unfolding layer.

Data Types: double

**OutputNames — Output names**

{"out','miniBatchSize'} (default)

Output names of the layer.

The layer has two outputs:

- 'out' – Output feature map corresponding to reshaped input.
• `'miniBatchSize'` – Size of the mini-batch passed into the layer. This output must be connected to the `'miniBatchSize'` input of the corresponding sequence unfolding layer.

Data Types: `cell`

### Examples

**Create Sequence Folding Layer**

Create a sequence folding layer with name the `'fold1'`.

```matlab
layer = sequenceFoldingLayer('Name','fold1')
```

```matlab
layer = 
    SequenceFoldingLayer with properties:
    
    Name: 'fold1'
    NumOutputs: 2
    OutputNames: {'out'  'miniBatchSize'}
```

**Create Network for Video Classification**

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.
Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

- A sequence input layer with an input size of [28 28 1].
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

```matlab
inputSize = [28 28 1];
filterSize = 5;
umFilters = 20;
numHiddenUnits = 200;
numClasses = 10;
layers = [ ...
    sequenceInputLayer(inputSize,'Name','input')
    sequenceFoldingLayer('Name','fold')
    convolution2dLayer(filterSize,numFilters,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    sequenceUnfoldingLayer('Name','unfold')
    flattenLayer('Name','flatten')
    lstmLayer(numHiddenUnits,'OutputMode','last','Name','lstm')
    fullyConnectedLayer(numClasses, 'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')]
```
Convert the layers to a layer graph and connect the `miniBatchSize` output of the sequence folding layer to the corresponding input of the sequence unfolding layer.

```matlab
lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'fold/miniBatchSize','unfold/miniBatchSize');
```

View the final network architecture using the `plot` function.

```matlab
figure
plot(lgraph)
```
See Also
bilstmLayer | classifyAndUpdateState | flattenLayer | lstmLayer |
predictAndUpdateState | resetState | sequenceInputLayer |
sequenceUnfoldingLayer

Topics
“Classify Videos Using Deep Learning”
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
"Long Short-Term Memory Networks"
"Specify Layers of Convolutional Neural Network"
"Set Up Parameters and Train Convolutional Neural Network"
"Deep Learning in MATLAB"
"List of Deep Learning Layers"

**Introduced in R2019a**
sequenceUnfoldingLayer

Sequence unfolding layer

Description

A sequence unfolding layer restores the sequence structure of the input data after sequence folding.

To use a sequence unfolding layer, you must connect the miniBatchSize output of the corresponding sequence folding layer to the miniBatchSize input of the sequence unfolding layer. For an example, see “Create Network for Video Classification” on page 1-1128.

Creation

Syntax

layer = sequenceUnfoldingLayer
layer = sequenceUnfoldingLayer('Name',Name)

Description

layer = sequenceUnfoldingLayer creates a sequence unfolding layer.

layer = sequenceUnfoldingLayer('Name',Name) creates a sequence unfolding layer and sets the optional Name property using a name-value pair. For example, sequenceUnfoldingLayer('Name','unfold1') creates a sequence unfolding layer with the name 'unfold1'. Enclose the property name in single quotes.

Properties

Name — Layer name

' ' (default) | character vector | string scalar
Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**  
2 (default)

Number of inputs of the layer.

This layer has two inputs:

- 'in' — Input feature map.
- 'miniBatchSize' — Size of the mini-batch from the corresponding sequence folding layer. This output must be connected to the 'miniBatchSize' output of the corresponding sequence folding layer.

Data Types: double

**InputNames — Input names**  
{'in','miniBatchSize'} (default)

Input names of the layer.

This layer has two inputs:

- 'in' — Input feature map.
- 'miniBatchSize' — Size of the mini-batch from the corresponding sequence folding layer. This output must be connected to the 'miniBatchSize' output of the corresponding sequence folding layer.

Data Types: cell

**NumOutputs — Number of outputs**  
1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**  
{'out'} (default)
Output names of the layer. This layer has a single output only.

Data Types: cell

Examples

Create Sequence Unfolding Layer

Create a sequence unfolding layer with the name 'unfold1'.

```matlab
layer = sequenceUnfoldingLayer('Name','unfold1')
```

```
layer = 
    SequenceUnfoldingLayer with properties:
        Name: 'unfold1'
        NumInputs: 2
        InputNames: {'in'  'miniBatchSize'}
```

Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.
Define the following network architecture:

- A sequence input layer with an input size of [28 28 1].
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.

To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

```matlab
inputSize = [28 28 1];
filterSize = 5;
umFilters = 20;
umHiddenUnits = 200;
umClasses = 10;
layers = [
    sequenceInputLayer(inputSize,'Name','input')
    sequenceFoldingLayer('Name','fold')
    convolution2dLayer(filterSize,numFilters,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')
    sequenceUnfoldingLayer('Name','unfold')
    flattenLayer('Name','flatten')
    lstmLayer(numHiddenUnits,'OutputMode','last','Name','lstm')
    fullyConnectedLayer(numClasses, 'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')
];

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'fold/miniBatchSize','unfold/miniBatchSize');
```

Convert the layers to a layer graph and connect the miniBatchSize output of the sequence folding layer to the corresponding input of the sequence unfolding layer.
View the final network architecture using the `plot` function.

```matlab
figure
plot(lgraph)
```

See Also

- `bilstmLayer`
- `classifyAndUpdateState`
- `flattenLayer`
- `lstmLayer`
- `predictAndUpdateState`
- `resetState`
- `sequenceFoldingLayer`
- `sequenceInputLayer`
Topics

“Classify Videos Using Deep Learning”
“Classify Videos Using Deep Learning”
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Long Short-Term Memory Networks”
“Visualize Activations of LSTM Network”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

Introduced in R2019a
flattenLayer

Flatten layer

Description

A flatten layer collapses the spatial dimensions of the input into the channel dimension.

For example, if the input to the layer is an $H$-by-$W$-by-$C$-by-$N$-by-$S$ array (sequences of images), then the flattened output is an $(H \times W \times C)$-by-$N$-by-$S$ array.

This layer supports sequence input only.

Creation

Syntax

```matlab
layer = flattenLayer
layer = flattenLayer('Name',Name)
```

Description

`layer = flattenLayer` creates a flatten layer.

`layer = flattenLayer('Name',Name)` sets the optional `Name` property using a name-value pair. For example, `flattenLayer('Name','flatten1')` creates a flatten layer with name 'flatten1'.

Properties

Name — Layer name

' ' (default) | character vector | string scalar
Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and Name is set to '', then the software automatically assigns a name to the layer at training time.

Data Types: char | string

**NumInputs — Number of inputs**

1 (default)

Number of inputs of the layer. This layer accepts a single input only.

Data Types: double

**InputNames — Input names**

{'in'} (default)

Input names of the layer. This layer accepts a single input only.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

### Object Functions

### Examples
Create Flatten Layer

Create a flatten layer with the name 'flatten1'.

```
layer = flattenLayer('Name','flatten1')
```

Create Network for Video Classification

Create a deep learning network for data containing sequences of images, such as video and medical image data.

- To input sequences of images into a network, use a sequence input layer.
- To apply convolutional operations independently to each time step, first convert the sequences of images to an array of images using a sequence folding layer.
- To restore the sequence structure after performing these operations, convert this array of images back to image sequences using a sequence unfolding layer.
- To convert images to feature vectors, use a flatten layer.

You can then input vector sequences into LSTM and BiLSTM layers.

Define Network Architecture

Create a classification LSTM network that classifies sequences of 28-by-28 grayscale images into 10 classes.

Define the following network architecture:

- A sequence input layer with an input size of [28 28 1].
- A convolution, batch normalization, and ReLU layer block with 20 5-by-5 filters.
- An LSTM layer with 200 hidden units that outputs the last time step only.
- A fully connected layer of size 10 (the number of classes) followed by a softmax layer and a classification layer.
To perform the convolutional operations on each time step independently, include a sequence folding layer before the convolutional layers. LSTM layers expect vector sequence input. To restore the sequence structure and reshape the output of the convolutional layers to sequences of feature vectors, insert a sequence unfolding layer and a flatten layer between the convolutional layers and the LSTM layer.

inputSize = [28 28 1];
filterSize = 5;
numFilters = 20;
numHiddenUnits = 200;
numClasses = 10;
layers = [ ...
    sequenceInputLayer(inputSize,'Name','input')

    sequenceFoldingLayer('Name','fold')

    convolution2dLayer(filterSize,numFilters,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')

    sequenceUnfoldingLayer('Name','unfold')
    flattenLayer('Name','flatten')

    lstmLayer(numHiddenUnits,'OutputMode','last','Name','lstm')

    fullyConnectedLayer(numClasses, 'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')
];

Convert the layers to a layer graph and connect the miniBatchSize output of the sequence folding layer to the corresponding input of the sequence unfolding layer.

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'fold/miniBatchSize','unfold/miniBatchSize');

View the final network architecture using the plot function.

figure
plot(lgraph)
See Also
bilstmLayer | classifyAndUpdateState | lstmLayer | predictAndUpdateState | resetState | sequenceFoldingLayer | sequenceInputLayer | sequenceUnfoldingLayer

Topics
“Classify Videos Using Deep Learning”
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Deep Learning in MATLAB”
“List of Deep Learning Layers”

**Introduced in R2019a**
Layer
Network layer for deep learning

Description
Layers that define the architecture of neural networks for deep learning.

Creation
For a list of deep learning layers in MATLAB, see “List of Deep Learning Layers”. To specify the architecture of a neural network with all layers connected sequentially, create an array of layers directly. To specify the architecture of a network where layers can have multiple inputs or outputs, use a LayerGraph object.

Alternatively, you can import layers from Caffe, Keras, and ONNX using importCaffeLayers, importKerasLayers, and importONNXLayers respectively.

To learn how to create your own custom layers, see “Define Custom Deep Learning Layers”.

Object Functions
trainNetwork  Train neural network for deep learning

Examples

Construct Network Architecture
Define a convolutional neural network architecture for classification with one convolutional layer, a ReLU layer, and a fully connected layer.

layers = [ ...  
    imageInputLayer([28 28 3])
convolution2dLayer([5 5],10)
reluLayer
fullyConnectedLayer(10)
softmaxLayer
classificationLayer]

layers =
6x1 Layer array with layers:

1 ' ' Image Input 28x28x3 images with 'zerocenter' normalization
2 ' ' Convolution 10 5x5 convolutions with stride [1 1] and padding
3 ' ' ReLU ReLU
4 ' ' Fully Connected 10 fully connected layer
5 ' ' Softmax softmax
6 ' ' Classification Output crossentropyex

layers is a Layer object.

Alternatively, you can create the layers individually and then concatenate them.

input = imageInputLayer([28 28 3]);
conv = convolution2dLayer([5 5],10);
relu = reluLayer;
fc = fullyConnectedLayer(10);
sm = softmaxLayer;
co = classificationLayer;

layers = [ ...
input
conv
relu
fc
sm
co]

layers =
6x1 Layer array with layers:

1 ' ' Image Input 28x28x3 images with 'zerocenter' normalization
2 ' ' Convolution 10 5x5 convolutions with stride [1 1] and padding
3 ' ' ReLU ReLU
4 ' ' Fully Connected 10 fully connected layer
5 ' ' Softmax softmax
6 ' ' Classification Output crossentropyex
**Access Layers and Properties in Layer Array**

Define a convolutional neural network architecture for classification with one convolutional layer, a ReLU layer, and a fully connected layer.

```matlab
layers = [...
    imageInputLayer([28 28 3])
    convolution2dLayer([5 5],10)
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Display the image input layer by selecting the first layer.

```matlab
layers(1)
```

```matlab
ans = 
    ImageInputLayer with properties:

    Name: ''
    InputSize: [28 28 3]

    Hyperparameters
    DataAugmentation: 'none'
    Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
    Mean: []
```

View the input size of the image input layer.

```matlab
layers(1).InputSize
```

```matlab
ans = 1x3
    28   28    3
```

Display the stride for the convolutional layer.

```matlab
layers(2).Stride
```
access the bias learn rate factor for the fully connected layer.

layers(4).BiasLearnRateFactor

ans = 1

Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A shortcut connection containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. All layers must have names and all names must be unique.

layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(5,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')

    convolution2dLayer(3,32,'Padding','same','Stride',2,'Name','conv_2')
    batchNormalizationLayer('Name','BN_2')
    reluLayer('Name','relu_2')

    convolution2dLayer(3,32,'Padding','same','Name','conv_3')
    batchNormalizationLayer('Name','BN_3')
    reluLayer('Name','relu_3')

    additionLayer(2,'Name','add')

    averagePooling2dLayer(2,'Stride',2,'Name','avpool')
    fullyConnectedLayer(10,'Name','fc')
]
softmaxLayer('Name','softmax')
classificationLayer('Name','classOutput')];

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```

Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This arrangement enables the addition layer to add the outputs of
the 'skipConv' and 'relu_3' layers. To check that the layer is in the graph, plot the layer graph.

```matlab
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```

Create the shortcut connection from the 'relu_1' layer to the 'add' layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named 'in1' and 'in2'. The 'relu_3' layer is already connected to the 'in1' input. Connect the 'relu_1' layer to the 'skipConv' layer and the 'skipConv' layer to the 'in2' input of the 'add' layer. The addition layer now sums the outputs of
the 'relu_3' and 'skipConv' layers. To check that the layers are connected correctly, plot the layer graph.

```matlab
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```

Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;
```
Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```matlab
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...  
    'Shuffle','every-epoch', ...  
    'ValidationData',{XValidation,YValidation}, ...  
    'ValidationFrequency',30, ...  
    'Verbose',false, ...  
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```

Display the properties of the trained network. The network is a DAGNetwork object.

```matlab
net
```

```matlab
net = 
    DAGNetwork with properties:
```
Layers: [16×1 nnet.cnn.layer.Layer]
Connections: [16×2 table]

Classify the validation images and calculate the accuracy. The network is very accurate.

YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)

accuracy = 0.9968

See Also
Layer | LayerGraph | assembleNetwork | importCaffeLayers | importKerasLayers | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”

Introduced in R2016a
**SeriesNetwork**

Series network for deep learning

**Description**

A series network is a neural network for deep learning with layers arranged one after the other. It has a single input layer and a single output layer.

**Creation**

There are several ways to create a `SeriesNetwork` object:

- Load a pretrained network using `alexnet`, `vgg16`, or `vgg19`. For an example, see “Load Pretrained AlexNet Convolutional Neural Network” on page 1-1148.
- Import a pretrained network from Keras using `importKerasNetwork`. For an example, see “Import and Plot Keras Network” on page 1-1384.
- Import a pretrained network from Caffe using `importCaffeNetwork`. For an example, see “Import Caffe Network” on page 1-788.
- Train or fine-tune a network using `trainNetwork`. For an example, see “Train Network for Image Classification” on page 1-1150.

**Note** To learn about other pretrained networks, such as `googlenet` and `resnet50`, see “Pretrained Deep Neural Networks”.

**Properties**

- **Layers — Network layers**
  
  *Layer array*
  
  Network layers, specified as a `Layer` array.

- **InputNames — Network input layer names**
  
  *cell array*
  
  Network input layer names.
Network input layer names, specified as a cell array of character vectors.
Data Types: cell

**OutputNames — Network output layer names**
cell array

Network output layer names, specified as a cell array of character vectors.
Data Types: cell

## Object Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activations</td>
<td>Compute deep learning network layer activations</td>
</tr>
<tr>
<td>classify</td>
<td>Classify data using a trained deep learning neural network</td>
</tr>
<tr>
<td>predict</td>
<td>Predict responses using a trained deep learning neural network</td>
</tr>
<tr>
<td>predictAndUpdateState</td>
<td>Predict responses using a trained recurrent neural network and update the network state</td>
</tr>
<tr>
<td>classifyAndUpdateState</td>
<td>Classify data using a trained recurrent neural network and update the network state</td>
</tr>
<tr>
<td>resetState</td>
<td>Reset the state of a recurrent neural network</td>
</tr>
</tbody>
</table>

## Examples

### Load Pretrained AlexNet Convolutional Neural Network

Load a pretrained AlexNet convolutional neural network and examine the layers and classes.

Load the pretrained AlexNet network using `alexnet`. The output `net` is a `SeriesNetwork` object.

```matlab
net = alexnet
```

```matlab
net = 
SeriesNetwork with properties: 
    Layers: [25×1 nnet.cnn.layer.Layer]
```
Using the Layers property, view the network architecture. The network comprises of 25 layers. There are 8 layers with learnable weights: 5 convolutional layers, and 3 fully connected layers.

\[\text{net.Layers}\]

\[
\begin{align*}
\text{ans} &= \text{25x1 Layer array with layers:} \\
1 &\text{'data'} \quad \text{Image Input} \\
2 &\text{'conv1'} \quad \text{Convolution} \\
3 &\text{'relu1'} \quad \text{ReLU} \\
4 &\text{'norm1'} \quad \text{Cross Channel Normalization} \\
5 &\text{'pool1'} \quad \text{Max Pooling} \\
6 &\text{'conv2'} \quad \text{Grouped Convolution} \\
7 &\text{'relu2'} \quad \text{ReLU} \\
8 &\text{'norm2'} \quad \text{Cross Channel Normalization} \\
9 &\text{'pool2'} \quad \text{Max Pooling} \\
10 &\text{'conv3'} \quad \text{Convolution} \\
11 &\text{'relu3'} \quad \text{ReLU} \\
12 &\text{'conv4'} \quad \text{Grouped Convolution} \\
13 &\text{'relu4'} \quad \text{ReLU} \\
14 &\text{'conv5'} \quad \text{Grouped Convolution} \\
15 &\text{'relu5'} \quad \text{ReLU} \\
16 &\text{'pool5'} \quad \text{Max Pooling} \\
17 &\text{'fc6'} \quad \text{Fully Connected} \\
18 &\text{'relu6'} \quad \text{ReLU} \\
19 &\text{'drop6'} \quad \text{Dropout} \\
20 &\text{'fc7'} \quad \text{Fully Connected} \\
21 &\text{'relu7'} \quad \text{ReLU} \\
22 &\text{'drop7'} \quad \text{Dropout} \\
23 &\text{'fc8'} \quad \text{Fully Connected} \\
24 &\text{'prob'} \quad \text{Softmax} \\
25 &\text{'output'} \quad \text{Classification Output}
\end{align*}
\]

You can view the names of the classes learned by the network by viewing the Classes property of the classification output layer (the final layer). View the first 10 classes by selecting the first 10 elements.

\[\text{net.Layers(end).Classes(1:10)}\]

\[
\begin{align*}
\text{ans} &= 10\times1 \text{ categorical array} \\
\text{tench} \\
\text{goldfish} \\
\text{great white shark}
\end{align*}
\]
tiger shark
hammerhead
electric ray
stingray
cock
hen
ostrich

**Import Layers from Caffe Network**

Specify the example file 'digitsnet.prototxt' to import.

```matlab
protofile = 'digitsnet.prototxt';
```

Import the network layers.

```matlab
layers = importCaffeLayers(protofile);
```

1x7 Layer array with layers:

1  'testdata'  Image Input  28x28x1 images
2  'conv1'    Convolution  20 5x5x1 convolutions with stride [1 1]
3  'relu1'    ReLU        ReLU
4  'pool1'    Max Pooling  2x2 max pooling with stride [2 2] and padding [0 0]
5  'ipl1'     Fully Connected 10 fully connected layer
6  'loss'     Softmax     softmax
7  'output'   Classification Output crossentropyex with 'class1', 'class2', and
```

**Train Network for Image Classification**

Load the data as an ImageDatastore object.

```matlab
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```
The datastore contains 10,000 synthetic images of digits from 0 to 9. The images are generated by applying random transformations to digit images created with different fonts. Each digit image is 28-by-28 pixels. The datastore contains an equal number of images per category.

Display some of the images in the datastore.

```matlab
figure
tnumImages = 10000;
perm = randperm(numImages,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end
```

![Image of digit images](image-url)
Divide the datastore so that each category in the training set has 750 images and the testing set has the remaining images from each label.

\[
\text{numTrainingFiles} = 750;
\]

\[
[\text{imdsTrain}, \text{imdsTest}] = \text{splitEachLabel}(\text{imds}, \text{numTrainingFiles}, '\text{randomize}');
\]

\text{splitEachLabel} splits the image files in digitData into two new datastores, imdsTrain and imdsTest.

Define the convolutional neural network architecture.

\[
\text{layers} = [...
    \text{imageInputLayer([28 28 1])}
    \text{convolution2dLayer(5,20)}
    \text{reluLayer}
    \text{maxPooling2dLayer(2,'\text{Stride}',2)}
    \text{fullyConnectedLayer(10)}
    \text{softmaxLayer}
    \text{classificationLayer}];
\]

Set the options to the default settings for the stochastic gradient descent with momentum. Set the maximum number of epochs at 20, and start the training with an initial learning rate of 0.0001.

\[
\text{options} = \text{trainingOptions('sgdm', ...}
    \text{   'MaxEpochs',20,...}
    \text{   'InitialLearnRate',1e-4,...}
    \text{   'Verbose',false,...}
    \text{   'Plots','training-progress'});
\]

Train the network.

\[
\text{net} = \text{trainNetwork}(\text{imdsTrain}, \text{layers}, \text{options});
\]
Run the trained network on the test set, which was not used to train the network, and predict the image labels (digits).

```matlab
YPred = classify(net,imdsTest);
YTest = imdsTest.Labels;
```

Calculate the accuracy. The accuracy is the ratio of the number of true labels in the test data matching the classifications from `classify` to the number of images in the test data.
accuracy = sum(YPred == YTest)/numel(YTest)
accuracy = 0.9412

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the activations and predict object functions are supported.
- To create a SeriesNetwork object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the activations and predict object functions are supported.
- To create a SeriesNetwork object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also
DAGNetwork | alexnet | analyzeNetwork | assembleNetwork | importCaffeNetwork | trainNetwork | trainingOptions | vgg16 | vgg19

Topics
“Create Simple Deep Learning Network for Classification”
“Train Convolutional Neural Network for Regression”
“Sequence Classification Using Deep Learning”
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”
“Long Short-Term Memory Networks”

**Introduced in R2016a**
TrainingOptionsSGDM

Training options for stochastic gradient descent with momentum

Description

Training options for stochastic gradient descent with momentum, including learning rate information, L_2 regularization factor, and mini-batch size.

Creation

Create a TrainingOptionsSGDM object using trainingOptions and specifying 'sgdm' as the solverName input argument.

Properties

Plots and Display

Plots — Plots to display during network training

'none' | 'training-progress'

Plots to display during network training, specified as one of the following:

- 'none' — Do not display plots during training.
- 'training-progress' — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.

Verbose — Indicator to display training progress information

1 | 0

Indicator to display training progress information in the command window, specified as 1 (true) or 0 (false).
The displayed information includes the epoch number, iteration number, time elapsed, mini-batch loss, mini-batch accuracy, and base learning rate. When you train a regression network, root mean square error (RMSE) is shown instead of accuracy. If you validate the network during training, then the displayed information also includes the validation loss and validation accuracy (or RMSE).

Data Types: logical

**VerboseFrequency — Frequency of verbose printing**

*positive integer*

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as a positive integer. This property only has an effect when the *Verbose* value equals *true*.

If you validate the network during training, then *trainNetwork* prints to the command window every time validation occurs.

**Mini-Batch Options**

**MaxEpochs — Maximum number of epochs**

*positive integer*

Maximum number of epochs to use for training, specified as a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

**MiniBatchSize — Size of mini-batch**

*positive integer*

Size of the mini-batch to use for each training iteration, specified as a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights.

**Shuffle — Option for data shuffling**

*once* | *never* | *every-epoch*

Option for data shuffling, specified as one of the following:

* 'once' — Shuffle the training and validation data once before training.*
• 'never' — Do not shuffle the data.
• 'every-epoch' — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then trainNetwork discards the training data that does not fit into the final complete mini-batch of each epoch. Set the Shuffle value to 'every-epoch' to avoid discarding the same data every epoch.

**Validation**

**ValidationData — Data to use for validation during training**
image datastore | datastore | table | cell array

Data to use for validation during training, specified as an image datastore, a datastore that returns data in a two-column table or two-column cell array, a table, or a cell array. The format of the validation data depends on the type of task and correspond to valid inputs to the trainNetwork function.

This option supports networks with a single input only.

**Image Data**

For image data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image datastore</td>
<td>ImageDatastore object with categorical labels.</td>
<td>imds argument of trainNetwork</td>
</tr>
<tr>
<td>Datastore</td>
<td>Datastore that returns data in a two-column table or two-column cell array, where the two columns specify the network inputs and expected responses, respectively.</td>
<td>ds argument of trainNetwork</td>
</tr>
</tbody>
</table>
### Sequence and Time Series Data

For sequence and time series data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array {sequences, Y}</td>
<td>sequences</td>
<td>sequences argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>Cell array of sequences or time series data.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>Numeric array of images.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>Categorical vector of labels, cell array of categorical sequences, matrix of numeric responses, or cell array of numeric sequences.</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>Table containing absolute or relative file paths to a MAT files containing sequence or time series data.</td>
<td>tbl argument of trainNetwork</td>
</tr>
</tbody>
</table>
During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the 'ValidationFrequency' name-value pair argument. You can also use the validation data to stop training automatically when the validation loss stops decreasing. To turn on automatic validation stopping, use the 'ValidationPatience' name-value pair argument.

If your network has layers that behave differently during prediction than during training (for example, dropout layers), then the validation accuracy can be higher than the training (mini-batch) accuracy.

The validation data is shuffled according to the 'Shuffle' value. If the 'Shuffle' value equals 'every-epoch', then the validation data is shuffled before each network validation.

**ValidationFrequency — Frequency of network validation**

positive integer

Frequency of network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics.

**ValidationPatience — Patience of validation stopping**

positive integer | Inf

Patience of validation stopping of network training, specified as a positive integer or Inf.

The 'ValidationPatience' value is the number of times that the loss on the validation set can be larger than or equal to the previously smallest loss before network training stops.

**Solver Options**

**InitialLearnRate — Initial learning rate**

positive scalar

Initial learning rate used for training, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result.
LearnRateScheduleSettings — Settings for learning rate schedule
structure

Settings for the learning rate schedule, specified as a structure. LearnRateScheduleSettings has the field Method, which specifies the type of method for adjusting the learning rate. The possible methods are:

• 'none' — The learning rate is constant throughout training.
• 'piecewise' — The learning rate drops periodically during training.

If Method is 'piecewise', then LearnRateScheduleSettings contains two more fields:

• DropRateFactor — The multiplicative factor by which the learning rate drops during training
• DropPeriod — The number of epochs that passes between adjustments to the learning rate during training

Specify the settings for the learning schedule rate using trainingOptions.

Data Types: struct

L2Regularization — Factor for L2 regularizer
nonnegative scalar

Factor for L2 regularizer (weight decay), specified as a nonnegative scalar.

You can specify a multiplier for the L2 regularizer for network layers with learnable parameters.

Momentum — Contribution of previous gradient step
scalar from 0 to 1

Contribution of the gradient step from the previous iteration to the current iteration of the training, specified as a scalar value from 0 to 1. A value of 0 means no contribution from the previous step, whereas a value of 1 means maximal contribution from the previous step. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.
Gradient Clipping

**GradientThreshold — Gradient threshold**
positive scalar | Inf

Positive threshold for the gradient, specified as positive scalar or Inf. When the gradient exceeds the value of GradientThreshold, then the gradient is clipped according to GradientThresholdMethod.

**GradientThresholdMethod — Gradient threshold method**
'l2norm' | 'global-l2norm' | 'absolute-value'

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following:

- 'l2norm' — If the L2 norm of the gradient of a learnable parameter is larger than GradientThreshold, then scale the gradient so that the L2 norm equals GradientThreshold.
- 'global-l2norm' — If the global L2 norm, \( L \), is larger than GradientThreshold, then scale all gradients by a factor of \( \text{GradientThreshold}/L \). The global L2 norm considers all learnable parameters.
- 'absolute-value' — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than GradientThreshold, then scale the partial derivative to have magnitude equal to GradientThreshold and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-694.

**ResetInputNormalization — Option to reset input layer normalization**
true (default) | false

Option to reset input layer normalization, specified as one of the following:

- true - Reset the input layer normalization statistics and recalculate them at training time.
- false - Calculate normalization statistics at training time when they are empty.

**Sequence Options**

**SequenceLength — Option to pad or truncate sequences**
'longest' | 'shortest' | positive integer
Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.

- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.

- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingDirection** — Direction of padding or truncation

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.

- 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer **OutputMode** property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the **OutputMode** property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the 'SequencePaddingDirection' option to 'right'.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

1-1163
**SequencePaddingValue — Value to pad sequences**

scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

**Hardware Options**

**ExecutionEnvironment — Hardware resource for training network**

'auto' | 'cpu' | 'gpu' | 'multi-gpu' | 'parallel'

Hardware resource for training network, specified as one of the following:

- 'auto' — Use a GPU if one is available. Otherwise, use the CPU.
- 'cpu' — Use the CPU.
- 'gpu' — Use the GPU.
- 'multi-gpu' — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- 'parallel' — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel and in the Cloud”.

GPU, multi-GPU, and parallel options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

To see an improvement in performance when training in parallel, try scaling up the `MiniBatchSize` and `InitialLearnRate` training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.
Specify the execution environment using `trainingOptions`.

Data Types: `char` | `string`

**WorkerLoad — Parallel worker load division**

scalar from 0 to 1 | positive integer | numeric vector

Worker load division for GPUs or CPUs, specified as a scalar from 0 to 1, a positive integer, or a numeric vector. This property has an effect only when the `ExecutionEnvironment` value equals 'multi-gpu' or 'parallel'.

**Checkpoints**

**CheckpointPath — Path for saving checkpoint networks**

character vector

Path where checkpoint networks are saved, specified as a character vector.

Data Types: `char`

**OutputFcn — Output functions**

function handle | cell array of function handles

Output functions to call during training, specified as a function handle or cell array of function handles. `trainNetwork` calls the specified functions once before the start of training, after each iteration, and once after training has finished. `trainNetwork` passes a structure containing information in the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch</td>
<td>Current epoch number</td>
</tr>
<tr>
<td>Iteration</td>
<td>Current iteration number</td>
</tr>
<tr>
<td>TimeSinceStart</td>
<td>Time in seconds since the start of training</td>
</tr>
<tr>
<td>TrainingLoss</td>
<td>Current mini-batch loss</td>
</tr>
<tr>
<td>ValidationLoss</td>
<td>Loss on the validation data</td>
</tr>
<tr>
<td>BaseLearnRate</td>
<td>Current base learning rate</td>
</tr>
<tr>
<td>TrainingAccuracy</td>
<td>Accuracy on the current mini-batch (classification networks)</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrainingRMSE</td>
<td>RMSE on the current mini-batch (regression networks)</td>
</tr>
<tr>
<td>ValidationAccuracy</td>
<td>Accuracy on the validation data (classification networks)</td>
</tr>
<tr>
<td>ValidationRMSE</td>
<td>RMSE on the validation data (regression networks)</td>
</tr>
<tr>
<td>State</td>
<td>Current training state, with a possible value of &quot;start&quot;, &quot;iteration&quot;, or &quot;done&quot;.</td>
</tr>
</tbody>
</table>

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

You can use output functions to display or plot progress information, or to stop training. To stop training early, make your output function return `true`. If any output function returns `true`, then training finishes and `trainNetwork` returns the latest network. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training”.

Data Types: `function_handle` | `cell`

### Examples

#### Specify Training Options

Create a set of options for training a network using stochastic gradient descent with momentum. Reduce the learning rate by a factor of 0.2 every 5 epochs. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Turn on the training progress plot.

```matlab
options = trainingOptions('sgdm', ...
   'LearnRateSchedule','piecewise', ...)
   'LearnRateDropFactor',0.2, ...
   'LearnRateDropPeriod',5, ...
   'MaxEpochs',20, ...
   'MiniBatchSize',64, ...
   'Plots','training-progress')
```
options = TrainingOptionsSGDM with properties:

    Momentum: 0.9000
    InitialLearnRate: 0.0100
    LearnRateScheduleSettings: [1x1 struct]
    L2Regularization: 1.0000e-04
    GradientThresholdMethod: 'l2norm'
    GradientThreshold: Inf
    MaxEpochs: 20
    MiniBatchSize: 64
    Verbose: 1
    VerboseFrequency: 50
    ValidationData: []
    ValidationFrequency: 50
    ValidationPatience: Inf
    Shuffle: 'once'
    CheckpointPath: ''
    ExecutionEnvironment: 'auto'
    WorkerLoad: []
    OutputFcn: []
    Plots: 'training-progress'
    SequenceLength: 'longest'
    SequencePaddingValue: 0
    SequencePaddingDirection: 'right'
    DispatchInBackground: 0
    ResetInputNormalization: 1

See Also

trainNetwork | trainingOptions

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
Introduced in R2016a
TrainingOptionsRMSProp

Training options for RMSProp optimizer

Description

Training options for RMSProp (root mean square propagation) optimizer, including learning rate information, L2 regularization factor, and mini-batch size.

Creation

Create a TrainingOptionsRMSProp object using trainingOptions and specifying 'rmsprop' as the solverName input argument.

Properties

Plots and Display

Plots — Plots to display during network training

'none' | 'training-progress'

Plots to display during network training, specified as one of the following:

- 'none' — Do not display plots during training.
- 'training-progress' — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.

Verbose — Indicator to display training progress information

1 | 0

Indicator to display training progress information in the command window, specified as 1 (true) or 0 (false).
The displayed information includes the epoch number, iteration number, time elapsed, mini-batch loss, mini-batch accuracy, and base learning rate. When you train a regression network, root mean square error (RMSE) is shown instead of accuracy. If you validate the network during training, then the displayed information also includes the validation loss and validation accuracy (or RMSE).

Data Types: logical

**VerboseFrequency — Frequency of verbose printing**

positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as a positive integer. This property only has an effect when the Verbose value equals true.

If you validate the network during training, then trainNetwork prints to the command window every time validation occurs.

**Mini-Batch Options**

**MaxEpochs — Maximum number of epochs**

positive integer

Maximum number of epochs to use for training, specified as a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

**MiniBatchSize — Size of mini-batch**

positive integer

Size of the mini-batch to use for each training iteration, specified as a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights.

**Shuffle — Option for data shuffling**

'once' | 'never' | 'every-epoch'

Option for data shuffling, specified as one of the following:

- 'once' — Shuffle the training and validation data once before training.
- 'never' — Do not shuffle the data.
- 'every-epoch' — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then \texttt{trainNetwork} discards the training data that does not fit into the final complete mini-batch of each epoch. Set the Shuffle value to 'every-epoch' to avoid discarding the same data every epoch.

### Validation

**ValidationData — Data to use for validation during training**

image datastore | datastore | table | cell array

Data to use for validation during training, specified as an image datastore, a datastore that returns data in a two-column table or two-column cell array, a table, or a cell array. The format of the validation data depends on the type of task and correspond to valid inputs to the \texttt{trainNetwork} function.

This option supports networks with a single input only.

### Image Data

For image data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image datastore</td>
<td>\texttt{ImageDatastore} object with categorical labels.</td>
<td>\texttt{imds} argument of \texttt{trainNetwork}</td>
</tr>
<tr>
<td>Datastore</td>
<td>Datastore that returns data in a two-column table or two-column cell array,</td>
<td>\texttt{ds} argument of \texttt{trainNetwork}</td>
</tr>
<tr>
<td></td>
<td>where the two columns specify the network inputs and expected responses, respectively.</td>
<td></td>
</tr>
</tbody>
</table>
### Sequence and Time Series Data

For sequence and time series data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array {sequences, Y}</td>
<td>sequences</td>
<td>sequences argument of trainNetwork</td>
</tr>
<tr>
<td><strong>sequences</strong></td>
<td>Cell array of sequences or time series data.</td>
<td></td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>Categorical vector of labels, cell array of categorical sequences, matrix of numeric responses, or cell array of numeric sequences.</td>
<td>Y argument of trainNetwork</td>
</tr>
<tr>
<td>Table</td>
<td>Table containing absolute or relative file paths to a MAT files containing sequence or time series data.</td>
<td>tbl argument of trainNetwork</td>
</tr>
</tbody>
</table>
During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the 'ValidationFrequency' name-value pair argument. You can also use the validation data to stop training automatically when the validation loss stops decreasing. To turn on automatic validation stopping, use the 'ValidationPatience' name-value pair argument.

If your network has layers that behave differently during prediction than during training (for example, dropout layers), then the validation accuracy can be higher than the training (mini-batch) accuracy.

The validation data is shuffled according to the 'Shuffle' value. If the 'Shuffle' value equals 'every-epoch', then the validation data is shuffled before each network validation.

**ValidationFrequency — Frequency of network validation**

positive integer

Frequency of network validation in number of iterations, specified as a positive integer.

The ValidationFrequency value is the number of iterations between evaluations of validation metrics.

**ValidationPatience — Patience of validation stopping**

positive integer | Inf

Patience of validation stopping of network training, specified as a positive integer or Inf.

The 'ValidationPatience' value is the number of times that the loss on the validation set can be larger than or equal to the previously smallest loss before network training stops.

**Solver Options**

**InitialLearnRate — Initial learning rate**

positive scalar

Initial learning rate used for training, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result.
LearnRateScheduleSettings — Settings for learning rate schedule
structure

Settings for the learning rate schedule, specified as a structure. LearnRateScheduleSettings has the field Method, which specifies the type of method for adjusting the learning rate. The possible methods are:

- 'none' — The learning rate is constant throughout training.
- 'piecewise' — The learning rate drops periodically during training.

If Method is 'piecewise', then LearnRateScheduleSettings contains two more fields:

- DropRateFactor — The multiplicative factor by which the learning rate drops during training
- DropPeriod — The number of epochs that passes between adjustments to the learning rate during training

Specify the settings for the learning schedule rate using trainingOptions.

Data Types: struct

L2Regularization — Factor for $L_2$ regularizer
nonnegative scalar

Factor for $L_2$ regularizer (weight decay), specified as a nonnegative scalar.

You can specify a multiplier for the $L_2$ regularizer for network layers with learnable parameters.

SquaredGradientDecayFactor — Decay rate of squared gradient moving average
scalar from 0 to 1

Decay rate of squared gradient moving average, specified as a scalar from 0 to 1. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.

Epsilon — Denominator offset
positive scalar

Denominator offset, specified as a positive scalar. The solver adds the offset to the denominator in the network parameter updates to avoid division by zero.
**ResetInputNormalization — Option to reset input layer normalization**
true (default) | false

Option to reset input layer normalization, specified as one of the following:

- **true** - Reset the input layer normalization statistics and recalculate them at training time.
- **false** - Calculate normalization statistics at training time when they are empty.

**Gradient Clipping**

**GradientThreshold — Gradient threshold**
positive scalar | Inf

Positive threshold for the gradient, specified as positive scalar or Inf. When the gradient exceeds the value of **GradientThreshold**, then the gradient is clipped according to **GradientThresholdMethod**.

**GradientThresholdMethod — Gradient threshold method**
'l2norm' | 'global-l2norm' | 'absolutevalue'

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following:

- **'l2norm'** — If the L \(_2\) norm of the gradient of a learnable parameter is larger than **GradientThreshold**, then scale the gradient so that the L \(_2\) norm equals **GradientThreshold**.
- **'global-l2norm'** — If the global L \(_2\) norm, \(L\), is larger than **GradientThreshold**, then scale all gradients by a factor of **GradientThreshold**/\(L\). The global L \(_2\) norm considers all learnable parameters.
- **'absolute-value'** — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than **GradientThreshold**, then scale the partial derivative to have magnitude equal to **GradientThreshold** and retain the sign of the partial derivative.

For more information, see Gradient Clipping on page 1-694.
**Sequence Options**

**SequenceLength — Option to pad or truncate sequences**

'longest' | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingDirection — Direction of padding or truncation**

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer OutputMode property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the OutputMode property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the
predictions for the earlier time steps. To pad or truncate sequence data on the right, set the 'SequencePaddingDirection' option to 'right'.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingValue — Value to pad sequences**

scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when SequenceLength is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

**Hardware Options**

**ExecutionEnvironment — Hardware resource for training network**

'auto' | 'cpu' | 'gpu' | 'multi-gpu' | 'parallel'

Hardware resource for training network, specified as one of the following:

- 'auto' — Use a GPU if one is available. Otherwise, use the CPU.
- 'cpu' — Use the CPU.
- 'gpu' — Use the GPU.
- 'multi-gpu' — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- 'parallel' — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel and in the Cloud”.

GPU, multi-GPU, and parallel options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
To see an improvement in performance when training in parallel, try scaling up the MiniBatchSize and InitialLearnRate training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Specify the execution environment using trainingOptions. 
Data Types: char | string

WorkerLoad — Parallel worker load division

Worker load division for GPUs or CPUs, specified as a scalar from 0 to 1, a positive integer, or a numeric vector. This property has an effect only when the ExecutionEnvironment value equals 'multi-gpu' or 'parallel'.

Checkpoints

CheckpointPath — Path for saving checkpoint networks

Path where checkpoint networks are saved, specified as a character vector.
Data Types: char

OutputFcn — Output functions

Output functions to call during training, specified as a function handle or cell array of function handles. trainNetwork calls the specified functions once before the start of training, after each iteration, and once after training has finished. trainNetwork passes a structure containing information in the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch</td>
<td>Current epoch number</td>
</tr>
<tr>
<td>Iteration</td>
<td>Current iteration number</td>
</tr>
<tr>
<td>TimeSinceStart</td>
<td>Time in seconds since the start of training</td>
</tr>
<tr>
<td>TrainingLoss</td>
<td>Current mini-batch loss</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>ValidationLoss</td>
<td>Loss on the validation data</td>
</tr>
<tr>
<td>BaseLearnRate</td>
<td>Current base learning rate</td>
</tr>
<tr>
<td>TrainingAccuracy</td>
<td>Accuracy on the current mini-batch (classification networks)</td>
</tr>
<tr>
<td>TrainingRMSE</td>
<td>RMSE on the current mini-batch (regression networks)</td>
</tr>
<tr>
<td>ValidationAccuracy</td>
<td>Accuracy on the validation data (classification networks)</td>
</tr>
<tr>
<td>ValidationRMSE</td>
<td>RMSE on the validation data (regression networks)</td>
</tr>
<tr>
<td>State</td>
<td>Current training state, with a possible value of &quot;start&quot;, &quot;iteration&quot;, or &quot;done&quot;.</td>
</tr>
</tbody>
</table>

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

You can use output functions to display or plot progress information, or to stop training. To stop training early, make your output function return `true`. If any output function returns `true`, then training finishes and `trainNetwork` returns the latest network. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training”.

Data Types: `function_handle` | `cell`

**Examples**

**Create Training Options for the RMSProp Optimizer**

Create a set of options for training a neural network using the RMSProp optimizer. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Specify the learning rate and the decay rate of the moving average of the squared gradient. Turn on the training progress plot.

```matlab
options = trainingOptions('rmsprop', ...
    'InitialLearnRate',3e-4, ...
    'SquaredGradientDecayFactor',0.99, ...
    'MaxEpochs',20, ...)
```
options = TrainingOptionsRMSProp with properties:

    SquaredGradientDecayFactor: 0.9900
        Epsilon: 1.0000e-08
        InitialLearnRate: 3.0000e-04
    LearnRateScheduleSettings: [1x1 struct]
        L2Regularization: 1.0000e-04
        GradientThresholdMethod: 'l2norm'
        GradientThreshold: Inf
        MaxEpochs: 20
        MiniBatchSize: 64
        Verbose: 1
        VerboseFrequency: 50
        ValidationData: []
        ValidationFrequency: 50
        ValidationPatience: Inf
        Shuffle: 'once'
        CheckpointPath: ''
        ExecutionEnvironment: 'auto'
        WorkerLoad: []
        OutputFcn: []
        Plots: 'training-progress'
    SequenceLength: 'longest'
        SequencePaddingValue: 0
        SequencePaddingDirection: 'right'
        DispatchInBackground: 0
        ResetInputNormalization: 1

See Also
trainNetwork | trainingOptions

Topics
“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

**Introduced in R2018a**
TrainingOptionsADAM

Training options for Adam optimizer

Description

Training options for Adam (adaptive moment estimation) optimizer, including learning rate information, $L_2$ regularization factor, and mini-batch size.

Creation

Create a TrainingOptionsADAM object using trainingOptions and specifying 'adam' as the solverName input argument.

Properties

Plots and Display

Plots — Plots to display during network training

'none' | 'training-progress'

Plots to display during network training, specified as one of the following:

- 'none' — Do not display plots during training.
- 'training-progress' — Plot training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot has a stop button in the top-right corner. Click the button to stop training and return the current state of the network.

Verbose — Indicator to display training progress information

1 | 0

Indicator to display training progress information in the command window, specified as 1 (true) or 0 (false).
The displayed information includes the epoch number, iteration number, time elapsed, mini-batch loss, mini-batch accuracy, and base learning rate. When you train a regression network, root mean square error (RMSE) is shown instead of accuracy. If you validate the network during training, then the displayed information also includes the validation loss and validation accuracy (or RMSE).

Data Types: logical

VerboseFrequency — Frequency of verbose printing

positive integer

Frequency of verbose printing, which is the number of iterations between printing to the command window, specified as a positive integer. This property only has an effect when the Verbose value equals true.

If you validate the network during training, then trainNetwork prints to the command window every time validation occurs.

Mini-Batch Options

MaxEpochs — Maximum number of epochs

positive integer

Maximum number of epochs to use for training, specified as a positive integer.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini-batch. An epoch is the full pass of the training algorithm over the entire training set.

MiniBatchSize — Size of mini-batch

positive integer

Size of the mini-batch to use for each training iteration, specified as a positive integer. A mini-batch is a subset of the training set that is used to evaluate the gradient of the loss function and update the weights.

Shuffle — Option for data shuffling

'onces' | 'never' | 'every-epoch'

Option for data shuffling, specified as one of the following:

- 'once' — Shuffle the training and validation data once before training.
• 'never' — Do not shuffle the data.
• 'every-epoch' — Shuffle the training data before each training epoch, and shuffle the validation data before each network validation. If the mini-batch size does not evenly divide the number of training samples, then trainNetwork discards the training data that does not fit into the final complete mini-batch of each epoch. Set the Shuffle value to 'every-epoch' to avoid discarding the same data every epoch.

**Validation**

**ValidationData — Data to use for validation during training**
image datastore | datastore | table | cell array

Data to use for validation during training, specified as an image datastore, a datastore that returns data in a two-column table or two-column cell array, a table, or a cell array. The format of the validation data depends on the type of task and correspond to valid inputs to the trainNetwork function.

This option supports networks with a single input only.

**Image Data**

For image data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image datastore</td>
<td>ImageDatastore object with categorical labels.</td>
<td>imds argument of trainNetwork</td>
</tr>
<tr>
<td>Datastore</td>
<td>Datastore that returns data in a two-column table or two-column cell array, where the two columns specify the network inputs and expected responses, respectively.</td>
<td>ds argument of trainNetwork</td>
</tr>
<tr>
<td>Input</td>
<td>Description</td>
<td>More Information</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Table</td>
<td>Table, where the first column contains either image paths or images, and the</td>
<td>tbl argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>subsequent columns contain the responses.</td>
<td></td>
</tr>
<tr>
<td>Cell array {X, Y}</td>
<td>X Numeric array of images.</td>
<td>X argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>Y Categorical vector of labels, matrix of numeric responses, or array of</td>
<td>Y argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>images.</td>
<td></td>
</tr>
</tbody>
</table>

**Sequence and Time Series Data**

For sequence and time series data, specify the validation data as one of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array {sequences, Y}</td>
<td>sequences Cell array of sequences or time series data.</td>
<td>sequences argument of trainNetwork</td>
</tr>
<tr>
<td></td>
<td>Y Categorical vector of labels, cell array of categorical sequences, matrix of numeric responses, or cell array of numeric sequences.</td>
<td>Y argument of trainNetwork</td>
</tr>
<tr>
<td>Table</td>
<td>Table containing absolute or relative file paths to a MAT files containing sequence or time series data.</td>
<td>tbl argument of trainNetwork</td>
</tr>
</tbody>
</table>
During training, `trainNetwork` calculates the validation accuracy and validation loss on the validation data. To specify the validation frequency, use the 'ValidationFrequency' name-value pair argument. You can also use the validation data to stop training automatically when the validation loss stops decreasing. To turn on automatic validation stopping, use the 'ValidationPatience' name-value pair argument.

If your network has layers that behave differently during prediction than during training (for example, dropout layers), then the validation accuracy can be higher than the training (mini-batch) accuracy.

The validation data is shuffled according to the 'Shuffle' value. If the 'Shuffle' value equals 'every-epoch', then the validation data is shuffled before each network validation.

**ValidationFrequency — Frequency of network validation**
positive integer

Frequency of network validation in number of iterations, specified as a positive integer.

The `ValidationFrequency` value is the number of iterations between evaluations of validation metrics.

**ValidationPatience — Patience of validation stopping**
positive integer | Inf

Patience of validation stopping of network training, specified as a positive integer or `Inf`.

The 'ValidationPatience' value is the number of times that the loss on the validation set can be larger than or equal to the previously smallest loss before network training stops.

**Solver Options**

**InitialLearnRate — Initial learning rate**
positive scalar

Initial learning rate used for training, specified as a positive scalar. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result.
LearnRateScheduleSettings — Settings for learning rate schedule
structure

Settings for the learning rate schedule, specified as a structure.
LearnRateScheduleSettings has the field Method, which specifies the type of method for adjusting the learning rate. The possible methods are:

- 'none' — The learning rate is constant throughout training.
- 'piecewise' — The learning rate drops periodically during training.

If Method is 'piecewise', then LearnRateScheduleSettings contains two more fields:

- DropRateFactor — The multiplicative factor by which the learning rate drops during training
- DropPeriod — The number of epochs that passes between adjustments to the learning rate during training

Specify the settings for the learning schedule rate using trainingOptions.

Data Types: struct

L2Regularization — Factor for L\textsuperscript{2} regularizer
nonnegative scalar

Factor for L\textsuperscript{2} regularizer (weight decay), specified as a nonnegative scalar.

You can specify a multiplier for the L\textsuperscript{2} regularizer for network layers with learnable parameters.

GradientDecayFactor — Decay rate of gradient moving average
scalar from 0 to 1

Decay rate of gradient moving average, specified as a scalar from 0 to 1. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.

SquaredGradientDecayFactor — Decay rate of squared gradient moving average
scalar from 0 to 1

Decay rate of squared gradient moving average, specified as a scalar from 0 to 1. For more information about the different solvers, see “Stochastic Gradient Descent” on page 1-691.
**Epsilon — Denominator offset**

Positive scalar

Denominator offset, specified as a positive scalar. The solver adds the offset to the denominator in the network parameter updates to avoid division by zero.

**ResetInputNormalization — Option to reset input layer normalization**

**true (default) | false**

Option to reset input layer normalization, specified as one of the following:

- **true** - Reset the input layer normalization statistics and recalculate them at training time.
- **false** - Calculate normalization statistics at training time when they are empty.

**Gradient Clipping**

**GradientThreshold — Gradient threshold**

Positive scalar | Inf

Positive threshold for the gradient, specified as positive scalar or Inf. When the gradient exceeds the value of **GradientThreshold**, then the gradient is clipped according to **GradientThresholdMethod**.

**GradientThresholdMethod — Gradient threshold method**

'**l2norm' | 'global-l2norm' | 'absolutevalue'**

Gradient threshold method used to clip gradient values that exceed the gradient threshold, specified as one of the following:

- **'l2norm'** — If the L$_2$ norm of the gradient of a learnable parameter is larger than **GradientThreshold**, then scale the gradient so that the L$_2$ norm equals **GradientThreshold**.
- **'global-l2norm'** — If the global L$_2$ norm, L, is larger than **GradientThreshold**, then scale all gradients by a factor of **GradientThreshold**/L. The global L$_2$ norm considers all learnable parameters.
- **'absolute-value'** — If the absolute value of an individual partial derivative in the gradient of a learnable parameter is larger than **GradientThreshold**, then scale the partial derivative to have magnitude equal to **GradientThreshold** and retain the sign of the partial derivative.
For more information, see Gradient Clipping on page 1-694.

**Sequence Options**

**SequenceLength — Option to pad or truncate sequences**

'longest' | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.

- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.

- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingDirection — Direction of padding or truncation**

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.

- 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer `OutputMode` property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.
For sequence-to-sequence networks (when the `OutputMode` property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the 'SequencePaddingDirection' option to 'right'.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingValue — Value to pad sequences**

scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

**Hardware Options**

**ExecutionEnvironment — Hardware resource for training network**

'auto' | 'cpu' | 'gpu' | 'multi-gpu' | 'parallel'

Hardware resource for training network, specified as one of the following:

- 'auto' — Use a GPU if one is available. Otherwise, use the CPU.
- 'cpu' — Use the CPU.
- 'gpu' — Use the GPU.
- 'multi-gpu' — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs.
- 'parallel' — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For more information on when to use the different execution environments, see “Scale Up Deep Learning in Parallel and in the Cloud”.

GPU, multi-GPU, and parallel options require Parallel Computing Toolbox. To use a GPU for deep learning, you must also have a CUDA enabled NVIDIA GPU with compute...
capability 3.0 or higher. If you choose one of these options and Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.

To see an improvement in performance when training in parallel, try scaling up the `MiniBatchSize` and `InitialLearnRate` training options by the number of GPUs.

Training long short-term memory networks supports single CPU or single GPU training only.

Specify the execution environment using `trainingOptions`.

Data Types: `char` | `string`

**WorkerLoad — Parallel worker load division**

Scalar from 0 to 1 | Positive integer | Numeric vector

Worker load division for GPUs or CPUs, specified as a scalar from 0 to 1, a positive integer, or a numeric vector. This property has an effect only when the `ExecutionEnvironment` value equals `'multi-gpu'` or `'parallel'`.

**Checkpoints**

**CheckpointPath — Path for saving checkpoint networks**

Character vector

Path where checkpoint networks are saved, specified as a character vector.

Data Types: `char`

**OutputFcn — Output functions**

Function handle | Cell array of function handles

Output functions to call during training, specified as a function handle or cell array of function handles. `trainNetwork` calls the specified functions once before the start of training, after each iteration, and once after training has finished. `trainNetwork` passes a structure containing information in the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch</td>
<td>Current epoch number</td>
</tr>
<tr>
<td>Iteration</td>
<td>Current iteration number</td>
</tr>
</tbody>
</table>

1-1191
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeSinceStart</td>
<td>Time in seconds since the start of training</td>
</tr>
<tr>
<td>TrainingLoss</td>
<td>Current mini-batch loss</td>
</tr>
<tr>
<td>ValidationLoss</td>
<td>Loss on the validation data</td>
</tr>
<tr>
<td>BaseLearnRate</td>
<td>Current base learning rate</td>
</tr>
<tr>
<td>TrainingAccuracy</td>
<td>Accuracy on the current mini-batch (classification networks)</td>
</tr>
<tr>
<td>TrainingRMSE</td>
<td>RMSE on the current mini-batch (regression networks)</td>
</tr>
<tr>
<td>ValidationAccuracy</td>
<td>Accuracy on the validation data (classification networks)</td>
</tr>
<tr>
<td>ValidationRMSE</td>
<td>RMSE on the validation data (regression networks)</td>
</tr>
<tr>
<td>State</td>
<td>Current training state, with a possible value of &quot;start&quot;, &quot;iteration&quot;, or &quot;done&quot;.</td>
</tr>
</tbody>
</table>

If a field is not calculated or relevant for a certain call to the output functions, then that field contains an empty array.

You can use output functions to display or plot progress information, or to stop training. To stop training early, make your output function return true. If any output function returns true, then training finishes and `trainNetwork` returns the latest network. For an example showing how to use output functions, see “Customize Output During Deep Learning Network Training”.

Data Types: `function_handle` | `cell`

### Examples

**Create Training Options for the Adam Optimizer**

Create a set of options for training a neural network using the Adam optimizer. Set the maximum number of epochs for training to 20, and use a mini-batch with 64 observations at each iteration. Specify the learning rate and the decay rate of the moving average of the squared gradient. Turn on the training progress plot.
options = trainingOptions('adam', ...
    'InitialLearnRate',3e-4, ...
    'SquaredGradientDecayFactor',0.99, ...
    'MaxEpochs',20, ...
    'MiniBatchSize',64, ...
    'Plots','training-progress')

options = TrainingOptionsADAM with properties:

    GradientDecayFactor: 0.9000
    SquaredGradientDecayFactor: 0.9900
       Epsilon: 1.0000e-08
    InitialLearnRate: 3.0000e-04
          LearnRateScheduleSettings: [1x1 struct]
   L2Regularization: 1.0000e-04
        GradientThresholdMethod: 'l2norm'
    GradientThreshold: Inf
        MaxEpochs: 20
  MiniBatchSize: 64
     Verbose: 1
   VerboseFrequency: 50
    ValidationData: []
      ValidationFrequency: 50
  ValidationPatience: Inf
        Shuffle: 'once'
      CheckpointPath: ''
ExecutionEnvironment: 'auto'
      WorkerLoad: []
     OutputFcn: []
            Plots: 'training-progress'
      SequenceLength: 'longest'
   SequencePaddingValue: 0
SequencePaddingDirection: 'right'
  DispatchInBackground: 0
   ResetInputNormalization: 1

References

See Also

trainNetwork | trainingOptions

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

Introduced in R2018a
activations

Compute deep learning network layer activations

Syntax

features = activations(net,imds,layer)
features = activations(net,ds,layer)
features = activations(net,X,layer)
features = activations(net,sequences,layer)
features = activations(net,tbl,layer)
features = activations(___,Name,Value)

Description

You can extract features using a trained deep learning network on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. Specify the hardware requirements using the ExecutionEnvironment name-value pair argument.

features = activations(net,imds,layer) returns network activations for a specific layer using the trained network net and the image data in the image datastore imds.

features = activations(net,ds,layer) returns network activations using the data in the datastore ds. For networks with multiple inputs, use this syntax with a combined or transformed datastore object.

features = activations(net,X,layer) returns network activations using the image data in the numeric array X.

features = activations(net,sequences,layer) returns network activations for an LSTM or BiLSTM network, where sequences contains sequence or time series predictors.
features = activations(net,tbl,layer) returns network activations using the data in the table of images tbl.

features = activations(___,Name,Value) returns network activations with additional options specified by one or more name-value pair arguments. For example, 'OutputAs','rows' specifies the activation output format as 'rows'. Specify name-value pair arguments after all other input arguments.

Examples

Feature Extraction Using AlexNet

This example shows how to extract learned image features from a pretrained convolutional neural network, and use those features to train an image classifier. Feature extraction is the easiest and fastest way use the representational power of pretrained deep networks. For example, you can train a support vector machine (SVM) using fitcecoc (Statistics and Machine Learning Toolbox™) on the extracted features. Because feature extraction only requires a single pass through the data, it is a good starting point if you do not have a GPU to accelerate network training with.

Load Data

Unzip and load the sample images as an image datastore. imageDatastore automatically labels the images based on folder names and stores the data as an ImageDatastore object. An image datastore lets you store large image data, including data that does not fit in memory. Split the data into 70% training and 30% test data.

unzip('MerchData.zip');

imds = imageDatastore('MerchData',...'
    IncludeSubfolders',true,...
    'LabelSource','foldernames');

[imdsTrain,imdsTest] = splitEachLabel(imds,0.7,'randomized');

There are now 55 training images and 20 validation images in this very small data set. Display some sample images.

numImagesTrain = numel(imdsTrain.Labels);
idx = randperm(numImagesTrain,16);
for i = 1:16
    I{i} = readimage(imdsTrain, idx(i));
end

figure
imshow(imtile(I))

activations
Load Pretrained Network

Load a pretrained AlexNet network. If the Deep Learning Toolbox Model for AlexNet Network support package is not installed, then the software provides a download link. AlexNet is trained on more than a million images and can classify images into 1000 object categories. For example, keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

net = alexnet;

Display the network architecture. The network has five convolutional layers and three fully connected layers.

net.Layers

ans =
25x1 Layer array with layers:
1  'data'   Image Input
2  'conv1'  Convolution
3   'relu1'  ReLU
4  'norm1'  Cross Channel Normalization
5  'pool1'  Max Pooling
6  'conv2'  Grouped Convolution
7   'relu2'  ReLU
8  'norm2'  Cross Channel Normalization
9  'pool2'  Max Pooling
10  'conv3'  Convolution
11  'relu3'  ReLU
12  'drop6'  Dropout
13  'fc6'    Fully Connected
14  'relu6'  ReLU
15  'fc7'    Fully Connected
16  'relu7'  ReLU
17  'drop7'  Dropout
18  'fc8'    Fully Connected
19  'prob'   Softmax
20  'output' Classification Output

 activs

1-1199
The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

\[ \text{inputSize} = \text{net.Layers}(1).\text{InputSize} \]

\[ \text{inputSize} = 1\times3 \]

\[ 227 \quad 227 \quad 3 \]

**Extract Image Features**

The network constructs a hierarchical representation of input images. Deeper layers contain higher-level features, constructed using the lower-level features of earlier layers. To get the feature representations of the training and test images, use \textit{activations} on the fully connected layer 'fc7'. To get a lower-level representation of the images, use an earlier layer in the network.

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. To automatically resize the training and test images before they are input to the network, create augmented image datastores, specify the desired image size, and use these datastores as input arguments to \textit{activations}.

\[ \text{augimdsTrain} = \text{augmentedImageDatastore}('inputSize'(1:2),\text{imdsTrain}); \]
\[ \text{augimdsTest} = \text{augmentedImageDatastore}('inputSize'(1:2),\text{imdsTest}); \]

\[ \text{layer} = 'fc7'; \]
\[ \text{featuresTrain} = \text{activations}('net',\text{augimdsTrain},\text{layer},'OutputAs','rows'); \]
\[ \text{featuresTest} = \text{activations}('net',\text{augimdsTest},\text{layer},'OutputAs','rows'); \]

Extract the class labels from the training and test data.

\[ \text{YTrain} = \text{imdsTrain}.\text{Labels}; \]
\[ \text{YTest} = \text{imdsTest}.\text{Labels}; \]

**Fit Image Classifier**

Use the features extracted from the training images as predictor variables and fit a multiclass support vector machine (SVM) using \textit{fitcecoc} (Statistics and Machine Learning Toolbox).

\[ \text{mdl} = \text{fitcecoc}('featuresTrain',\text{YTrain}); \]
Classify Test Images

Classify the test images using the trained SVM model and the features extracted from the test images.

YPred = predict(mdl,featuresTest);

Display four sample test images with their predicted labels.

idx = [1 5 10 15];
figure
for i = 1:numel(idx)
    subplot(2,2,i)
    I = readimage(imdsTest,idx(i));
    label = YPred(idx(i));
    imshow(I)
    title(label)
end
Calculate the classification accuracy on the test set. Accuracy is the fraction of labels that the network predicts correctly.

\[
\text{accuracy} = \frac{\text{mean}(\text{YPred} == \text{YTest})}{1}
\]

accuracy = 1
This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

**Input Arguments**

**net** — Trained network

*SeriesNetwork object | DAGNetwork object*

Trained network, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can get a trained network by importing a pretrained network (for example, by using the `googlenet` function) or by training your own network using `trainNetwork`.

**imds** — Image datastore

*ImageDatastore object*

Image datastore, specified as an `ImageDatastore` object. `ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` does not prefetch.

**ds** — Datastore
datastore

Dataset of input data.

For networks with a single input, `ds` can return either:

- a single image
- a cell array of images
- a table, where the first column contains images
For networks with multiple inputs, \( ds \) must be a combined or transformed datastore that returns a cell array with \( \text{numInputs} \) columns containing the input data, where \( \text{numInputs} \) is the number of network inputs. The \( i \)th element of the cell array corresponds to the input \( \text{net.InputNames}(i) \).

For more information, see “Datastores for Deep Learning”.

**X — Image data**

**numeric array**

Image data, specified as a numeric array. The size of the array depends on the type of image input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D images</td>
<td>A ( h )-by-( w )-by-( c )-by-( N ) numeric array, where ( h ), ( w ), and ( c ) are the height, width, and number of channels of the images, respectively, and ( N ) is the number of images.</td>
</tr>
<tr>
<td>3-D images</td>
<td>A ( h )-by-( w )-by-( d )-by-( c )-by-( N ) numeric array, where ( h ), ( w ), ( d ), and ( c ) are the height, width, depth, and number of channels of the images, respectively, and ( N ) is the number of images.</td>
</tr>
</tbody>
</table>

If the array contains \( \text{NaN}s \), then they are propagated through the network.

For image input, if the 'OutputAs' option is 'channels', then the images in the input data \( X \) can be larger than the input size of the image input layer of the network. For other output formats, the images in \( X \) must have the same size as the input size of the image input layer of the network.

**sequences — Sequence or time series data**

**cell array of numeric arrays | numeric array | datastore**

Sequence or time series data, specified as an \( N \)-by-1 cell array of numeric arrays, where \( N \) is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.
Vector sequences  
\[ c \text{-by-} s \] matrices, where \( c \) is the number of features of the sequences and \( s \) is the sequence length.

2-D image sequences  
\[ h\text{-by-}w\text{-by-}c\text{-by-}s \] arrays, where \( h \), \( w \), and \( c \) correspond to the height, width, and number of channels of the images, respectively, and \( s \) is the sequence length.

3-D image sequences  
\[ h\text{-by-}w\text{-by-}d\text{-by-}c\text{-by-}s \], where \( h \), \( w \), \( d \), and \( c \) correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and \( s \) is the sequence length.

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

**tbl — Table of images**

Table of images containing the input data in the first column. Each row in the table corresponds to an observation. The table contains absolute or relative file paths to an image, specified as a character vector or images specified as a numeric array.

Data Types: `table`

**layer — Layer to extract features from**

Layer to extract features from, specified as a numeric index or a character vector.

To compute the activations of a `SeriesNetwork` object, specify the layer using its numeric index, or as a character vector corresponding to the layer name.

To compute the activations of a `DAGNetwork` object, specify the layer as the character vector corresponding to the layer name. If the layer has multiple outputs, specify the layer and output as the layer name, followed by the character “/”, followed by the name of the layer output. That is, `layer` is of the form 'layerName/outputName'.

Example: `3`

Example: `'conv1'`
Example: 'mpool/out'

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: activations(net,X,layer,'OutputAs','rows')

**OutputAs — Format of output activations**

'channels' (default) | 'rows' | 'columns'

Format of output activations, specified as the comma-separated pair consisting of 'OutputAs' and either 'channels', 'rows', or 'columns'. For descriptions of the different output formats, see features.

For image input, if the 'OutputAs' option is 'channels', then the images in the input data X can be larger than the input size of the image input layer of the network. For other output formats, the images in X must have the same size as the input size of the image input layer of the network.

Example: 'OutputAs','rows'

**MiniBatchSize — Size of mini-batches**

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

Example: 'MiniBatchSize',256

**SequenceLength — Option to pad, truncate, or split input sequences**

'longest' (default) | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
• 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.

• Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: 'SequenceLength','shortest'

**SequencePaddingValue — Value to pad input sequences**

0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

Example: 'SequencePaddingValue',-1

**SequencePaddingDirection — Direction of padding or truncation**

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

• 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.

• 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer `OutputMode` property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the `OutputMode` property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the
predictions for the earlier time steps. To pad or truncate sequence data on the right, set
the 'SequencePaddingDirection' option to 'right'.

To learn more about the effect of padding, truncating, and splitting the input sequences,
see “Sequence Padding, Truncation, and Splitting”.

**Acceleration — Performance optimization**

'auto' (default) | 'mex' | 'none'

Performance optimization, specified as the comma-separated pair consisting of
'Acceleration' and one of the following:

- 'auto' — Automatically apply a number of optimizations suitable for the input
  network and hardware resource.
- 'mex' — Compile and execute a MEX function. This option is available when using a
  GPU only. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled
  NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a
  suitable GPU is not available, then the software returns an error.
- 'none' — Disable all acceleration.

The default option is 'auto'. If 'auto' is specified, MATLAB will apply a number of
compatible optimizations. If you use the 'auto' option, MATLAB does not ever generate
a MEX function.

Using the 'Acceleration' options 'auto' and 'mex' can offer performance benefits,
but at the expense of an increased initial run time. Subsequent calls with compatible
parameters are faster. Use performance optimization when you plan to call the function
multiple times using new input data.

The 'mex' option generates and executes a MEX function based on the network and
parameters used in the function call. You can have several MEX functions associated with
a single network at one time. Clearing the network variable also clears any MEX functions
associated with that network.

The 'mex' option is only available when you are using a GPU. You must have a C/C++
compiler installed and the GPU Coder Interface for Deep Learning Libraries support
package. Install the support package using the Add-On Explorer in MATLAB. For setup
instructions, see “MEX Setup” (GPU Coder). GPU Coder is not required.

The 'mex' option does not support all layers. For a list of supported layers, see
“Supported Layers” (GPU Coder). Recurrent neural networks (RNNs) containing a
sequenceInputLayer are not supported.
You cannot use MATLAB Compiler to deploy your network when using the 'mex' option.

Example: 'Acceleration','mex'

**ExecutionEnvironment — Hardware resource**

'auto' (default) | 'gpu' | 'cpu'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
- 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'cpu' — Use the CPU.

Example: 'ExecutionEnvironment','cpu'

**Output Arguments**

**features — Activations from the network layer**

numeric array | cell array

Activations from the network layer, returned as a numeric array or a cell array of numeric arrays. The format of features depends on the type of input data, the type of layer output, and the 'OutputAs' option.

**Image Input**

If net has an image input layer, then features is a numeric array.
<table>
<thead>
<tr>
<th>'OutputAs'</th>
<th>features</th>
</tr>
</thead>
</table>
| 'channels' | For 2-D image data, features is an $h$-by-$w$-by-$c$-by-$n$ array, where $h$, $w$, and $c$ are the height, width, and number of channels for the output of the chosen layer, respectively, and $n$ is the number of images. In this case, features($,:,:,:,i$) contains the activations for the $i$th image.  
For 3-D image data, features is an $h$-by-$w$-by-$d$-by-$c$-by-$n$ array, where $h$, $w$, $d$, and $c$ are the height, width, depth, and number of channels for the output of the chosen layer, respectively, and $n$ is the number of images. In this case, features($,:,:,:,i$) contains the activations for the $i$th image. |
| 'rows'     | $n$-by-$m$ matrix, where $n$ is the number of observations, and $m$ is the number of output elements from the chosen layer. In this case, features$(i,:)$ contains the activations for the $i$th image. |
| 'columns'  | $m$-by-$n$ matrix, where $m$ is the number of output elements from the chosen layer, and $n$ is the number of observations. In this case, features($:,:,i$) contains the activations for the $i$th image. |

**Sequence Input**

If net has a sequence input layer, and layer has sequence output (for example, LSTM layers with output mode 'sequence'), then features is a cell array. In this case, the 'OutputAs' option must be 'channels'.
<table>
<thead>
<tr>
<th>'OutputAs'</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>'channels'</td>
<td></td>
</tr>
</tbody>
</table>

For vector sequence input, `features` is a `n`-by-1 cell array, of `c`-by-`s` matrices, where `n` is the number of sequences, `c` is the number of features in the sequence, and `s` is the sequence length.

For 2-D image sequence input, `features` is a `n`-by-1 cell array, of `h`-by-`w`-by-`c`-by-`s` matrices, where `n` is the number of sequences, `h`, `w`, and `c` are the height, width, and the number of channels of the images, respectively, and `s` is the sequence length.

For 3-D image sequence input, `features` is a `n`-by-1 cell array, of `h`-by-`w`-by-`c`-by-`d`-by-`s` matrices, where `n` is the number of sequences, `h`, `w`, `d`, and `c` are the height, width, depth, and the number of channels of the images, respectively, and `s` is the sequence length.

In these cases, `features{1}` contains the activations of the 1st sequence.

If `net` has a sequence input layer and `layer` outputs non-sequence data (for example, an LSTM layer with output mode 'last'), then `features` is a numeric array.

<table>
<thead>
<tr>
<th>'OutputAs'</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>'channels'</td>
<td></td>
</tr>
</tbody>
</table>

For vector sequence input, `features` is a `c`-by-`n` matrix, where `n` is the number of sequences and `c` is the number of features in the sequence.

For 2-D image sequence input, `features` is a `h`-by-`w`-by-`c`-by-`n` array, where `n` is the number of sequences, `h`, `w`, and `c` are the height, width, and the number of channels of the images, respectively.

For 3-D image sequence input, `features` is a `h`-by-`w`-by-`c`-by-`d`-by-`n` array, where `n` is the number of sequences, `h`, `w`, `d`, and `c` are the height, width, depth, and the number of channels of the images, respectively.

In these cases, `features{1}` contains the activations of the 1st sequence.
\textbf{'OutputAs'} & \textbf{features} \\
\hline
'\textit{rows}' & \textit{n-by-}\textit{m} matrix, where \textit{n} is the number of observations, and \textit{m} is the number of output elements from the chosen layer. In this case, \texttt{features(i,:) \textsubscript{}} contains the activations for the \textit{i}th sequence. \\
\hline
'\textit{columns}' & \textit{m-by-n} matrix, where \textit{m} is the number of output elements from the chosen layer, and \textit{n} is the number of observations. In this case, \texttt{features(:,i) \textsubscript{}} contains the activations for the \textit{i}th image. \\
\hline

\section*{Algorithms}

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include \texttt{trainNetwork}, \texttt{predict}, \texttt{classify}, and \texttt{activations}. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

\section*{References}


\section*{Extended Capabilities}

\subsection*{C/C++ Code Generation}

Generate C and C++ code using MATLAB\textsuperscript{®} Coder\textsuperscript{™}.

Usage notes and limitations:

- The input \texttt{X} must not have a variable size. The size must be fixed at code generation time.
• The layer argument must be constant.
• Only the 'OutputAs' name-value pair argument is supported. The value must be 'channels'.

For more information about generating code for deep learning neural networks, see “Workflow for Deep Learning Code Generation with MATLAB Coder” (MATLAB Coder).

**GPU Code Generation**
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:
• The input X must not have a variable size. The size must be fixed at code generation time.
• The layer argument must be constant.
• Only the 'OutputAs' name-value pair argument is supported. The value must be 'channels'.

**See Also**
classify|deepDreamImage|predict|trainNetwork

**Topics**
“Transfer Learning Using AlexNet”
“Visualize Activations of a Convolutional Neural Network”
“Visualize Activations of LSTM Network”
“Deep Learning in MATLAB”

**Introduced in R2016a**
**classify**

Classify data using a trained deep learning neural network

**Syntax**

\[
[YPred, scores] = classify(net, imds)
\]

\[
[YPred, scores] = classify(net, ds)
\]

\[
[YPred, scores] = classify(net, X)
\]

\[
[YPred, scores] = classify(net, sequences)
\]

\[
[YPred, scores] = classify(net, tbl)
\]

\[
[YPred, scores] = classify(\_, Name, Value)
\]

**Description**

You can make predictions using a trained neural network for deep learning on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. Specify the hardware requirements using the `ExecutionEnvironment` name-value pair argument.

For networks with multiple outputs, use the `predict` ans set the `'ReturnCategorial'` option to `true`.

\[
[YPred, scores] = classify(net, imds)
\] predicts class labels for the image data in `imds` using the trained network, `net`.

\[
[YPred, scores] = classify(net, ds)
\] predicts class labels for the data in `ds` using the trained network, `net`. For networks with multiple inputs, use this syntax with a combined or transformed datastore object.

\[
[YPred, scores] = classify(net, X)
\] predicts class labels for the image data in `X` using the trained network, `net`.

\[
[YPred, scores] = classify(net, sequences)
\] predicts class labels for the time series or sequence data in `sequences` using the trained LSTM network, `net`. 
[YPred, scores] = classify(net, tbl) predicts class labels for the data in tbl using the trained network, net.

[YPred, scores] = classify(___ ,Name, Value) predicts class labels with additional options specified by one or more name-value pair arguments.

Tip When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

Examples

Classify Images Using Trained ConvNet

Load the sample data.

[XTrain, YTrain] = digitTrain4DArrayData;

digitTrain4DArrayData loads the digit training set as 4-D array data. XTrain is a 28-by-28-by-1-by-5000 array, where 28 is the height and 28 is the width of the images. 1 is the number of channels and 5000 is the number of synthetic images of handwritten digits. YTrain is a categorical vector containing the labels for each observation.

Construct the convolutional neural network architecture.

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5, 20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer;];

Set the options to default settings for the stochastic gradient descent with momentum.

options = trainingOptions('sgdm');
Train the network.

```matlab
rng('default')
net = trainNetwork(XTrain,YTrain,layers,options);
```

Training on single CPU.

Initializing input data normalization.

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Iteration</th>
<th>Time Elapsed (hh:mm:ss)</th>
<th>Mini-batch Accuracy</th>
<th>Mini-batch Loss</th>
<th>Base Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>00:00:00</td>
<td>10.16%</td>
<td>2.3195</td>
<td>0.0100</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>00:00:05</td>
<td>50.78%</td>
<td>1.7102</td>
<td>0.0100</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>00:00:10</td>
<td>63.28%</td>
<td>1.1632</td>
<td>0.0100</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>00:00:15</td>
<td>60.16%</td>
<td>1.0859</td>
<td>0.0100</td>
</tr>
<tr>
<td>6</td>
<td>200</td>
<td>00:00:21</td>
<td>68.75%</td>
<td>0.8996</td>
<td>0.0100</td>
</tr>
<tr>
<td>7</td>
<td>250</td>
<td>00:00:26</td>
<td>76.56%</td>
<td>0.7919</td>
<td>0.0100</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
<td>00:00:31</td>
<td>73.44%</td>
<td>0.8411</td>
<td>0.0100</td>
</tr>
<tr>
<td>9</td>
<td>350</td>
<td>00:00:37</td>
<td>81.25%</td>
<td>0.5514</td>
<td>0.0100</td>
</tr>
<tr>
<td>11</td>
<td>400</td>
<td>00:00:42</td>
<td>90.63%</td>
<td>0.4744</td>
<td>0.0100</td>
</tr>
<tr>
<td>12</td>
<td>450</td>
<td>00:00:47</td>
<td>92.19%</td>
<td>0.3614</td>
<td>0.0100</td>
</tr>
<tr>
<td>13</td>
<td>500</td>
<td>00:00:53</td>
<td>94.53%</td>
<td>0.3159</td>
<td>0.0100</td>
</tr>
<tr>
<td>15</td>
<td>550</td>
<td>00:00:58</td>
<td>96.09%</td>
<td>0.2543</td>
<td>0.0100</td>
</tr>
<tr>
<td>16</td>
<td>600</td>
<td>00:01:03</td>
<td>92.19%</td>
<td>0.2765</td>
<td>0.0100</td>
</tr>
<tr>
<td>17</td>
<td>650</td>
<td>00:01:09</td>
<td>95.31%</td>
<td>0.2461</td>
<td>0.0100</td>
</tr>
<tr>
<td>18</td>
<td>700</td>
<td>00:01:14</td>
<td>99.22%</td>
<td>0.1418</td>
<td>0.0100</td>
</tr>
<tr>
<td>20</td>
<td>750</td>
<td>00:01:19</td>
<td>98.44%</td>
<td>0.1000</td>
<td>0.0100</td>
</tr>
<tr>
<td>21</td>
<td>800</td>
<td>00:01:24</td>
<td>98.44%</td>
<td>0.1448</td>
<td>0.0100</td>
</tr>
<tr>
<td>22</td>
<td>850</td>
<td>00:01:30</td>
<td>98.44%</td>
<td>0.0989</td>
<td>0.0100</td>
</tr>
<tr>
<td>24</td>
<td>900</td>
<td>00:01:35</td>
<td>96.88%</td>
<td>0.1316</td>
<td>0.0100</td>
</tr>
<tr>
<td>25</td>
<td>950</td>
<td>00:01:41</td>
<td>100.00%</td>
<td>0.0859</td>
<td>0.0100</td>
</tr>
<tr>
<td>26</td>
<td>1000</td>
<td>00:01:46</td>
<td>100.00%</td>
<td>0.0761</td>
<td>0.0100</td>
</tr>
<tr>
<td>27</td>
<td>1050</td>
<td>00:01:51</td>
<td>100.00%</td>
<td>0.0759</td>
<td>0.0100</td>
</tr>
<tr>
<td>29</td>
<td>1100</td>
<td>00:01:57</td>
<td>99.22%</td>
<td>0.0663</td>
<td>0.0100</td>
</tr>
<tr>
<td>30</td>
<td>1150</td>
<td>00:02:02</td>
<td>98.44%</td>
<td>0.0775</td>
<td>0.0100</td>
</tr>
<tr>
<td>30</td>
<td>1170</td>
<td>00:02:04</td>
<td>99.22%</td>
<td>0.0732</td>
<td>0.0100</td>
</tr>
</tbody>
</table>

Run the trained network on a test set.

```matlab
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net,XTest);
```

Display the first 10 images in the test data and compare to the classification from `classify`.  

1-1216
The results from `classify` match the true digits for the first ten images.

Calculate the accuracy over all test data.

```matlab
accuracy = sum(YPred == YTest)/numel(YTest)
accuracy = 0.9820
```

### Classify Sequences Using a Trained LSTM Network

Load pretrained network. `JapaneseVowelsNet` is a pretrained LSTM network trained on the Japanese Vowels dataset as described in [1] and [2]. It was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```matlab
load JapaneseVowelsNet
```

View the network architecture.

```matlab
net.Layers
```

ans =

5x1 Layer array with layers:

1  sequenceinput  Sequence Input  Sequence input with 12 dimensions
2  lstm            LSTM           LSTM with 100 hidden units
3  fc              Fully Connected 9 fully connected layer
4  softmax         Softmax        softmax
5  classoutput     Classification Output crossentropyex with '1' and 8 other
Load the test data.
[XTest,YTest] = japaneseVowelsTestData;

Classify the test data.
YPred = classify(net,XTest);

View the labels of the first 10 sequences with their predicted labels.

[YTest(1:10) YPred(1:10)]

ans = 10x2 categorical array
     1     1
     1     1
     1     1
     1     1
     1     1
     1     1
     1     1
     1     1
     1     1

Calculate the classification accuracy of the predictions.

accuracy = sum(YPred == YTest)/numel(YTest)

accuracy = 0.8595

Input Arguments

net — Trained network
SeriesNetwork object | DAGNetwork object

Trained network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network (for example, by using the googlenet function) or by training your own network using trainNetwork.

imds — Image datastore
ImageDatastore object
Image datastore, specified as an `ImageDatastore` object.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then `ImageDatastore` does not prefetch.

**Tip** Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `ImageDatastore` as this option is usually significantly slower.

**ds — Datastore**

datastore

Datstore of input data.

For networks with a single input, `ds` can return either:

- a single image
- a cell array of images
- a table, where the first column contains images

For networks with multiple inputs, `ds` must be a combined or transformed datastore that returns a cell array with `numInputs` columns containing the input data, where `numInputs` is the number of network inputs. The i-th element of the cell array corresponds to the input `net.InputNames(i)`.

For more information, see “Datastores for Deep Learning”.

**X — Image data**

numeric array

Image data, specified as a numeric array. The size of the array depends on the type of image input:
### Input Description

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D images</td>
<td>A $h$-by-$w$-by-$c$-by-$N$ numeric array, where $h$, $w$, and $c$ are the height, width, and number of channels of the images, respectively, and $N$ is the number of images.</td>
</tr>
<tr>
<td>3-D images</td>
<td>A $h$-by-$w$-by-$d$-by-$c$-by-$N$ numeric array, where $h$, $w$, $d$, and $c$ are the height, width, depth, and number of channels of the images, respectively, and $N$ is the number of images.</td>
</tr>
</tbody>
</table>

If the array contains NaNs, then they are propagated through the network.

**sequences — Sequence or time series data**

*cell array of numeric arrays | numeric array | datastore*

Sequence or time series data, specified as an $N$-by-1 cell array of numeric arrays, where $N$ is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector sequences</td>
<td>$c$-by-$s$ matrices, where $c$ is the number of features of the sequences and $s$ is the sequence length.</td>
</tr>
<tr>
<td>2-D image sequences</td>
<td>$h$-by-$w$-by-$c$-by-$s$ arrays, where $h$, $w$, and $c$ correspond to the height, width, and number of channels of the images, respectively, and $s$ is the sequence length.</td>
</tr>
<tr>
<td>3-D image sequences</td>
<td>$h$-by-$w$-by-$d$-by-$c$-by-$s$, where $h$, $w$, $d$, and $c$ correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and $s$ is the sequence length.</td>
</tr>
</tbody>
</table>

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

**tbl — Table of images**

*table*
Table of images containing the input data in the first column. Each row in the table corresponds to an observation. The table contains absolute or relative file paths to an image, specified as a character vector or images specified as a numeric array.

Data Types: table

**Name-Value Pair Arguments**

Specify optional comma-separated pair of Name,Value argument. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' ').

Example: 'MiniBatchSize', '256' specifies the mini-batch size as 256.

**MiniBatchSize — Size of mini-batches**

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

Example: 'MiniBatchSize', 256

**Acceleration — Performance optimization**

'auto' (default) | 'mex' | 'none'

Performance optimization, specified as the comma-separated pair consisting of 'Acceleration' and one of the following:

- 'auto' — Automatically apply a number of optimizations suitable for the input network and hardware resource.
- 'mex' — Compile and execute a MEX function. This option is available when using a GPU only. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'none' — Disable all acceleration.
The default option is 'auto'. If 'auto' is specified, MATLAB will apply a number of compatible optimizations. If you use the 'auto' option, MATLAB does not ever generate a MEX function.

Using the 'Acceleration' options 'auto' and 'mex' can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The 'mex' option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The 'mex' option is only available when you are using a GPU. You must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning Libraries support package. Install the support package using the Add-On Explorer in MATLAB. For setup instructions, see “MEX Setup” (GPU Coder). GPU Coder is not required.

The 'mex' option does not support all layers. For a list of supported layers, see “Supported Layers” (GPU Coder). Recurrent neural networks (RNNs) containing a sequenceInputLayer are not supported.

You cannot use MATLAB Compiler to deploy your network when using the 'mex' option.

Example: 'Acceleration','mex'

**ExecutionEnvironment — Hardware resource**

'auto' (default) | 'gpu' | 'cpu'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
- 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'cpu' — Use the CPU.

Example: 'ExecutionEnvironment','cpu'
**SequenceLength** — Option to pad, truncate, or split input sequences

'longest' (default) | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: 'SequenceLength','shortest'

**SequencePaddingDirection** — Direction of padding or truncation

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer `OutputMode` property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the `OutputMode` property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the 'SequencePaddingDirection' option to 'right'.
To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingValue — Value to pad input sequences**
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

Example: 'SequencePaddingValue',-1

## Output Arguments

**YPred — Predicted class labels**
categorical vector | cell array of categorical vectors

Predicted class labels, returned as a categorical vector, or a cell array of categorical vectors. The format of `YPred` depends on the type of problem.

The following table describes the format for classification problems.

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>N-by-1 categorical vector of labels, where N is the number of observations.</td>
</tr>
<tr>
<td>Sequence-to-label classification</td>
<td></td>
</tr>
<tr>
<td>Sequence-to-sequence classification</td>
<td>N-by-1 cell array of categorical sequences of labels, where N is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after applying the <code>SequenceLength</code> option to each mini-batch independently.</td>
</tr>
</tbody>
</table>

For sequence-to-sequence classification problems with one observation, sequences can be a matrix. In this case, `YPred` is a categorical sequence of labels.

**scores — Predicted class scores**
matrix | cell array of matrices
Predicted scores or responses, returned as a matrix or a cell array of matrices. The format of scores depends on the type of problem.

The following table describes the format of scores.

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>$N$-by-$K$ matrix, where $N$ is the number of observations, and $K$ is the number of classes</td>
</tr>
<tr>
<td>Sequence-to-label classification</td>
<td>$N$-by-1 cell array of matrices, where $N$ is the number of observations. The sequences are matrices with $K$ rows, where $K$ is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the SequenceLength option to each mini-batch independently.</td>
</tr>
<tr>
<td>Sequence-to-sequence classification</td>
<td>$N$-by-1 cell array of matrices, where $N$ is the number of observations. The sequences are matrices with $K$ rows, where $K$ is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the SequenceLength option to each mini-batch independently.</td>
</tr>
</tbody>
</table>

For sequence-to-sequence classification problems with one observation, sequences can be a matrix. In this case, scores is a matrix of predicted class scores.

For an example exploring classification scores, see “Classify Webcam Images Using Deep Learning”.

**Algorithms**

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include trainNetwork, predict, classify, and activations. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

**Alternatives**

For networks with multiple outputs, use the predict and set the 'ReturnCategorical' option to true.

You can compute the predicted scores from a trained network using predict.
You can also compute the activations from a network layer using activations.

For sequence-to-label and sequence-to-sequence classification networks, you can make predictions and update the network state using classifyAndUpdateState and predictAndUpdateState.

References


See Also

activations | classifyAndUpdateState | predict | predictAndUpdateState

Topics

“Classify Image Using GoogLeNet”
“Classify Webcam Images Using Deep Learning”

Introduced in R2016a
predict

Predict responses using a trained deep learning neural network

Syntax

YPred = predict(net,imds)
YPred = predict(net,ds)
YPred = predict(net,X)
YPred = predict(net,tbl)

[YPred1,...,YPredM] = predict(___)

YPred = predict(net,sequences)

___ = predict(___ ,Name,Value)

Description

You can make predictions using a trained neural network for deep learning on either a
CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled
NVIDIA GPU with compute capability 3.0 or higher. Specify the hardware requirements
using the ExecutionEnvironment name-value pair argument.

YPred = predict(net,imds) predicts responses for the image data in imds using the
trained SeriesNetwork or DAGNetwork object net. For dlnetwork input, see predict.

YPred = predict(net,ds) predicts responses for the data in the datastore ds. For
networks with multiple inputs, use this syntax with a combined or transformed datastore
object.

YPred = predict(net,X) predicts responses for the data in the numeric array X.

YPred = predict(net,tbl) predicts responses for the data in the table tbl.

[YPred1,...,YPredM] = predict(____) predicts responses for the M outputs of a
multi-output network using any of the previous syntaxes. The output YPredj corresponds
to the network output `net.OutputNames(j)`. To return categorical outputs for the classification output layers, set the 'ReturnCategorical' option to `true`.

```matlab
YPred = predict(net,sequences)
```
predicts responses for the sequence or time series data in `sequences` using the trained LSTM network `net`.

```matlab
___ = predict(___,Name,Value)
```
predicts responses with additional options specified by one or more name-value pair arguments.

**Tip** When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

### Examples

**Predict Output Scores Using a Trained ConvNet**

Load the sample data.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
```

digitTrain4DArrayData loads the digit training set as 4-D array data. `XTrain` is a 28-by-28-by-1-by-5000 array, where 28 is the height and 28 is the width of the images. 1 is the number of channels and 5000 is the number of synthetic images of handwritten digits. `YTrain` is a categorical vector containing the labels for each observation.

Construct the convolutional neural network architecture.

```matlab
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the options to default settings for the stochastic gradient descent with momentum.
options = trainingOptions('sgdm');

Train the network.

rng('default')
net = trainNetwork(XTrain,YTrain,layers,options);

Training on single CPU.
Initializing input data normalization.

|========================================================================================|
|  Epoch |  Iteration |  Time Elapsed |  Mini-batch |  Mini-batch |  Base Learning |
|        |            |   (hh:mm:ss)  |   Accuracy  |     Loss    |      Rate      |
|========================================================================================|
|       1 |           1 |       00:00:01 |       10.16% |       2.3195 |          0.0100 |
|       2 |          50 |       00:00:09 |       50.78% |       1.7102 |          0.0100 |
|       3 |         100 |       00:00:18 |       63.28% |       1.1632 |          0.0100 |
|       4 |         150 |       00:00:25 |       60.16% |       1.0859 |          0.0100 |
|       6 |         200 |       00:00:34 |       68.75% |       0.8996 |          0.0100 |
|       7 |         250 |       00:00:44 |       76.56% |       0.7919 |          0.0100 |
|       8 |         300 |       00:00:53 |       73.44% |       0.8411 |          0.0100 |
|       9 |         350 |       00:01:14 |       81.25% |       0.5514 |          0.0100 |
|      11 |         400 |       00:01:24 |       90.63% |       0.4744 |          0.0100 |
|      12 |         450 |       00:01:34 |       92.19% |       0.3614 |          0.0100 |
|      13 |         500 |       00:01:43 |       94.53% |       0.3159 |          0.0100 |
|      15 |         550 |       00:01:53 |       96.09% |       0.2543 |          0.0100 |
|      16 |         600 |       00:02:02 |       92.19% |       0.2765 |          0.0100 |
|      17 |         650 |       00:02:12 |       95.31% |       0.2461 |          0.0100 |
|      18 |         700 |       00:02:22 |       99.22% |       0.1418 |          0.0100 |
|      20 |         750 |       00:02:32 |       98.44% |       0.1000 |          0.0100 |
|      21 |         800 |       00:02:41 |       98.44% |       0.1448 |          0.0100 |
|      22 |         850 |       00:02:51 |       98.44% |       0.0989 |          0.0100 |
|      24 |         900 |       00:03:01 |       96.88% |       0.1316 |          0.0100 |
|      25 |         950 |       00:03:11 |       96.88% |       0.1316 |          0.0100 |
|      26 |        1000 |       00:03:21 |       98.44% |       0.0859 |          0.0100 |
|      27 |        1050 |       00:03:31 |       100.00% |       0.0701 |          0.0100 |
|      29 |        1100 |       00:03:41 |       100.00% |       0.0759 |          0.0100 |
|      30 |        1150 |       00:03:51 |       100.00% |       0.0775 |          0.0100 |
|      30 |        1170 |       00:04:01 |       100.00% |       0.0732 |          0.0100 |

Training on single CPU.
Initializing input data normalization.

Run the trained network on a test set and predict the scores.

[XTest,YTest] = digitTest4DArrayData;
YPred = predict(net,XTest);
predict, by default, uses a CUDA® enabled GPU with compute capability 3.0, when available. You can also choose to run predict on a CPU using the 'ExecutionEnvironment','cpu' name-value pair argument.

Display the first 10 images in the test data and compare to the predictions from predict.

YTest(1:10,:) 
ans = 10x1 categorical array
0
0
0
0
0
0
0
0
0
0

YPred(1:10,:) 
ans = 10x10 single matrix
0.9978    0.0001    0.0008    0.0002    0.0003    0.0000    0.0004    0.0000    0.0000
0.8883    0.0000    0.0472    0.0001    0.0000    0.0002    0.0029    0.0001    0.0000
0.9998    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.9814    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0046
0.9748    0.0000    0.0132    0.0003    0.0000    0.0000    0.0000    0.0000    0.0002
0.9872    0.0000    0.0001    0.0000    0.0000    0.0000    0.0000    0.0007    0.0000
0.9981    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0018    0.0000
1.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.9270    0.0000    0.0046    0.0000    0.0006    0.0009    0.0001    0.0000    0.0000
0.9328    0.0000    0.0139    0.0012    0.0001    0.0001    0.0378    0.0000    0.0000

YTest contains the digits corresponding to the images in XTest. The columns of YPred contain predict’s estimation of a probability that an image contains a particular digit. That is, the first column contains the probability estimate that the given image is digit 0, the second column contains the probability estimate that the image is digit 1, the third column contains the probability estimate that the image is digit 2, and so on. You can see that predict’s estimation of probabilities for the correct digits are almost 1 and the probability for any other digit is almost 0. predict correctly estimates the first 10 observations as digit 0.
Predict Output Scores Using a Trained LSTM Network

Load pretrained network. JapaneseVowelsNet is a pretrained LSTM network trained on the Japanese Vowels dataset as described in [1] and [2]. It was trained on the sequences sorted by sequence length with a mini-batch size of 27.

load JapaneseVowelsNet

View the network architecture.

net.Layers

ans =
5x1 Layer array with layers:
1    'sequenceinput'    Sequence Input    Sequence input with 12 dimensions
2    'lstm'              LSTM              LSTM with 100 hidden units
3    'fc'                Fully Connected   9 fully connected layer
4    'softmax'           Softmax          softmax
5    'classoutput'       Classification Output  crossentropyex with '1' and 8 other classes

Load the test data.

[XTest,YTest] = japaneseVowelsTestData;

Make predictions on the test data.

YPred = predict(net,XTest);

View the prediction scores for the first 10 sequences.

YPred(1:10,:)

ans = 10x9 single matrix
0.9918 0.0000 0.0000 0.0000 0.0006 0.0010 0.0001 0.0006 0.0000
0.9868 0.0000 0.0000 0.0000 0.0006 0.0010 0.0001 0.0010 0.0000
0.9924 0.0000 0.0000 0.0000 0.0006 0.0010 0.0001 0.0006 0.0000
0.9896 0.0000 0.0000 0.0000 0.0006 0.0009 0.0001 0.0007 0.0000
0.9965 0.0000 0.0000 0.0000 0.0007 0.0009 0.0000 0.0003 0.0000
0.9888 0.0000 0.0000 0.0000 0.0006 0.0010 0.0001 0.0008 0.0000
0.9886 0.0000 0.0000 0.0000 0.0006 0.0010 0.0001 0.0008 0.0000
Compare these prediction scores to the labels of these sequences. The function assigns high prediction scores to the correct class.

YTest(1:10)

ans = 10x1 categorical array
1
1
1
1
1
1
1
1
1
1

Input Arguments

**net — Trained network**

SeriesNetwork object | DAGNetwork object

Trained network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network (for example, by using the googlenet function) or by training your own network using trainNetwork.

**imds — Image datastore**

ImageDatastore object

Image datastore, specified as an ImageDatastore object.

ImageDatastore allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, then ImageDatastore does not prefetch.
**Tip** Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing.

Do not use the `readFcn` option of `ImageDatastore` as this option is usually significantly slower.

**ds — Datastore**

datastore

Datastore of input data.

For networks with a single input, `ds` can return either:

- a single image
- a cell array of images
- a table, where the first column contains images

For networks with multiple inputs, `ds` must be a combined or transformed datastore that returns a cell array with `numInputs` columns containing the input data, where `numInputs` is the number of network inputs. The `i`th element of the cell array corresponds to the input `net.InputNames(i)`.

For more information, see “Datastores for Deep Learning”.

**X — Image data**

numeric array

Image data, specified as a numeric array. The size of the array depends on the type of image input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D images</td>
<td>A <code>h</code>-by-<code>w</code>-by-<code>c</code>-by-<code>N</code> numeric array, where <code>h</code>, <code>w</code>, and <code>c</code> are the height, width, and number of channels of the images, respectively, and <code>N</code> is the number of images.</td>
</tr>
<tr>
<td>3-D images</td>
<td>A <code>h</code>-by-<code>w</code>-by-<code>d</code>-by-<code>c</code>-by-<code>N</code> numeric array, where <code>h</code>, <code>w</code>, <code>d</code>, and <code>c</code> are the height, width, depth, and number of channels of the images, respectively, and <code>N</code> is the number of images.</td>
</tr>
</tbody>
</table>

If the array contains NaNs, then they are propagated through the network.
**sequences — Sequence or time series data**
cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an N-by-1 cell array of numeric arrays, where N is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector sequences</td>
<td>c-by-s matrices, where c is the number of features of the sequences and s is the sequence length.</td>
</tr>
<tr>
<td>2-D image sequences</td>
<td>h-by-w-by-c-by-s arrays, where h, w, and c correspond to the height, width, and number of channels of the images, respectively, and s is the sequence length.</td>
</tr>
<tr>
<td>3-D image sequences</td>
<td>h-by-w-by-d-by-c-by-s, where h, w, d, and c correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and s is the sequence length.</td>
</tr>
</tbody>
</table>

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

**tbl — Table of images**
table

Table of images containing the input data in the first column. Each row in the table corresponds to an observation. The table contains absolute or relative file paths to an image, specified as a character vector or images specified as a numeric array.

Data Types: table
Name-Value Pair Arguments

Specify optional comma-separated pair of Name, Value argument. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (').

Example: 'MiniBatchSize', 256 specifies the mini-batch size as 256.

MiniBatchSize — Size of mini-batches
128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger minibatch sizes require more memory, but can lead to faster predictions.

When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

Example: 'MiniBatchSize', 256

Acceleration — Performance optimization
'auto' (default) | 'mex' | 'none'

Performance optimization, specified as the comma-separated pair consisting of 'Acceleration' and one of the following:

- 'auto' — Automatically apply a number of optimizations suitable for the input network and hardware resource.
- 'mex' — Compile and execute a MEX function. This option is available when using a GPU only. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'none' — Disable all acceleration.

The default option is 'auto'. If 'auto' is specified, MATLAB will apply a number of compatible optimizations. If you use the 'auto' option, MATLAB does not ever generate a MEX function.

Using the 'Acceleration' options 'auto' and 'mex' can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible
parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

The 'mex' option generates and executes a MEX function based on the network and parameters used in the function call. You can have several MEX functions associated with a single network at one time. Clearing the network variable also clears any MEX functions associated with that network.

The 'mex' option is only available when you are using a GPU. You must have a C/C++ compiler installed and the GPU Coder Interface for Deep Learning Libraries support package. Install the support package using the Add-On Explorer in MATLAB. For setup instructions, see “MEX Setup” (GPU Coder). GPU Coder is not required.

The 'mex' option does not support all layers. For a list of supported layers, see “Supported Layers” (GPU Coder). Recurrent neural networks (RNNs) containing a sequenceInputLayer are not supported.

You cannot use MATLAB Compiler to deploy your network when using the 'mex' option.

Example: 'Acceleration','mex'

**ExecutionEnvironment — Hardware resource**

'auto' (default) | 'gpu' | 'cpu'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
- 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'cpu' — Use the CPU.

Example: 'ExecutionEnvironment','cpu'

**ReturnCategorical — Option to return categorical labels**

false (default) | true

Option to return categorical labels, specified as true or false.

If ReturnCategorical is true, then the function returns categorical labels for classification output layers. Otherwise, the function returns the prediction scores for classification output layers.
**SequenceLength — Option to pad, truncate, or split input sequences**

'longest' (default) | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- **'longest'** — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.

- **'shortest'** — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.

- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: 'SequenceLength','shortest'

**SequencePaddingDirection — Direction of padding or truncation**

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- **'right'** — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.

- **'left'** — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer **OutputMode** property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the **SequencePaddingDirection' option to 'left'**.

For sequence-to-sequence networks (when the **OutputMode** property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the **SequencePaddingDirection' option to 'right'**.
To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingValue — Value to pad input sequences**  
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

Example: 'SequencePaddingValue', -1

**Output Arguments**

**YPred — Predicted scores or responses**

matrix | 4-D numeric array | cell array of matrices

Predicted scores or responses, returned as a matrix, a 4-D numeric array, or a cell array of matrices. The format of `YPred` depends on the type of problem.

The following table describes the format for classification problems.

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image classification</td>
<td>N-by-K matrix, where N is the number of observations, and K is the number of classes</td>
</tr>
<tr>
<td>Sequence-to-label classification</td>
<td>N-by-1 cell array of matrices, where N is the number of observations. The sequences are matrices with K rows, where K is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the <code>SequenceLength</code> option to each mini-batch independently.</td>
</tr>
</tbody>
</table>

The following table describes the format for regression problems.
### Task | Format
--- | ---
2-D image regression | • $N$-by-$R$ matrix, where $N$ is the number of images and $R$ is the number of responses.
• $h$-by-$w$-by-$c$-by-$N$ numeric array, where $h$, $w$, and $c$ are the height, width, and number of channels of the images, respectively, and $N$ is the number of images.

3-D image regression | • $N$-by-$R$ matrix, where $N$ is the number of images and $R$ is the number of responses.
• $h$-by-$w$-by-$d$-by-$c$-by-$N$ numeric array, where $h$, $w$, $d$, and $c$ are the height, width, depth, and number of channels of the images, respectively, and $N$ is the number of images.

Sequence-to-one regression | $N$-by-$R$ matrix, where $N$ is the number of sequences and $R$ is the number of responses.

Sequence-to-sequence regression | $N$-by-1 cell array of numeric sequences, where $N$ is the number of sequences. The sequences are matrices with $R$ rows, where $R$ is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after applying the `SequenceLength` option to each mini-batch independently.

For sequence-to-sequence regression problems with one observation, sequences can be a matrix. In this case, `YPred` is a matrix of responses.

### Algorithms
If the image data contains NaNs, `predict` propagates them through the network. If the network has ReLU layers, these layers ignore NaNs. However, if the network does not have a ReLU layer, then `predict` returns NaNs as predictions.
All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

**Alternatives**

You can compute the predicted scores and the predicted classes from a trained network using `classify`.

You can also compute the activations from a network layer using `activations`.

For sequence-to-label and sequence-to-sequence classification networks (LSTM networks), you can make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.

**References**


**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the syntax `YPred = predict(net,X)` is supported.
- The input `X` must not have a variable size. The size must be fixed at code generation time.
For more information about generating code for deep learning neural networks, see “Workflow for Deep Learning Code Generation with MATLAB Coder” (MATLAB Coder).

**GPU Code Generation**  
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the syntax `YPred = predict(net,X)` is supported.
- The input `X` must not have a variable size. The size must be fixed at code generation time.

**See Also**  
activations | classify | classifyAndUpdateState | predictAndUpdateState

*Introduced in R2016a*
setLearnRateFactor

**Package:** nnet.cnn.layer

Set learn rate factor of layer learnable parameter

**Syntax**

```
layer = setLearnRateFactor(layer,parameterName,factor)
```

**Description**

```
layer = setLearnRateFactor(layer,parameterName,factor) sets the learn rate factor of the parameter with the name parameterName in layer to factor.
```

For built-in layers, you can set the learn rate factor directly by using the corresponding property. For example, for a `convolution2dLayer` layer, the syntax `layer = setLearnRateFactor(layer,'Weights',factor)` is equivalent to `layer.WeightLearnRateFactor = factor`.

**Examples**

**Set and Get Learning Rate Factor of Learnable Parameter**

Set and get the learning rate factor of a learnable parameter of a custom PReLU layer.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

Create a layer array including the custom layer `preluLayer`.

```
layers = [ ...  
    imageInputLayer([28 28 1])  
    convolution2dLayer(5,20)  
    batchNormalizationLayer
];
```
preluLayer(20,'prelu')
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer;

Set the learn rate factor of the 'Alpha' learnable parameter of the preluLayer to 2.

layers(4) = setLearnRateFactor(layers(4),'Alpha',2);

View the updated learn rate factor.

factor = getLearnRateFactor(layers(4),'Alpha')

factor = 2

**Input Arguments**

- **layer — Input layer**
  scalar Layer object

  Input layer, specified as a scalar Layer object.

- **parameterName — Parameter name**
  character vector | string scalar

  Parameter name, specified as a character vector or a string scalar.

- **factor — Learning rate factor**
  nonnegative scalar

  Learning rate factor for the parameter, specified as a nonnegative scalar.

  The software multiplies this factor by the global learning rate to determine the learning rate for the specified parameter. For example, if `factor` is 2, then the learning rate for the specified parameter is twice the current global learning rate. The software determines the global learning rate based on the settings specified with the `trainingOptions` function.

  Example: 2
See Also
getL2Factor | getLearnRateFactor | setL2Factor | trainNetwork | trainingOptions

Topics
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

Introduced in R2017b
getLearnRateFactor

Package: nnet.cnn.layer

Get learn rate factor of layer learnable parameter

Syntax

factor = getLearnRateFactor(layer,parameterName)

Description

factor = getLearnRateFactor(layer,parameterName) returns the learn rate factor of the parameter with the name parameterName in layer.

For built-in layers, you can get the learn rate factor directly by using the corresponding property. For example, for a convolution2dLayer layer, the syntax factor = getLearnRateFactor(layer,'Weights',factor) is equivalent to factor = layer.WeightLearnRateFactor.

Examples

Set and Get Learning Rate Factor of Learnable Parameter

Set and get the learning rate factor of a learnable parameter of a custom PReLU layer.

Define a custom PReLU layer. To create this layer, save the file preluLayer.m in the current folder.

Create a layer array including the custom layer preluLayer.

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer]
preluLayer(20,'prelu')
fullyConnectedLayer(10)
softmaxLayer
classificationLayer;

Set the learn rate factor of the 'Alpha' learnable parameter of the preluLayer to 2.

layers(4) = setLearnRateFactor(layers(4), 'Alpha', 2);

View the updated learn rate factor.

factor = getLearnRateFactor(layers(4), 'Alpha')

factor = 2

### Input Arguments

**layer — Input layer**

*scalar Layer object*

Input layer, specified as a scalar Layer object.

**parameterName — Parameter name**

*character vector | string scalar*

Parameter name, specified as a character vector or a string scalar.

### See Also

getL2Factor | setL2Factor | setLearnRateFactor | trainNetwork | trainingOptions

### Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

### Introduced in R2017b
setL2Factor

**Package:** nnet.cnn.layer

Set L2 regularization factor of layer learnable parameter

**Syntax**

layer = setL2Factor(layer,parameterName,factor)

**Description**

layer = setL2Factor(layer,parameterName,factor) sets the L2 regularization factor of the parameter with the name parameterName in layer to factor.

For built-in layers, you can set the L2 regularization factor directly by using the corresponding property. For example, for a convolution2dLayer layer, the syntax layer = setL2Factor(layer,'Weights',factor) is equivalent to layer.WeightL2Factor = factor.

**Examples**

**Set and Get L2 Regularization Factor of Learnable Parameter**

Set and get the L2 regularization factor of a learnable parameter of a layer.

Define a custom PReLU layer. To create this layer, save the file preluLayer.m in the current folder.

Create a layer array including a custom layer preluLayer.

layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer]
preluLayer(20,'prelu')
fullyConnectedLayer(10)
softmaxLayer
classificationLayer];

Set the L2 regularization factor of the 'Alpha' learnable parameter of the preluLayer to 2.
layers(4) = setL2Factor(layers(4),'Alpha',2);

View the updated L2 regularization factor.
factor = getL2Factor(layers(4),'Alpha')
factor = 2

### Input Arguments

**layer — Input layer**
scalar Layer object

Input layer, specified as a scalar Layer object.

**parameterName — Parameter name**
character vector | string scalar

Parameter name, specified as a character vector or a string scalar.

**factor — L2 regularization factor**
nonnegative scalar

L2 regularization factor for the parameter, specified as a nonnegative scalar.

The software multiplies this factor with the global L2 regularization factor to determine the L2 regularization factor for the specified parameter. For example, if `factor` is 2, then the L2 regularization for the specified parameter is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2
See Also
getL2Factor | getLearnRateFactor | setLearnRateFactor | trainNetwork | trainingOptions

Topics
“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

Introduced in R2017b
getL2Factor

**Package:** nnet.cnn.layer

Get L2 regularization factor of layer learnable parameter

**Syntax**

```matlab
factor = getL2RateFactor(layer,parameterName)
```

**Description**

`factor = getL2RateFactor(layer,parameterName)` returns the L2 regularization factor of the parameter with the name `parameterName` in `layer`.

For built-in layers, you can get the L2 regularization factor directly by using the corresponding property. For example, for a `convolution2dLayer` layer, the syntax `factor = getL2Factor(layer,'Weights',factor)` is equivalent to `factor = layer.WeightL2Factor`.

**Examples**

**Set and Get L2 Regularization Factor of Learnable Parameter**

Set and get the L2 regularization factor of a learnable parameter of a layer.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

Create a layer array including a custom layer `preluLayer`.

```matlab
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    ];
```
preluLayer(20,'prelu')
fullyConnectedLayer(10)
softmaxLayer
classificationLayer;

Set the L2 regularization factor of the 'Alpha' learnable parameter of the preluLayer to 2.

layers(4) = setL2Factor(layers(4),'Alpha',2);

View the updated L2 regularization factor.

factor = getL2Factor(layers(4),'Alpha')

factor = 2

**Input Arguments**

*layer — Input layer*
scalar Layer object

Input layer, specified as a scalar Layer object.

*parameterName — Parameter name*

character vector | string scalar

Parameter name, specified as a character vector or a string scalar.

**See Also**

getLearnRateFactor | setL2Factor | setLearnRateFactor | trainNetwork | trainingOptions

**Topics**

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Define Custom Deep Learning Layers”

**Introduced in R2017b**
**classifyAndUpdateState**

Classify data using a trained recurrent neural network and update the network state.

**Syntax**

```
[updatedNet,YPred] = classifyAndUpdateState(recNet,sequences)
[updatedNet,YPred] = classifyAndUpdateState(___ ,Name,Value)
[updatedNet,YPred,scores] = classifyAndUpdateState(____)
```

**Description**

You can make predictions using a trained deep learning network on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. Specify the hardware requirements using the “ExecutionEnvironment” on page 1-0 name-value pair argument.

```
[updatedNet,YPred] = classifyAndUpdateState(recNet,sequences) classifies the data in sequences using the trained recurrent neural network recNet and updates the network state.
```

This function supports recurrent neural networks only. The input recNet must have at least one recurrent layer.

```
[updatedNet,YPred] = classifyAndUpdateState(___ ,Name,Value) uses any of the arguments in the previous syntaxes and additional options specified by one or more Name,Value pair arguments. For example, 'MiniBatchSize',27 classifies data using mini-batches of size 27
```

“Classify and Update Network State” on page 1-1253

```
[updatedNet,YPred,scores] = classifyAndUpdateState(____) uses any of the arguments in the previous syntaxes, returns a matrix of classification scores, and updates the network state.
```

**Tip** When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different
predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

Examples

Classify and Update Network State

Classify data using a recurrent neural network and update the network state.

Load JapaneseVowelsNet, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

load JapaneseVowelsNet

View the network architecture.

net.Layers

ans =
  5x1 Layer array with layers:

  1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
  2   'lstm'            LSTM                    LSTM with 100 hidden units
  3   'fc'              Fully Connected         9 fully connected layer
  4   'softmax'         Softmax                 softmax
  5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other

Load the test data.

[XTest,YTest] = japaneseVowelsTestData;

Loop over the time steps in a sequence. Classify each time step and update the network state.

X = XTest{94};
numTimeSteps = size(X,2);
for i = 1:numTimeSteps
    v = X(:,i);
    [net,label,score] = classifyAndUpdateState(net,v);
labels(i) = label;
end

Plot the predicted labels in a stair plot. The plot shows how the predictions change between time steps.

```matlab
figure
stairs(labels, '-o')
xlim([1 numTimeSteps])
xlabel("Time Step")
ylabel("Predicted Class")
title("Classification Over Time Steps")
```

![Classification Over Time Steps](image-url)
Compare the predictions with the true label. Plot a horizontal line showing the true label of the observation.

```matlab
tureLabel = YTest(94)
tureLabel = categorical
3

hold on
line([1 numTimeSteps],[trueLabel trueLabel], ...
    'Color','red', ...
    'LineStyle','--')
legend(["Prediction" "True Label")
```
**Input Arguments**

**recNet — Trained recurrent neural network**  
SeriesNetwork object | DAGNetwork object

Trained recurrent neural network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function.

`recNet` is a recurrent neural network. It must have at least one recurrent layer (for example, an LSTM network).

**sequences — Sequence or time series data**  
cell array of numeric arrays | numeric array | datastore

Sequence or time series data, specified as an `N`-by-1 cell array of numeric arrays, where `N` is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector sequences</td>
<td><code>c</code>-by-<code>s</code> matrices, where <code>c</code> is the number of features of the sequences and <code>s</code> is the sequence length.</td>
</tr>
<tr>
<td>2-D image sequences</td>
<td><code>h</code>-by-<code>w</code>-by-<code>c</code>-by-<code>s</code> arrays, where <code>h</code>, <code>w</code>, and <code>c</code> correspond to the height, width, and number of channels of the images, respectively, and <code>s</code> is the sequence length.</td>
</tr>
<tr>
<td>3-D image sequences</td>
<td><code>h</code>-by-<code>w</code>-by-<code>d</code>-by-<code>c</code>-by-<code>s</code>, where <code>h</code>, <code>w</code>, <code>d</code>, and <code>c</code> correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and <code>s</code> is the sequence length.</td>
</tr>
</tbody>
</table>

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.
**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.


**MiniBatchSize — Size of mini-batches**

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

Example: `'MiniBatchSize', 256`

**Acceleration — Performance optimization**

'auto' (default) | 'none'

Performance optimization, specified as the comma-separated pair consisting of 'Acceleration' and one of the following:

- 'auto' — Automatically apply a number of optimizations suitable for the input network and hardware resource.
- 'none' — Disable all acceleration.

The default option is 'auto'.

Using the 'Acceleration' option 'auto' can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

Example: `'Acceleration', 'auto'`
**ExecutionEnvironment — Hardware resource**

'auto' (default) | 'gpu' | 'cpu'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
- 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'cpu' — Use the CPU.

Example: 'ExecutionEnvironment','cpu'

**SequenceLength — Option to pad, truncate, or split input sequences**

'longest' (default) | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: 'SequenceLength','shortest'

**SequencePaddingDirection — Direction of padding or truncation**

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:
• 'right' — Pad or truncate sequences on the right. The sequences start at the same
time step and the software truncates or adds padding to the end of the sequences.
• 'left' — Pad or truncate sequences on the left. The software truncates or adds
padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer
OutputMode property is 'last', any padding in the final time steps can negatively
influence the layer output. To pad or truncate sequence data on the left, set the
'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the OutputMode property is 'sequence' for
each LSTM layer), any padding in the first time steps can negatively influence the
predictions for the earlier time steps. To pad or truncate sequence data on the right, set
the 'SequencePaddingDirection' option to 'right'.

To learn more about the effect of padding, truncating, and splitting the input sequences,
see “Sequence Padding, Truncation, and Splitting”.

SequencePaddingValue — Value to pad input sequences
0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when
SequenceLength is 'longest' or a positive integer. Do not pad sequences with NaN,
because doing so can propagate errors throughout the network.

Example: 'SequencePaddingValue','-1

Output Arguments

updatedNet — Updated network
SeriesNetwork object | DAGNetwork object

Updated network. updatedNet is the same type of network as the input network.

YPred — Predicted class labels
categorical vector | cell array of categorical vectors

Predicted class labels, returned as a categorical vector, or a cell array of categorical
vectors. The format of YPred depends on the type of problem.

The following table describes the format of YPred.
### Task | Format
--- | ---
Sequence-to-label classification | N-by-1 categorical vector of labels, where N is the number of observations.
Sequence-to-sequence classification | N-by-1 cell array of categorical sequences of labels, where N is the number of observations. Each sequence has the same number of time steps as the corresponding input sequence after applying the SequenceLength option.

For sequence-to-sequence classification problems with one observation, sequences can be a matrix. In this case, YPred is a categorical sequence of labels.

**scores — Predicted class scores**
matrix | cell array of matrices

Predicted class scores, returned as a matrix or a cell array of matrices. The format of scores depends on the type of problem.

The following table describes the format of scores.

### Task | Format
--- | ---
Sequence-to-label classification | N-by-K matrix, where N is the number of observations, and K is the number of classes.
Sequence-to-sequence classification | N-by-1 cell array of matrices, where N is the number of observations. The sequences are matrices with K rows, where K is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the SequenceLength option to each mini-batch independently.

For sequence-to-sequence classification problems with one observation, sequences can be a matrix. In this case, scores is a matrix of predicted class scores.
Algorithms

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include trainNetwork, predict, classify, and activations. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

References


See Also

bilstmLayer | classify | lstmLayer | predict | predictAndUpdateState | resetState | sequenceInputLayer

Topics

“Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”

Introduced in R2017b
predictAndUpdateState

Predict responses using a trained recurrent neural network and update the network state

Syntax

[updatedNet,YPred] = predictAndUpdateState(recNet,sequences)
[updatedNet,YPred] = predictAndUpdateState(____,Name,Value)

Description

You can make predictions using a trained deep learning network on either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. Specify the hardware requirements using the “'ExecutionEnvironment'” on page 1-0 name-value pair argument.

[updatedNet,YPred] = predictAndUpdateState(recNet,sequences) predicts responses for data in sequences using the trained recurrent neural network recNet and updates the network state.

This function supports recurrent neural networks only. The input recNet must have at least one recurrent layer.

[updatedNet,YPred] = predictAndUpdateState(____,Name,Value) uses any of the arguments in the previous syntaxes and additional options specified by one or more Name,Value pair arguments. For example, 'MiniBatchSize',27 makes predictions using mini-batches of size 27.

Tip When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.
Examples

Predict and Update Network State

Predict responses using a trained recurrent neural network and update the network state. 

Load JapaneseVowelsNet, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

load JapaneseVowelsNet

View the network architecture.

net.Layers

ans =
4x1 Layer array with layers:
 1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
 2   'lstm'            LSTM                    LSTM with 100 hidden units
 3   'fc'              Fully Connected         9 fully connected layer
 4   'softmax'         Softmax                 softmax
 5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes

Load the test data.

[XTest,YTest] = japaneseVowelsTestData;

Loop over the time steps in a sequence. Predict the scores of each time step and update the network state.

X = XTest{94};
numTimeSteps = size(X,2);
for i = 1:numTimeSteps
    v = X(:,i);
    [net,score] = predictAndUpdateState(net,v);
    scores(:,i) = score;
end

Plot the prediction scores. The plot shows how the prediction scores change between time steps.
classNames = string(net.Layers(end).Classes);
figure
lines = plot(scores');
xlim([1 numTimeSteps])
legend("Class " + classNames,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
title("Prediction Scores Over Time Steps")

Highlight the prediction scores over time steps for the correct class.

trueLabel = YTest(94)
trueLabel = categorical
3

lines(trueLabel).LineWidth = 3;
Display the final time step prediction in a bar chart.

```matlab
figure
bar(score)
title("Final Prediction Scores")
xlabel("Class")
ylabel("Score")
```
Input Arguments

**recNet — Trained recurrent neural network**

SeriesNetwork object | DAGNetwork object

Trained recurrent neural network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network or by training your own network using the `trainNetwork` function.

`recNet` is a recurrent neural network. It must have at least one recurrent layer (for example, an LSTM network).
sequences — Sequence or time series data

Sequence or time series data, specified as an \(N\)-by-1 cell array of numeric arrays, where \(N\) is the number of observations, a numeric array representing a single sequence, or a datastore.

For cell array or numeric array input, the dimensions of the numeric arrays containing the sequences depend on the type of data.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector sequences</td>
<td>(c)-by-(s) matrices, where (c) is the number of features of the sequences and (s) is the sequence length.</td>
</tr>
<tr>
<td>2-D image sequences</td>
<td>(h)-by-(w)-by-(c)-by-(s) arrays, where (h), (w), and (c) correspond to the height, width, and number of channels of the images, respectively, and (s) is the sequence length.</td>
</tr>
<tr>
<td>3-D image sequences</td>
<td>(h)-by-(w)-by-(d)-by-(c)-by-(s), where (h), (w), (d), and (c) correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and (s) is the sequence length.</td>
</tr>
</tbody>
</table>

For datastore input, the datastore must return data as a cell array of sequences or a table whose first column contains sequences. The dimensions of the sequence data must correspond to the table above.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1,** **Value1,** ..., **NameN,** **ValueN**.

Example: `[updatedNet, YPred] = predictAndUpdateState(recNet, C,'MiniBatchSize',27)` makes predictions using mini-batches of size 27.

**MiniBatchSize — Size of mini-batches**

128 (default) | positive integer
Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network.

To specify mini-batch size and padding options, use the 'MiniBatchSize' and 'SequenceLength' options.

Example: 'MiniBatchSize',256

**Acceleration — Performance optimization**

'auto' (default) | 'none'

Performance optimization, specified as the comma-separated pair consisting of 'Acceleration' and one of the following:

- 'auto' — Automatically apply a number of optimizations suitable for the input network and hardware resource.
- 'none' — Disable all acceleration.

The default option is 'auto'.

Using the 'Acceleration' option 'auto' can offer performance benefits, but at the expense of an increased initial run time. Subsequent calls with compatible parameters are faster. Use performance optimization when you plan to call the function multiple times using new input data.

Example: 'Acceleration','auto'

**ExecutionEnvironment — Hardware resource**

'auto' (default) | 'gpu' | 'cpu'

Hardware resource, specified as the comma-separated pair consisting of 'ExecutionEnvironment' and one of the following:

- 'auto' — Use a GPU if one is available; otherwise, use the CPU.
- 'gpu' — Use the GPU. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. If Parallel Computing Toolbox or a suitable GPU is not available, then the software returns an error.
- 'cpu' — Use the CPU.

Example: 'ExecutionEnvironment','cpu'
SequenceLength — Option to pad, truncate, or split input sequences

'longest' (default) | 'shortest' | positive integer

Option to pad, truncate, or split input sequences, specified as one of the following:

- 'longest' — Pad sequences in each mini-batch to have the same length as the longest sequence. This option does not discard any data, though padding can introduce noise to the network.
- 'shortest' — Truncate sequences in each mini-batch to have the same length as the shortest sequence. This option ensures that no padding is added, at the cost of discarding data.
- Positive integer — For each mini-batch, pad the sequences to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch, and then split the sequences into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches. Use this option if the full sequences do not fit in memory. Alternatively, try reducing the number of sequences per mini-batch by setting the 'MiniBatchSize' option to a lower value.

To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

Example: 'SequenceLength','shortest'

SequencePaddingDirection — Direction of padding or truncation

'right' (default) | 'left'

Direction of padding or truncation, specified as one of the following:

- 'right' — Pad or truncate sequences on the right. The sequences start at the same time step and the software truncates or adds padding to the end of the sequences.
- 'left' — Pad or truncate sequences on the left. The software truncates or adds padding to the start of the sequences so that the sequences end at the same time step.

Because LSTM layers process sequence data one time step at a time, when the layer OutputMode property is 'last', any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the 'SequencePaddingDirection' option to 'left'.

For sequence-to-sequence networks (when the OutputMode property is 'sequence' for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the 'SequencePaddingDirection' option to 'right'.

1-1269
To learn more about the effect of padding, truncating, and splitting the input sequences, see “Sequence Padding, Truncation, and Splitting”.

**SequencePaddingValue — Value to pad input sequences**

0 (default) | scalar

Value by which to pad input sequences, specified as a scalar. The option is valid only when `SequenceLength` is 'longest' or a positive integer. Do not pad sequences with NaN, because doing so can propagate errors throughout the network.

Example: 'SequencePaddingValue', -1

## Output Arguments

**updatedNet — Updated network**

SeriesNetwork object | DAGNetwork object

Updated network. `updatedNet` is the same type of network as the input network.

**YPred — Predicted scores or responses**

matrix | cell array of matrices

Predicted scores or responses, returned as a matrix or a cell array of matrices. The format of `YPred` depends on the type of problem.

The following table describes the format for classification problems.

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence-to-label classification</td>
<td>N-by-K matrix, where N is the number of observations, and K is the number of classes.</td>
</tr>
<tr>
<td>Sequence-to-sequence classification</td>
<td>N-by-1 cell array of matrices, where N is the number of observations. The sequences are matrices with K rows, where K is the number of classes. Each sequence has the same number of time steps as the corresponding input sequence after applying the <code>SequenceLength</code> option to each mini-batch independently.</td>
</tr>
</tbody>
</table>
For sequence-to-sequence classification problems with one observation, sequences can be a matrix. In this case,YPred is a \( K \)-by-\( S \) matrix of scores, where \( K \) is the number of classes, and \( S \) is the total number of time steps in the corresponding input sequence.

The following table describes the format for regression problems.

<table>
<thead>
<tr>
<th>Task</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence-to-one regression</td>
<td>( N )-by-( R ) matrix, where ( N ) is the number of observations and ( R ) is the number of responses.</td>
</tr>
<tr>
<td>Sequence-to-sequence regression</td>
<td>( N )-by-1 cell array of numeric sequences, where ( N ) is the number of observations. The sequences are matrices with ( R ) rows, where ( R ) is the number of responses. Each sequence has the same number of time steps as the corresponding input sequence after applying the SequenceLength option to each mini-batch independently.</td>
</tr>
</tbody>
</table>

For sequence-to-sequence problems with one observation, sequences can be a matrix. In this case, YPred is a matrix of responses.

**Algorithms**

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include trainNetwork, predict, classify, and activations. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

**References**


See Also
bilstmLayer | classify | classifyAndUpdateState | lstmLayer | predict | resetState | sequenceInputLayer

Topics
“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“Sequence-to-Sequence Regression Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Deep Learning in MATLAB”

Introduced in R2017b
**resetState**

Reset the state of a recurrent neural network

**Syntax**

updatedNet = resetState(recNet)

**Description**

updatedNet = resetState(recNet) resets the state of a recurrent neural network (for example, an LSTM network) to the initial state.

**Examples**

**Reset Network State**

Reset the network state between sequence predictions.

Load `JapaneseVowelsNet`, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

load JapaneseVowelsNet

View the network architecture.

net.Layers

ans =

5x1 Layer array with layers:

1    'sequenceinput'    Sequence Input    Sequence input with 12 dimensions
2    'lstm'              LSTM            LSTM with 100 hidden units
3    'fc'                Fully Connected  9 fully connected layer
Load the test data.

[XTest, YTest] = japaneseVowelsTestData;

Classify a sequence and update the network state. For reproducibility, set rng to 'shuffle'.

rng('shuffle')
X = XTest{94};
[net, label] = classifyAndUpdateState(net, X);
label

label = categorical
  3

Classify another sequence using the updated network.

X = XTest{1};
label = classify(net, X)

label = categorical
  7

Compare the final prediction with the true label.

trueLabel = YTest(1)
trueLabel = categorical
  1

The updated state of the network may have negatively influenced the classification. Reset the network state and predict on the sequence again.

net = resetState(net);
label = classify(net, XTest{1})

label = categorical
  1
Input Arguments

recNet — Trained recurrent neural network
SeriesNetwork object | DAGNetwork object

Trained recurrent neural network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network or by training your own network using the trainNetwork function.

recNet is a recurrent neural network. It must have at least one recurrent layer (for example, an LSTM network).

Output Arguments

updatedNet — Updated network
SeriesNetwork object | DAGNetwork object

Updated network. updatedNet is the same type of network as the input network.

References


See Also

bilstmLayer | classifyAndUpdateState | lstmLayer | predictAndUpdateState | sequenceInputLayer

Topics

“Sequence Classification Using Deep Learning”
“Visualize Activations of LSTM Network”
“Long Short-Term Memory Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”
“Deep Learning in MATLAB”

**Introduced in R2017b**
addLayers

Add layers to layer graph

Syntax

newlgraph = addLayers(lgraph,larray)

Description

newlgraph = addLayers(lgraph,larray) adds the network layers in larray to the layer graph lgraph. The new layer graph, newlgraph, contains the layers and connections of lgraph together with the layers in larray, connected sequentially. The layer names in larray must be unique, nonempty, and different from the names of the layers in lgraph.

Examples

Add Layers to Layer Graph

Create an empty layer graph and an array of layers. Add the layers to the layer graph and plot the graph. addLayers connects the layers sequentially.

lgraph = layerGraph;

layers = [imageInputLayer([32 32 3],'Name','input')
          convolution2dLayer(3,16,'Padding','same','Name','conv_1')
          batchNormalizationLayer('Name','BN_1')
          reluLayer('Name','relu_1')];

lgraph = addLayers(lgraph,layers);
figure
plot(lgraph)
### Input Arguments

**lgraph — Layer graph**

LayerGraph object

Layer graph, specified as a LayerGraph object. To create a layer graph, use `layerGraph`.

**larray — Network layers**

Layer array

Network layers, specified as a Layer array.
For a list of built-in layers, see “List of Deep Learning Layers”.

Output Arguments

newlgraph — Output layer graph
LayerGraph object

Output layer graph, returned as a LayerGraph object.

See Also

assembleNetwork | connectLayers | disconnectLayers | layerGraph | plot | removeLayers | replaceLayer

Topics

“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b
removeLayers

Remove layers from layer graph

Syntax

newlgraph = removeLayers(lgraph,layerNames)

Description

newlgraph = removeLayers(lgraph,layerNames) removes the layers specified by layerNames from the layer graph lgraph. The function also removes any connections to the removed layers.

Examples

Remove Layer from Layer Graph

Create a layer graph from an array of layers.

```
layers = [imageInputLayer([28 28 1], 'Name','input')
          convolution2dLayer(3,16,'Padding','same', 'Name','conv_1')
          batchNormalizationLayer('Name','BN_1')
          reluLayer('Name','relu_1')]
lg = layerGraph(layers);
figure
plot(lg)
```
Remove the 'BN_1' layer and its connections.

lgraph = removeLayers(lgraph,'BN_1');
figure
plot(lgraph)
Input Arguments

lgraph — Layer graph
LayerGraph object

Layer graph, specified as a LayerGraph object. To create a layer graph, use layerGraph.

layerNames — Names of layers to remove
character vector | cell array of character vectors | string array
Names of layers to remove, specified as a character vector, a cell array of character vectors, or a string array.

To remove a single layer from the layer graph, specify the name of the layer.

To remove multiple layers, specify the layer names in an array, where each element of the array is a layer name.

Example: 'conv1'
Example: {'conv1','add1'}

Output Arguments

newlgraph — Output layer graph
LayerGraph object

Output layer graph, returned as a LayerGraph object.

See Also

addLayers | assembleNetwork | connectLayers | disconnectLayers | layerGraph | plot | replaceLayer

Topics

“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b
connectLayers
Connect layers in layer graph

Syntax
newlgraph = connectLayers(lgraph,s,d)

Description
newlgraph = connectLayers(lgraph,s,d) connects the source layer s to the
destination layer d in the layer graph lgraph. The new layer graph, newlgraph, contains
the same layers as lgraph and includes the new connection.

Examples

Create and Connect Addition Layer
Create an addition layer with two inputs and the name 'add_1'.
add = additionLayer(2,'Name','add_1')
add =
    AdditionLayer with properties:
        Name: 'add_1'
        NumInputs: 2
        InputNames: {'in1' 'in2'}

Create two ReLU layers and connect them to the addition layer. The addition layer sums
the outputs from the ReLU layers.
relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');
lgraph = layerGraph;
lgraph = addLayers(lgraph, relu_1);
lgraph = addLayers(lgraph, relu_2);
lgraph = addLayers(lgraph, add);

lgraph = connectLayers(lgraph, 'relu_1', 'add_1/in1');
lgraph = connectLayers(lgraph, 'relu_2', 'add_1/in2');

plot(lgraph)
Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. All layers must have names and all names must be unique.

```matlab
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(5,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')
    convolution2dLayer(3,32,'Padding','same','Stride',2,'Name','conv_2')
    batchNormalizationLayer('Name','BN_2')
    reluLayer('Name','relu_2')
    convolution2dLayer(3,32,'Padding','same','Name','conv_3')
    batchNormalizationLayer('Name','BN_3')
    reluLayer('Name','relu_3')
    additionLayer(2,'Name','add')
    averagePooling2dLayer(2,'Stride',2,'Name','avpool')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classOutput')
];
```

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```matlab
lgraph = layerGraph(layers);
figure
plot(lgraph)
```
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the 'relu_3' layer. This arrangement enables the addition layer to add the outputs of the 'skipConv' and 'relu_3' layers. To check that the layer is in the graph, plot the layer graph.

```matlab
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv'); lgraph = addLayers(lgraph,skipConv); figure plot(lgraph)
```
Create the shortcut connection from the 'relu_1' layer to the 'add' layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named 'in1' and 'in2'. The 'relu_3' layer is already connected to the 'in1' input. Connect the 'relu_1' layer to the 'skipConv' layer and the 'skipConv' layer to the 'in2' input of the 'add' layer. The addition layer now sums the outputs of the 'relu_3' and 'skipConv' layers. To check that the layers are connected correctly, plot the layer graph.

```matlab
lgraph = connectLayers(lgraph, 'relu_1', 'skipConv');
lgraph = connectLayers(lgraph, 'skipConv', 'add/in2');
figure
plot(lgraph);
```
Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;

Specify training options and train the network. trainNetwork validates the network using the validation data every ValidationFrequency iterations.

options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ... 
    'Shuffle','every-epoch', ... 
    'ValidationData',{XValidation,YValidation}, ... 
    'ValidationFrequency',30, ...
'Verbose',false, ...
'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);

Display the properties of the trained network. The network is a DAGNetwork object.
net
net = DAGNetwork with properties:
    Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]

Classify the validation images and calculate the accuracy. The network is very accurate.
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
Input Arguments

\( lgraph \) — Layer graph
LayerGraph object

Layer graph, specified as a LayerGraph object. To create a layer graph, use \( \text{layerGraph} \).

\( s \) — Connection source
character vector | string scalar

Connection source, specified as a character vector or a string scalar.

- If the source layer has a single output, then \( s \) is the name of the layer.
- If the source layer has multiple outputs, then \( s \) is the layer name followed by the character / and the name of the layer output: 'layerName/outputName'.

Example: 'conv1'
Example: 'mpool/indices'

\( d \) — Connection destination
character vector | string scalar

Connection destination, specified as a character vector or a string scalar.

- If the destination layer has a single input, then \( d \) is the name of the layer.
- If the destination layer has multiple inputs, then \( d \) is the layer name followed by the character / and the name of the layer input: 'layerName/inputName'.

Example: 'fc'
Example: 'addlayer1/in2'

Output Arguments

\( \text{newlgraph} \) — Output layer graph
LayerGraph object
Output layer graph, returned as a `LayerGraph` object.

See Also

adding | assembleNetwork | disconnectLayers | layerGraph | plot | removeLayers | replaceLayer

Topics

“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b
disconnectLayers

Disconnect layers in layer graph

Syntax

newlgraph = disconnectLayers(lgraph,s,d)

Description

newlgraph = disconnectLayers(lgraph,s,d) disconnects the source layer s from the destination layer d in the layer graph lgraph. The new layer graph, newlgraph, contains the same layers as lgraph, but excludes the connection between s and d.

Examples

Disconnect Layers in Layer Graph

Create a layer graph from an array of layers.

layers = [
    imageInputLayer([28 28 1], 'Name', 'input')
    convolution2dLayer(3,16, 'Padding', 'same', 'Name', 'conv_1')
    batchNormalizationLayer('Name', 'BN_1')
    reluLayer('Name', 'relu_1')];

lgraph = layerGraph(layers);
figure
plot(lgraph)
Disconnect the 'conv_1' layer from the 'BN_1' layer.

```matlab
lgraph = disconnectLayers(lgraph,'conv_1','BN_1');
figure
plot(lgraph)
```
Input Arguments

lgraph — Layer graph
LayerGraph object

Layer graph, specified as a LayerGraph object. To create a layer graph, use layerGraph.

s — Connection source
character vector | string scalar

Connection source, specified as a character vector or a string scalar.
• If the source layer has a single output, then s is the name of the layer.
• If the source layer has multiple outputs, then s is the layer name followed by the character / and the name of the layer output: 'layerName/outputName'.

Example: 'conv1'
Example: 'mpool/indices'

\[d\] — Connection destination
character vector | string scalar

Connection destination, specified as a character vector or a string scalar.
• If the destination layer has a single input, then d is the name of the layer.
• If the destination layer has multiple inputs, then d is the layer name followed by the character / and the name of the layer input: 'layerName/inputName'.

Example: 'fc'
Example: 'addlayer1/in2'

**Output Arguments**

newlgraph — Output layer graph
LayerGraph object

Output layer graph, returned as a LayerGraph object.

**See Also**
addLayers | assembleNetwork | connectLayers | layerGraph | plot | removeLayers | replaceLayer

**Topics**
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

**Introduced in R2017b**
plot

Plot neural network layer graph

Syntax

plot(lgraph)
plot(dagNet)

Description

plot(lgraph) plots a diagram of the layer graph lgraph. The plot function labels each layer by its name and displays all layer connections.

Tip To analyze the network architecture and create an interactive network visualization, use analyzeNetwork.

plot(dagNet) plots a diagram of the layers in the directed acyclic graph (DAG) network dagNet.

Examples

Plot Layer Graph

Create a layer graph from an array of layers. Connect the 'relu_1' layer to the 'add' layer.

layers = [
    imageInputLayer([32 32 3], 'Name', 'input')
    convolution2dLayer(3, 16, 'Padding', 'same', 'Name', 'conv_1')
    batchNormalizationLayer('Name', 'BN_1')
    reluLayer('Name', 'relu_1')
]
convolution2dLayer(3,16,'Padding','same','Stride',2,'Name','conv_2')
batchNormalizationLayer('Name','BN_2')
reluLayer('Name','relu_2')
additionLayer(2,'Name','add'))

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'relu_1','add/in2');

Plot the layer graph.
figure
plot(lgraph);
**Plot DAG Network**

Load a pretrained GoogLeNet convolutional neural network as a DAGNetwork object. If the Deep Learning Toolbox™ Model for GoogLeNet Network support package is not installed, then the software provides a download link.

```
net = googlenet

net = DAGNetwork with properties:

    Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
```

Plot the network.

```
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
plot(net)
```
**Input Arguments**

**lgraph — Layer graph**  
LayerGraph object

Layer graph, specified as a LayerGraph object. To create a layer graph, use `layerGraph`.

**dagNet — DAG network**  
DAGNetwork object

DAG network, specified as a DAGNetwork object.
See Also
addLayers | analyzeNetwork | connectLayers | disconnectLayers | layerGraph
| removeLayers | replaceLayer

Topics
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2017b
layerGraph

Graph of network layers for deep learning

Description

A layer graph specifies the architecture of a deep learning network with a more complex graph structure in which layers can have inputs from multiple layers and outputs to multiple layers. Networks with this structure are called directed acyclic graph (DAG) networks. After you create a layerGraph object, you can use the object functions to plot the graph and modify it by adding, removing, connecting, and disconnecting layers. To train the network, use the layer graph as the layers input argument to trainNetwork.

Creation

Syntax

lgraph = layerGraph
lgraph = layerGraph(layers)
lgraph = layerGraph(dagNet)
lgraph = layerGraph(dlnet)

Description

lgraph = layerGraph creates an empty layer graph that contains no layers. You can add layers to the empty graph by using the addLayers function.

lgraph = layerGraph(layers) creates a layer graph from an array of network layers and sets the Layers property. The layers in lgraph are connected in the same sequential order as in layers. All layers must have unique, nonempty names.

lgraph = layerGraph(dagNet) extracts the layer graph of a DAGNetwork. For example, you can extract the layer graph of a pretrained network to perform transfer learning.
layerGraph = layerGraph(dlnet) extracts the layer graph of a dlnetwork. Use this syntax to use a dlnetwork with the trainNetwork function or Deep Network Designer.

Input Arguments

dagNet — DAG network
DAGNetwork object

DAG network, specified as a DAGNetwork object.

dlnet — Network
dlnetwork object

Network for custom training loops, specified as a dlnetwork object.

For dlnetwork input, the software extracts the numeric data from the learnable parameters and converts it to single precision.

Properties

Layers — Network layers
Layer array

Network layers, specified as a Layer array.

Connections — Layer connections
table

Layer connections, specified as a table with two columns.

Each table row represents a connection in the layer graph. The first column, Source, specifies the source of each connection. The second column, Destination, specifies the destination of each connection. The connection sources and destinations are either layer names or have the form 'layerName/IOName', where 'IOName' is the name of the layer input or output.

Data Types: table

InputNames — Network input layer names
cell array
Network input layer names, specified as a cell array of character vectors.
Data Types: cell

OutputNames — Network output layer names

cell array

Network output layer names, specified as a cell array of character vectors.
Data Types: cell

Object Functions

addLayers Add layers to layer graph
removeLayers Remove layers from layer graph
replaceLayer Replace layer in layer graph
connectLayers Connect layers in layer graph
disconnectLayers Disconnect layers in layer graph
plot Plot neural network layer graph

Examples

Add Layers to Layer Graph

Create an empty layer graph and an array of layers. Add the layers to the layer graph and plot the graph. addLayers connects the layers sequentially.

lgraph = layerGraph;

layers = [
    imageInputLayer([32 32 3],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')
];

lgraph = addLayers(lgraph,layers);
figure
plot(lgraph)
Create Layer Graph from an Array of Layers

Create an array of layers.

```matlab
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')
];
```

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.
Extract Layer Graph of DAG Network

Load a pretrained squeezenet network. If the Deep Learning Toolbox™ Model for SqueezeNet Network support package is not installed, then the software provides a download link. You can use this trained network for classification and prediction.

```matlab
net = squeezenet;
```
To modify the network structure, first extract the structure of the DAG network by using `layerGraph`. You can then use the object functions of `LayerGraph` to modify the network architecture.

```matlab
lgraph = layerGraph(net)
```

Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A shortcut connection containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. All layers must have names and all names must be unique.

```matlab
layers = [  
  imageInputLayer([28 28 1],'Name','input')  
  convolution2dLayer(5,16,'Padding','same','Name','conv_1')  
  batchNormalizationLayer('Name','BN_1')  
  reluLayer('Name','relu_1')  
  convolution2dLayer(3,32,'Padding','same','Stride',2,'Name','conv_2')  
  batchNormalizationLayer('Name','BN_2')  
  reluLayer('Name','relu_2')  
  convolution2dLayer(3,32,'Padding','same','Name','conv_3')  
  batchNormalizationLayer('Name','BN_3')
  ]
```
reluLayer('Name','relu_3')
additionLayer(2,'Name','add')
averagePooling2dLayer(2,'Stride',2,'Name','avpool')
fullyConnectedLayer(10,'Name','fc')
softmaxLayer('Name','softmax')
classificationLayer('Name','classOutput')

Create a layer graph from the layer array. layerGraph connects all the layers in layers sequentially. Plot the layer graph.
lgraph = layerGraph(layers);
figure
plot(lgraph)
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the 'relu_3' layer. This arrangement enables the addition layer to add the outputs of the 'skipConv' and 'relu_3' layers. To check that the layer is in the graph, plot the layer graph.

```matlab
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv'); lgraph = addLayers(lgraph,skipConv); figure plot(lgraph)
```
Create the shortcut connection from the 'relu_1' layer to the 'add' layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named 'in1' and 'in2'. The 'relu_3' layer is already connected to the 'in1' input. Connect the 'relu_1' layer to the 'skipConv' layer and the 'skipConv' layer to the 'in2' input of the 'add' layer. The addition layer now sums the outputs of the 'relu_3' and 'skipConv' layers. To check that the layers are connected correctly, plot the layer graph.

```matlab
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```
Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

```matlab
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ... 
    'Shuffle','every-epoch', ... 
    'ValidationData',{XValidation,YValidation}, ... 
    'ValidationFrequency',30, ... 
)```

```matlab
layerGraph
```
Display the properties of the trained network. The network is a DAGNetwork object.

```matlab
net
net = DAGNetwork with properties:
    Layers: [16x1 nnet.cnn.layer.Layer]
    Connections: [16x2 table]
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```matlab
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```
Tips

- Layer graphs cannot specify the architecture of long short-term memory (LSTM) networks. For more information on how to create an LSTM network, see “Long Short-Term Memory Networks”.

See Also

DAGNetwork | addLayers | additionLayer | analyzeNetwork | assembleNetwork | connectLayers | depthConcatenationLayer | disconnectLayers | googlenet | inceptionresnetv2 | inceptionv3 | plot | removeLayers | replaceLayer | resnet101 | resnet18 | resnet50 | squeezenet | trainNetwork

Topics

“Create Simple Deep Learning Network for Classification”
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”

Introduced in R2017b
DAGNetwork

Directed acyclic graph (DAG) network for deep learning

Description

A DAG network is a neural network for deep learning with layers arranged as a directed acyclic graph. A DAG network can have a more complex architecture in which layers have inputs from multiple layers and outputs to multiple layers.

Creation

There are several ways to create a DAGNetwork object:

- Load a pretrained network using googlenet, resnet50, resnet101, or inceptionv3. For an example, see “Download GoogLeNet Support Package” on page 1-1344.
- Import a pretrained network from Keras using importKerasNetwork. For an example, see “Import and Plot Keras Network” on page 1-1384.
- Train or fine-tune a network using trainNetwork. For an example, see “Train Deep Learning Network to Classify New Images”.

Note To learn about other pretrained networks, see “Pretrained Deep Neural Networks”.

Properties

Layers — Network layers
Layer array

Network layers, specified as a Layer array.

Connections — Layer connections
table
Layer connections, specified as a table with two columns.

Each table row represents a connection in the layer graph. The first column, `Source`, specifies the source of each connection. The second column, `Destination`, specifies the destination of each connection. The connection sources and destinations are either layer names or have the form `'layerName/IOName'`, where `'IOName'` is the name of the layer input or output.

Data Types: `table`

**InputNames — Network input layer names**

cell array

Network input layer names, specified as a cell array of character vectors.

Data Types: `cell`

**OutputNames — Network output layer names**

cell array

Network output layer names, specified as a cell array of character vectors.

Data Types: `cell`

**Object Functions**

- **activations**  Compute deep learning network layer activations
- **classify**     Classify data using a trained deep learning neural network
- **predict**      Predict responses using a trained deep learning neural network
- **plot**         Plot neural network layer graph

**Examples**

**Create Simple DAG Network**

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
• A **shortcut connection** containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. All layers must have names and all names must be unique.

```matlab
layers = [imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(5,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')
    convolution2dLayer(3,32,'Padding','same','Stride',2,'Name','conv_2')
    batchNormalizationLayer('Name','BN_2')
    reluLayer('Name','relu_2')
    convolution2dLayer(3,32,'Padding','same','Name','conv_3')
    batchNormalizationLayer('Name','BN_3')
    reluLayer('Name','relu_3')
    additionLayer(2,'Name','add')
    averagePooling2dLayer(2,'Stride',2,'Name','avpool')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classOutput')];
```

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```matlab
lgraph = layerGraph(layers);
figure
plot(lgraph)
```
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the 'relu_3' layer. This arrangement enables the addition layer to add the outputs of the 'skipConv' and 'relu_3' layers. To check that the layer is in the graph, plot the layer graph.

```matlab
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```
Create the shortcut connection from the 'relu_1' layer to the 'add' layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named 'in1' and 'in2'. The 'relu_3' layer is already connected to the 'in1' input. Connect the 'relu_1' layer to the 'skipConv' layer and the 'skipConv' layer to the 'in2' input of the 'add' layer. The addition layer now sums the outputs of the 'relu_3' and 'skipConv' layers. To check that the layers are connected correctly, plot the layer graph.

```matlab
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```
Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;

Specify training options and train the network. `trainNetwork` validates the network using the validation data every `ValidationFrequency` iterations.

options = trainingOptions('sgdm', ... 'MaxEpochs',8, ... 'Shuffle','every-epoch', ... 'ValidationData',{XValidation,YValidation}, ... 'ValidationFrequency',30, ...)
Display the properties of the trained network. The network is a DAGNetwork object.

```matlab
net
net = DAGNetwork with properties:

    Layers: [16×1 nnet.cnn.layer.Layer]
    Connections: [16×2 table]
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```matlab
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the activations and predict object functions are supported.
- To create a DAGNetwork object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the activations and predict object functions are supported.
- To create a DAGNetwork object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also
SeriesNetwork | analyzeNetwork | assembleNetwork | classify | googlenet | importKerasNetwork | inceptionresnetv2 | inceptionv3 | layerGraph | plot | predict | resnet101 | resnet18 | resnet50 | squeezenet | trainNetwork | trainingOptions

Topics
“Deep Learning in MATLAB”
“Classify Image Using GoogLeNet”
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”
“Pretrained Deep Neural Networks”

Introduced in R2017b
additionLayer

Addition layer

Description

An addition layer adds inputs from multiple neural network layers element-wise.

Specify the number of inputs to the layer when you create it. The inputs to the layer have the names 'in1', 'in2', ..., 'inN', where N is the number of inputs. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers. All inputs to an addition layer must have the same dimension.

Creation

Syntax

layer = additionLayer(numInputs)
layer = additionLayer(numInputs,'Name',Name)

Description

layer = additionLayer(numInputs) creates an addition layer that adds numInputs inputs element-wise. This function also sets the NumInputs property.

layer = additionLayer(numInputs,'Name',Name) also sets the Name property. To create a network containing an addition layer, you must specify a layer name.

Properties

NumInputs — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer.
The inputs have the names 'in1', 'in2', ..., 'inN', where N equals NumInputs. For example, if NumInputs equals 3, then the inputs have the names 'in1', 'in2', and 'in3'. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers.

**Name — Layer name**

' ' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include this layer in a layer graph, you must specify a layer name.

Data Types: char | string

**InputNames — Input Names**

{'}' (default)

Input names, specified as {'} in1', 'in2', ..., 'inN'}, where N is the number of inputs of the layer.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'} out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**

**Create and Connect Addition Layer**

Create an addition layer with two inputs and the name 'add_1'.

```
add = additionLayer(2,'Name','add_1')
```
add = AdditionLayer with properties:

    Name: 'add_1'
    NumInputs: 2
    InputNames: {'in1' 'in2'}

Create two ReLU layers and connect them to the addition layer. The addition layer sums
the outputs from the ReLU layers.

    relu_1 = reluLayer('Name','relu_1');
    relu_2 = reluLayer('Name','relu_2');

    lgraph = layerGraph;
    lgraph = addLayers(lgraph,relu_1);
    lgraph = addLayers(lgraph,relu_2);
    lgraph = addLayers(lgraph,add);

    lgraph = connectLayers(lgraph,'relu_1','add_1/in1');
    lgraph = connectLayers(lgraph,'relu_2','add_1/in2');

    plot(lgraph)
Create Simple DAG Network

Create a simple directed acyclic graph (DAG) network for deep learning. Train the network to classify images of digits. The simple network in this example consists of:

- A main branch with layers connected sequentially.
- A shortcut connection containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs for the addition layer to sum. All layers must have names and all names must be unique.
layers = [
    imageInputLayer([28 28 1], 'Name', 'input')
    convolution2dLayer(5,16, 'Padding', 'same', 'Name', 'conv_1')
    batchNormalizationLayer('Name', 'BN_1')
    reluLayer('Name', 'relu_1')
    convolution2dLayer(3,32, 'Padding', 'same', 'Stride', 2, 'Name', 'conv_2')
    batchNormalizationLayer('Name', 'BN_2')
    reluLayer('Name', 'relu_2')
    convolution2dLayer(3,32, 'Padding', 'same', 'Name', 'conv_3')
    batchNormalizationLayer('Name', 'BN_3')
    reluLayer('Name', 'relu_3')
    additionLayer(2, 'Name', 'add')
    averagePooling2dLayer(2, 'Stride', 2, 'Name', 'avpool')
    fullyConnectedLayer(10, 'Name', 'fc')
    softmaxLayer('Name', 'softmax')
    classificationLayer('Name', 'classOutput')];

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

`lgraph = layerGraph(layers);`
figure
plot(lgraph)
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the 'relu_3' layer. This arrangement enables the addition layer to add the outputs of the 'skipConv' and 'relu_3' layers. To check that the layer is in the graph, plot the layer graph.

```matlab
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```
Create the shortcut connection from the 'relu_1' layer to the 'add' layer. Because you specified two as the number of inputs to the addition layer when you created it, the layer has two inputs named 'in1' and 'in2'. The 'relu_3' layer is already connected to the 'in1' input. Connect the 'relu_1' layer to the 'skipConv' layer and the 'skipConv' layer to the 'in2' input of the 'add' layer. The addition layer now sums the outputs of the 'relu_3' and 'skipConv' layers. To check that the layers are connected correctly, plot the layer graph.

```matlab
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```
Load the training and validation data, which consists of 28-by-28 grayscale images of digits.

[XTrain,YTrain] = digitTrain4DArrayData;
[XValidation,YValidation] = digitTest4DArrayData;

Specify training options and train the network. trainNetwork validates the network using the validation data every ValidationFrequency iterations.

options = trainingOptions('sgdm', ... 
'MaxEpochs',8, ... 
'Shuffle','every-epoch', ..., 
'ValidationData',{XValidation,YValidation}, ..., 
'ValidationFrequency',30, ...
Display the properties of the trained network. The network is a DAGNetwork object.

```
net
net = DAGNetwork with properties:
    Layers: [16×1 nnet.cnn.layer.Layer]
    Connections: [16×2 table]
```

Classify the validation images and calculate the accuracy. The network is very accurate.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```
accuracy = 0.9968

Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
depthConcatenationLayer | layerGraph | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“Train Residual Network for Image Classification”
“List of Deep Learning Layers”

Introduced in R2017b
depthConcatenationLayer

Depth concatenation layer

Description

A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).

Specify the number of inputs to the layer when you create it. The inputs have the names 'in1','in2',...,'inN', where N is the number of inputs. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers.

Creation

Syntax

layer = depthConcatenationLayer(numInputs)
layer = depthConcatenationLayer(numInputs,'Name',name)

Description

layer = depthConcatenationLayer(numInputs) creates a depth concatenation layer that concatenates numInputs inputs along the third (channel) dimension. This function also sets the NumInputs property.

layer = depthConcatenationLayer(numInputs,'Name',name) also sets the Name property. To create a network containing a depth concatenation layer, you must specify a layer name.

Properties

NumInputs — Number of inputs

positive integer
Number of inputs to the layer, specified as a positive integer.

The inputs have the names 'in1','in2',..., 'inN', where N equals NumInputs. For example, if NumInputs equals 3, then the inputs have the names 'in1','in2', and 'in3'. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers.

**Name — Layer name**

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include this layer in a layer graph, you must specify a layer name.

Data Types: char | string

**InputNames — Input Names**

{'in1','in2',..., 'inN'} (default)

Input names, specified as {'in1','in2',..., 'inN'}, where N is the number of inputs of the layer.

Data Types: cell

**NumOutputs — Number of outputs**

1 (default)

Number of outputs of the layer. This layer has a single output only.

Data Types: double

**OutputNames — Output names**

{'out'} (default)

Output names of the layer. This layer has a single output only.

Data Types: cell

**Examples**

**Create and Connect Depth Concatenation Layer**

Create a depth concatenation layer with two inputs and the name 'concat_1'.
concat = depthConcatenationLayer(2,'Name','concat_1')

concat =
    DepthConcatenationLayer with properties:

        Name: 'concat_1'
        NumInputs: 2
        InputNames: {'in1'  'in2'}

Create two ReLU layers and connect them to the depth concatenation layer. The depth concatenation layer concatenates the outputs from the ReLU layers.

relu_1 = reluLayer('Name','relu_1');
relu_2 = reluLayer('Name','relu_2');

lgraph = layerGraph;
lgraph = addLayers(lgraph,relu_1);
lgraph = addLayers(lgraph,relu_2);
lgraph = addLayers(lgraph,concat);

lgraph = connectLayers(lgraph,'relu_1','concat_1/in1');
lgraph = connectLayers(lgraph,'relu_2','concat_1/in2');

plot(lgraph)
Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.
See Also
additionLayer | connectLayers | disconnectLayers | layerGraph | trainNetwork

Topics
“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“Train Residual Network for Image Classification”
“List of Deep Learning Layers”

Introduced in R2017b
concatenationLayer

Concatenation layer

Description

A concatenation layer takes inputs and concatenates them along a specified dimension. The inputs must have the same size in all dimensions except the concatenation dimension.

Specify the number of inputs to the layer when you create it. The inputs have the names 'in1', 'in2', ..., 'inN', where N is the number of inputs. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers.

Creation

Syntax

layer = concatenationLayer(dim,numInputs)
layer = concatenationLayer(dim,numInputs,'Name',name)

Description

layer = concatenationLayer(dim,numInputs) creates a concatenation layer that concatenates numInputs inputs along the specified dimension, dim. This function also sets the Dim and NumInputs properties.

layer = concatenationLayer(dim,numInputs,'Name',name) also sets the Name property. To create a network containing a concatenation layer, you must specify a layer name.
Properties

Concatenation

**Dim** — Concatenation dimension

positive integer

Concatenation dimension, specified as a positive integer.

Example: 4

Layer

**Name** — Layer name

'' (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. To include this layer in a layer graph, you must specify a layer name.

Data Types: char | string

**NumInputs** — Number of inputs

positive integer

Number of inputs to the layer, specified as a positive integer.

The inputs have the names 'in1','in2',..., 'inN', where N equals NumInputs. For example, if NumInputs equals 3, then the inputs have the names 'in1','in2', and 'in3'. Use the input names when connecting or disconnecting the layer by using connectLayers or disconnectLayers.

**InputNames** — Input Names

{'in1','in2',..., 'inN'} (default)

Input names, specified as {'in1','in2',..., 'inN'}, where N is the number of inputs of the layer.

Data Types: cell

**NumOutputs** — Number of outputs

1 (default)
Number of outputs of the layer. This layer has a single output only.
Data Types: double

**OutputNames — Output names**

`{'out'}` (default)

Output names of the layer. This layer has a single output only.
Data Types: cell

**Examples**

**Create and Connect Concatenation Layer**

Create a concatenation layer that concatenates two inputs along the fourth dimension (channels). Name the concatenation layer 'concat'.

```matlab
congat = concatenationLayer(4,2,'Name','concat')
```

```matlab
congat =  
    ConcatenationLayer with properties:  
        Name: 'concat'  
        Dim: 4  
        NumInputs: 2  
        InputNames: {'in1' 'in2'}
```

Create two ReLU layers and connect them to the concatenation layer. The concatenation layer concatenates the outputs from the ReLU layers.

```matlab
relu_1 = reluLayer('Name','relu_1');  
relu_2 = reluLayer('Name','relu_2');
```

```matlab
lgraph = layerGraph();  
lgraph = addLayers(lgraph, relu_1);  
lgraph = addLayers(lgraph, relu_2);  
lgraph = addLayers(lgraph, concat);
```

```matlab
lgraph = connectLayers(lgraph, 'relu_1', 'concat/in1');  
lgraph = connectLayers(lgraph, 'relu_2', 'concat/in2');  
plot(lgraph)
```
See Also
additionLayer | connectLayers | disconnectLayers | layerGraph | trainNetwork

Topics
“3-D Brain Tumor Segmentation Using Deep Learning”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”
Introduced in R2019a
**googlenet**

Pretrained GoogLeNet convolutional neural network

**Syntax**

```
net = googlenet
net = googlenet('Weights',weights)
```

**Description**

GoogLeNet is a pretrained convolutional neural network that is 22 layers deep. You can load a network trained on either the ImageNet [1] or Places365 [2] [3] data sets. The network trained on ImageNet classifies images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. The network trained on Places365 is similar to the network trained on ImageNet, but classifies images into 365 different place categories, such as field, park, runway, and lobby. These networks have learned different feature representations for a wide range of images. The networks both have an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

To classify new images using GoogLeNet, use `classify`. For an example, see “Classify Image Using GoogLeNet”.

You can retrain a GoogLeNet network to perform a new task using transfer learning. When performing transfer learning, the most common approach is to use networks pretrained on the ImageNet data set. If the new task is similar to classifying scenes, then using the network trained on Places-365 can give higher accuracies. For an example showing how to retrain GoogLeNet on a new classification task, see “Train Deep Learning Network to Classify New Images”

```
net = googlenet
```


This function requires the Deep Learning Toolbox Model for GoogLeNet Network support package. If this support package is not installed, then the function provides a download link.
net = googlenet('Weights',weights) returns a GoogLeNet network trained on either the ImageNet or Places365 data set.

The network trained on ImageNet requires the Deep Learning Toolbox Model *for GoogLeNet Network* support package. The network trained on Places365 requires the Deep Learning Toolbox Model *for Places365-GoogLeNet Network* support package. If the required support package is not installed, then the function provides a download link.

### Examples

**Download GoogLeNet Support Package**

Download and install the Deep Learning Toolbox Model *for GoogLeNet Network* support package.

Type googlenet at the command line.

```matlab
goooglenet
```

If the Deep Learning Toolbox Model *for GoogLeNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `googlenet` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```matlab
googlenet
ans =
```

```matlab
DAGNetwork with properties:

Layers: [144×1 nnet.cnn.layer.Layer]
Connections: [170×2 table]
```

### Input Arguments

- **`weights` — Source of network parameters**
  - `'imagenet'` (default) | `'places365'`
Source of network parameters, specified as 'imagenet' or 'places365'.

If weights equals 'imagenet', then the network has weights trained on the ImageNet data set. If weights equals 'places365' then the network has weights trained on the Places365 data set.

Example: 'places365'

**Output Arguments**

net — Pretrained GoogLeNet convolutional neural network

DAGNetwork object

Pretrained GoogLeNet convolutional neural network, returned as a DAGNetwork object.

**References**


**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.
For code generation, you can load the network by using the syntax `net = googlenet` or by passing the `googlenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('googlenet')`.

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

**GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = googlenet` or by passing the `googlenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('googlenet')`.

**See Also**

DAGNetwork | alexnet | densenet201 | inceptionresnetv2 | inceptionv3 | layerGraph | plot | resnet101 | resnet18 | resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

**Topics**

“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Train Residual Network for Image Classification”

**Introduced in R2017b**
resnet18

Pretrained ResNet-18 convolutional neural network

Syntax

net = resnet18

Description

ResNet-18 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 18 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the ResNet-18 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ResNet-18.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ResNet-18 instead of GoogLeNet.

net = resnet18 returns a pretrained ResNet-18 convolutional neural network.

This function requires the Deep Learning Toolbox Model for ResNet-18 Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download ResNet-18 Support Package

Type `resnet18` at the command line.

`resnet18`

If the Deep Learning Toolbox Model for ResNet-18 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet18` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

`resnet18`

```plaintext
ans =

DAGNetwork with properties:

    Layers: [72x1 nnet.cnn.layer.Layer]
    Connections: [79x2 table]
```

**Output Arguments**

- **net** — **Pretrained ResNet-18 convolutional neural network**
  
  DAGNetwork object

  Pretrained ResNet-18 convolutional neural network, returned as a `DAGNetwork` object.

**References**


**See Also**

- `DAGNetwork`
- `alexnet`
- `densenet201`
- `googlenet`
- `inceptionresnetv2`
- `layerGraph`
- `plot`
- `resnet101`
- `resnet50`
- `squeezenet`
- `trainNetwork`
- `vgg16`
- `vgg19`
Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2018a
**resnet50**

Pretrained ResNet-50 convolutional neural network

**Syntax**

```
net = resnet50
```

**Description**

ResNet-50 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 50 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the ResNet-50 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ResNet-50.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ResNet-50 instead of GoogLeNet.

```
net = resnet50
```

returns a pretrained ResNet-50 network.

This function requires the Deep Learning Toolbox Model for ResNet-50 Network support package. If this support package is not installed, then the function provides a download link.

**Examples**

**Download ResNet-50 Support Package**

Type `resnet50` at the command line.

```
resnet50
```

If the Deep Learning Toolbox Model for ResNet-50 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet50` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
resnet50
ans =
    DAGNetwork with properties:
        Layers: [177x1 nnet.cnn.layer.Layer]
        Connections: [192x2 table]
```

**Output Arguments**

- **net** — **Pretrained ResNet-50 convolutional neural network**
  DAGNetwork object

Pretrained ResNet-50 convolutional neural network, returned as a `DAGNetwork` object.

**References**


[3] https://keras.io/applications/#resnet50
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = resnet50` or by passing the `resnet50` function to `coder.loadDeepLearningNetwork`. For example:
```
net = coder.loadDeepLearningNetwork('resnet50')
```

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = resnet50` or by passing the `resnet50` function to `coder.loadDeepLearningNetwork`. For example:
```
net = coder.loadDeepLearningNetwork('resnet50')
```

See Also
DAGNetwork | alexnet | densenet201 | googlenet | inceptionresnetv2 | layerGraph | plot | resnet101 | resnet18 | squeezenet | trainNetwork | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b
resnet101

Pretrained ResNet-101 convolutional neural network

Syntax

net = resnet101

Description

ResNet-101 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 101 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.


To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ResNet-101 instead of GoogLeNet.

net = resnet101 returns a pretrained ResNet-101 network.

This function requires the Deep Learning Toolbox Model for ResNet-101 Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download ResNet-101 Support Package

Type `resnet101` at the command line.

```matlab
resnet101
```

If the Deep Learning Toolbox Model for ResNet-101 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `resnet101` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```matlab
resnet101
ans =
```

```matlab
DAGNetwork with properties:

Layers: [347×1 nnet.cnn.layer.Layer]
Connections: [379×2 table]
```

### Output Arguments

**net — Pretrained ResNet-101 convolutional neural network**

DAGNetwork object

Pretrained ResNet-101 convolutional neural network, returned as a `DAGNetwork` object.

### References


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax net = resnet101 or by passing the resnet101 function to coder.loadDeepLearningNetwork. For example: net = coder.loadDeepLearningNetwork('resnet101')

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax net = resnet101 or by passing the resnet101 function to coder.loadDeepLearningNetwork. For example: net = coder.loadDeepLearningNetwork('resnet101')

See Also
DAGNetwork | alexnet | densenet201 | googlenet | inceptionresnetv2 | inceptionv3 | layerGraph | plot | resnet18 | resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b
**densenet201**

Pretrained DenseNet-201 convolutional neural network

**Syntax**

```
net = densenet201
```

**Description**

DenseNet-201 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 201 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the DenseNet-201 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with DenseNet-201.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load DenseNet-201 instead of GoogLeNet.

```
net = densenet201
```
returns a pretrained DenseNet-201 convolutional neural network.

This function requires the Deep Learning Toolbox Model for DenseNet-201 Network support package. If this support package is not installed, then the function provides a download link.

**Examples**
Download DenseNet-201 Support Package

Download and install the Deep Learning Toolbox Model for DenseNet-201 Network support package.

Type densenet201 at the command line.

densenet201

If the Deep Learning Toolbox Model for DenseNet-201 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install. Check that the installation is successful by typing densenet201 at the command line. If the required support package is installed, then the function returns a DAGNetwork object.

densenet201

ans =

DAGNetwork with properties:

    Layers: [709×1 nnet.cnn.layer.Layer]
    Connections: [806×2 table]

Output Arguments

net — Pretrained DenseNet-201 convolutional neural network
DAGNetwork object

Pretrained DenseNet-201 convolutional neural network, returned as a DAGNetwork object.

References


See Also
DAGNetwork | alexnet | googlenet | inceptionresnetv2 | inceptionv3 |
layerGraph | plot | resnet101 | resnet18 | resnet50 | squeezenet |
trainNetwork | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2018a
inceptionv3

Pretrained Inception-v3 convolutional neural network

Syntax

net = inceptionv3

Description

Inception-v3 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 48 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299-by-299. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the Inception-v3 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with Inception-v3.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load Inception-v3 instead of GoogLeNet.

net = inceptionv3 returns a pretrained Inception-v3 network.

This function requires the Deep Learning Toolbox Model for Inception-v3 Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download Inception-v3 Support Package

Download and install the Deep Learning Toolbox Model for Inception-v3 Network support package.
Type `inceptionv3` at the command line.

```
inceptionv3
```

If the Deep Learning Toolbox Model for Inception-v3 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `inceptionv3` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
inceptionv3
ans =

DAGNetwork with properties:

    Layers: [316x1 nnet.cnn.layer.Layer]
    Connections: [350x2 table]
```

## Output Arguments

- **net** — Pretrained Inception-v3 convolutional neural network
  
  DAGNetwork object

  Pretrained Inception-v3 convolutional neural network, returned as a `DAGNetwork` object.

## References


[3] [https://keras.io/applications/#inceptionv3](https://keras.io/applications/#inceptionv3)
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, you can load the network by using the syntax `net = inceptionv3` or by passing the `inceptionv3` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionv3')`

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = inceptionv3` or by passing the `inceptionv3` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionv3')`.

See Also
DAGNetwork | alexnet | densenet201 | googlenet | inceptionresnetv2 | layerGraph | plot | resnet18 | resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b
inceptionresnetv2

Pretrained Inception-ResNet-v2 convolutional neural network

Syntax

net = inceptionresnetv2

Description

Inception-ResNet-v2 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 164 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299-by-299. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the Inception-ResNet-v2 network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with Inception-ResNet-v2.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load Inception-ResNet-v2 instead of GoogLeNet.

net = inceptionresnetv2 returns a pretrained Inception-ResNet-v2 network.

This function requires the Deep Learning Toolbox Model for Inception-ResNet-v2 Network support package. If this support package is not installed, then the function provides a download link.

Examples
**Load Inception-ResNet-v2 Network**


Type `inceptionresnetv2` at the command line.

`inceptionresnetv2`

If the Deep Learning Toolbox Model for Inception-ResNet-v2 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `inceptionresnetv2` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```matlab
net = inceptionresnetv2
```

```matlab
net = DAGNetwork with properties:
    Layers: [825×1 nnet.cnn.layer.Layer]
    Connections: [922×2 table]
```

**Output Arguments**

- **net** — Pretrained Inception-ResNet-v2 convolutional neural network
  
  **DAGNetwork object**

Pretrained Inception-ResNet-v2 convolutional neural network, returned as a `DAGNetwork` object.

**References**


See Also
DAGNetwork | alexnet | densenet201 | googlenet | importKerasLayers |
importKerasNetwork | inceptionv3 | layerGraph | plot | resnet101 | resnet18 |
resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2017b

[3] https://keras.io/applications/#inceptionresnetv2
squeezenet

Pretrained SqueezeNet convolutional neural network

Syntax

net = squeezenet

Description

SqueezeNet is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 18 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. This function returns a SqueezeNet v1.1 network, which has similar accuracy to SqueezeNet v1.0 but requires fewer floating-point operations per prediction [3]. The network has an image input size of 227-by-227. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the SqueezeNet network. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with SqueezeNet.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load SqueezeNet instead of GoogLeNet.

net = squeezenet returns a pretrained SqueezeNet convolutional neural network.

This function requires the Deep Learning Toolbox Model for SqueezeNet Network support package. If this support package is not installed, then the function provides a download link.

Examples
Load SqueezeNet Network


Type squeezenet at the command line.

squeezenet

If the Deep Learning Toolbox Model for SqueezeNet Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install. Check that the installation is successful by typing squeezenet at the command line. If the required support package is installed, then the function returns a DAGNetwork object.

net = squeezenet

net = DAGNetwork with properties:

    Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]

Output Arguments

net — Pretrained SqueezeNet convolutional neural network
DAGNetwork object

Pretrained SqueezeNet convolutional neural network, returned as a DAGNetwork object.

References


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For code generation, load the network by passing the squeezenet function to coder.loadDeepLearningNetwork. For example: net = coder.loadDeepLearningNetwork('squeezenet')

For more information, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

For code generation, you can load the network by using the syntax net = squeezenet or by passing the squeezenet function to coder.loadDeepLearningNetwork. For example: net = coder.loadDeepLearningNetwork('squeezenet')

See Also
DAGNetwork| alexnet| densenet201| googlenet| inceptionresnetv2| inceptionv3| layerGraph| plot| resnet101| resnet18| resnet50| trainNetwork| vgg16| vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2018a
mobilenetv2

Pretrained MobileNet-v2 convolutional neural network

Syntax

net = mobilenetv2

Description

MobileNet-v2 is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 54 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the MobileNet-v2 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with MobileNet-v2.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load MobileNet-v2 instead of GoogLeNet.

net = mobilenetv2 returns a pretrained MobileNet-v2 convolutional neural network.

This function requires the Deep Learning Toolbox Model for MobileNet-v2 Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download MobileNet-v2 Support Package

Type `mobilenetv2` at the command line.

`mobilenetv2`

If the Deep Learning Toolbox Model for MobileNet-v2 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `mobilenetv2` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```matlab
mobilenetv2
ans =
    DAGNetwork with properties:
    Layers: [155×1 nnet.cnn.layer.Layer]
    Connections: [164×2 table]
```

**Output Arguments**

`net` — Pretrained MobileNet-v2 convolutional neural network

DAGNetwork object

Pretrained MobileNet-v2 convolutional neural network, returned as a `DAGNetwork` object.

**References**


Extended Capabilities

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:
For code generation, you can load the network by using the syntax `net = mobilenetv2` or by passing the `mobilenetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('mobilenetv2')`

See Also
DAGNetwork | alexnet | densenet201 | googlenet | inceptionresnetv2 | layerGraph | plot | resnet101 | resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

Topics
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2019a
xception

Pretrained Xception convolutional neural network

Syntax

net = xception

Description

Xception is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network is 71 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 299-by-299. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the Xception model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with Xception.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load Xception instead of GoogLeNet.

net = xception returns a pretrained Xception convolutional neural network.

This function requires the Deep Learning Toolbox Model for Xception Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download Xception Support Package

Download and install the Deep Learning Toolbox Model for Xception Network support package.
Type `xception` at the command line.

```
xception
```

If the Deep Learning Toolbox Model for Xception Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `xception` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
xception
```

```
ans =

DAGNetwork with properties:

    Layers: [171×1 nnet.cnn.layer.Layer]
    Connections: [182×2 table]
```

### Output Arguments

- **net** — Pretrained Xception convolutional neural network
  
  DAGNetwork object

  Pretrained Xception convolutional neural network, returned as a `DAGNetwork` object.

### References


### Extended Capabilities

#### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.
Usage notes and limitations:

For code generation, you can load the network by using the syntax `net = xception` or by passing the `xception` function to `coder.loadDeepLearningNetwork`. For example:

```matlab
net = coder.loadDeepLearningNetwork('xception')
```

See Also

DAGNetwork | alexnet | densenet201 | googlenet | inceptionresnetv2 |
layerGraph | plot | resnet101 | resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2019a
nasnetlarge

Pretrained NASNet-Large convolutional neural network

Syntax

net = nasnetlarge

Description

NASNet-Large is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 331-by-331. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the NASNet-Large model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with NASNet-Large.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load NASNet-Large instead of GoogLeNet.

net = nasnetlarge returns a pretrained NASNet-Large convolutional neural network.

This function requires the Deep Learning Toolbox Model for NASNet-Large Network support package. If this support package is not installed, then the function provides a download link.

Examples
Download NASNet-Large Support Package

Download and install the Deep Learning Toolbox Model for NASNet-Large Network support package.

Type nasnetlarge at the command line.

nasnetlarge

If the Deep Learning Toolbox Model for NASNet-Large Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install. Check that the installation is successful by typing nasnetlarge at the command line. If the required support package is installed, then the function returns a DAGNetwork object.

nasnetlarge

ans =

DAGNetwork with properties:

    Layers: [1244×1 nnet.cnn.layer.Layer]
    Connections: [1463×2 table]

Transfer Learning with NASNet-Large

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the NASNet-Large network instead of GoogLeNet.

net = nasnetlarge
Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

**Output Arguments**

- `net` — **Pretrained NASNet-Large convolutional neural network**
  
  DAGNetwork object

Pretrained NASNet-Large convolutional neural network, returned as a DAGNetwork object.

**References**


**See Also**

DAGNetwork | densenet201 | googlenet | inceptionresnetv2 | layerGraph | nasnetmobile | plot | resnet101 | resnet50 | shufflenet | squeezenet | trainNetwork | vgg16 | vgg19

**Topics**

- “Deep Learning in MATLAB”
- “Pretrained Deep Neural Networks”
- “Classify Image Using GoogLeNet”
- “Train Deep Learning Network to Classify New Images”
- “Train Residual Network for Image Classification”

**Introduced in R2019a**
nasnetmobile

Pretrained NASNet-Mobile convolutional neural network

**Syntax**

```matlab
net = nasnetmobile
```

**Description**

NASNet-Mobile is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use `classify` to classify new images using the NASNet-Mobile model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with NASNet-Mobile.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load NASNet-Mobile instead of GoogLeNet.

```matlab
net = nasnetmobile
```
returns a pretrained NASNet-Mobile convolutional neural network.

This function requires the *Deep Learning Toolbox Model for NASNet-Mobile Network* support package. If this support package is not installed, then the function provides a download link.

**Examples**
**Download NASNet-Mobile Support Package**

Download and install the *Deep Learning Toolbox Model for NASNet-Mobile Network* support package.

Type `nasnetmobile` at the command line.

nasnetmobile

If the *Deep Learning Toolbox Model for NASNet-Mobile Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `nasnetmobile` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

nasnetmobile

ans =

    DAGNetwork with properties:

          Layers: [914×1 nnet.cnn.layer.Layer]
    Connections: [1073×2 table]

**Transfer Learning with NASNet-Mobile**

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the NASNet-Mobile network instead of GoogLeNet.

net = nasnetmobile
Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

**Output Arguments**

- **net** — Pretrained NASNet-Mobile convolutional neural network
  
  DAGNetwork object

  Pretrained NASNet-Mobile convolutional neural network, returned as a DAGNetwork object.

**References**


**See Also**

DAGNetwork | densenet201 | googlenet | inceptionresnetv2 | layerGraph | nasnetlarge | plot | resnet101 | resnet50 | shufflenet | squeezenet | trainNetwork | vgg16 | vgg19

**Topics**

- “Deep Learning in MATLAB”
- “Pretrained Deep Neural Networks”
- “Classify Image Using GoogLeNet”
- “Train Deep Learning Network to Classify New Images”
- “Train Residual Network for Image Classification”

**Introduced in R2019a**
shuffle

tnet

Pretrained ShuffleNet convolutional neural network

Syntax

net = shufflenet

Description

ShuffleNet is a convolutional neural network that is trained on more than a million images from the ImageNet database [1]. The network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. For more pretrained networks in MATLAB, see “Pretrained Deep Neural Networks”.

You can use classify to classify new images using the ShuffleNet model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ShuffleNet.

To retrain the network on a new classification task, follow the steps of “Train Deep Learning Network to Classify New Images” and load ShuffleNet instead of GoogLeNet.

net = shufflenet returns a pretrained ShuffleNet convolutional neural network.

This function requires the Deep Learning Toolbox Model for ShuffleNet Network support package. If this support package is not installed, then the function provides a download link.

Examples

Download ShuffleNet Support Package

Type `shufflenet` at the command line.

```
shufflenet
```

If the *Deep Learning Toolbox Model for ShuffleNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by typing `shufflenet` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
shufflenet
ans =
DAGNetwork with properties:
    Layers: [173×1 nnet.cnn.layer.Layer]
    Connections: [188×2 table]
```

**Transfer Learning with ShuffleNet**

You can use transfer learning to retrain the network to classify a new set of images.

Open the example “Train Deep Learning Network to Classify New Images”. The original example uses the GoogLeNet pretrained network. To perform transfer learning using a different network, load your desired pretrained network and follow the steps in the example.

Load the ShuffleNet network instead of GoogLeNet.

```
net = shufflenet
```

Follow the remaining steps in the example to retrain your network. You must replace the last learnable layer and the classification layer in your network with new layers for training. The example shows you how to find which layers to replace.

**Output Arguments**

```
net — Pretrained ShuffleNet convolutional neural network
DAGNetwork object
```
Pretrained ShuffleNet convolutional neural network, returned as a DAGNetwork object.

References


See Also

DAGNetwork | densenet201 | googlenet | inceptionresnetv2 | layerGraph | nasnetlarge | nasnetmobile | plot | resnet101 | resnet50 | squeezenet | trainNetwork | vgg16 | vgg19

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Classify Image Using GoogLeNet”
“Train Deep Learning Network to Classify New Images”
“Train Residual Network for Image Classification”

Introduced in R2019a
importKerasNetwork

Import a pretrained Keras network and weights

Syntax

net = importKerasNetwork(modelfile)
net = importKerasNetwork(modelfile,Name,Value)

Description

net = importKerasNetwork(modelfile) imports a pretrained TensorFlow™-Keras network and its weights from modelfile.

This function requires Deep Learning Toolbox Importer for TensorFlow-Keras Models support package. If this support package is not installed, the function provides a download link.

net = importKerasNetwork(modelfile,Name,Value) imports a pretrained TensorFlow-Keras network and its weights with additional options specified by one or more name-value pair arguments.

For example, importKerasNetwork(modelfile,'WeightFile',weights) imports the network from the model file modelfile and weights from the weight file weights. In this case, modelfile can be in HDF5 or JSON format, and the weight file must be in HDF5 format.

Examples

Download and Install Deep Learning Toolbox Importer for TensorFlow-Keras Models

Type `importKerasNetwork` at the command line.

```matlab
importKerasNetwork
```

If the Deep Learning Toolbox Importer for TensorFlow-Keras Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file `'digitsDAGnet.h5'` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```matlab
modelfile = 'digitsDAGnet.h5';
net = importKerasNetwork(modelfile)
```

Warning: Saved Keras networks do not include classes. Classes will be set to `categorical(1:N)`, where `N` is the number of classes in the classification output layer of the network. To specify classes, use the 'Classes' argument.

```matlab
net =

    DAGNetwork with properties:

    Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
```

### Import and Plot Keras Network

Specify the file to import. The file `digitsDAGnet.h5` contains a directed acyclic graph convolutional neural network that classifies images of digits.

```matlab
modelfile = 'digitsDAGnet.h5';
```

Import the network.

```matlab
net = importKerasNetwork(modelfile)
```

Warning: Saved Keras networks do not include classes. Classes will be set to `categorical(1:N)`, where `N` is the number of classes in the classification output layer of the network. To specify classes, use the 'Classes' argument.

```matlab
net =

    DAGNetwork with properties:

    Layers: [13×1 nnet.cnn.layer.Layer]
Connections: [13×2 table]

Plot the network architecture.

```matlab
figure
plot(net);
title('DAG Network Architecture')
```

**Import Keras Network and Weights**

Specify the network and the weight files to import.
modelfile = 'digitsDAGnet.json';
weights = 'digitsDAGnet.weights.h5';

This is a directed acyclic graph convolutional neural network trained on the digits data.

Import network architecture and import the weights from separate files. The .json file
does not have an output layer or information on the cost function. Specify the output layer
type when you import the files.

net = importKerasNetwork(modelfile,'WeightFile',weights, ... 
    'OutputLayerType','classification')

Warning: Saved Keras networks do not include classes. Classes will be set to categorical(1:N),
where N is the number of classes in the classification output layer of the network. To specify 
classes, use the 'Classes' argument.

net = DAGNetwork with properties:
     Layers: [13×1 nnet.cnn.layer.Layer]
     Connections: [13×2 table]

**Import Pretrained Keras Network to Classify Image**

Specify the model file.

modelfile = 'digitsDAGnet.h5';

Specify class names.

classNames = {'0','1','2','3','4','5','6','7','8','9'};

Import the Keras network with the class names.

net = importKerasNetwork(modelfile,'Classes',classNames);

Read the image to classify.

digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets', ... 
    'DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));

Classify the image using the pretrained network.
label = classify(net,I);

Display the image and the classification result.

figure
imshow(I)
title(['Classification result: ' char(label)])

Classification result: 5

Input Arguments

modelfile — Name of Keras model file
character vector | string scalar

Name of the model file containing the network architecture, and possibly the weights, specified as a character vector or a string scalar. The file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

If `modelfile` includes

- The network architecture and weights, then it must be in HDF5 (.h5) format.
- Only the network architecture, then it can be in HDF5 or JSON (.json) format.

If `modelfile` includes only the network architecture, then you must supply the weights in an HDF5 file, using the 'WeightFile' name-value pair argument.

Example: 'digitsnet.h5'

Data Types: char | string
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example:
```
importKerasNetwork(modelfile,'OutputLayerType','classification','Classes',classes) imports a network from the model file modelfile, adds an output layer for a classification problem at the end of the Keras layers, and specifies classes as the classes of the output layer.
```

**WeightFile — Name of file containing weights**
character vector | string scalar

Name of file containing weights, specified as a character vector or a string scalar. **WeightFile** must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: 'WeightFile','weights.h5'

**OutputLayerType — Type of output layer**
'classification' | 'regression' | 'pixelclassification'

Type of the output layer that the function appends to the end of the imported network architecture when **modelfile** does not specify a loss function, specified as 'classification', 'regression', or 'pixelclassification'. Appending a **pixelClassificationLayer** object requires Computer Vision Toolbox.

Example: 'OutputLayerType','regression'

**ImageInputSize — Size of input images**
vector of two or three numerical values

Size of the input images for the network, specified as a vector of two or three numerical values corresponding to [height,width] for grayscale images and [height,width,channels] for color images, respectively. The network uses this information when the **modelfile** does not specify the input size.

Example: 'ImageInputSize',[28 28]

**Classes — Classes of the output layer**
'auto' (default) | categorical vector | string array | cell array of character vectors
Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If you specify a string array or cell array of character vectors str, then the software sets the classes of the output layer to categorical(str,str). If Classes is 'auto', then the function sets the classes to categorical(1:N), where N is the number of classes.

Data Types: char | categorical | string | cell

Output Arguments

net — Pretrained Keras network
SeriesNetwork object | DAGNetwork object

Pretrained Keras network, returned as one of the following:

• If the Keras network is of type Sequential, then net is a SeriesNetwork object.
• If the Keras network is of type Model, then net is a DAGNetwork object.

Tips

• importKerasNetwork can import a network with the following Keras layer types, with some limitations. If the network contains any other type of layer, then the software returns an error message. To import only the network architecture and weights, use the importKerasLayers function.
### Supported Keras Layers

<table>
<thead>
<tr>
<th>Keras Layer</th>
<th>Deep Learning Toolbox Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>additionLayer</td>
</tr>
<tr>
<td>Activation, with activation names:</td>
<td>Layers:</td>
</tr>
<tr>
<td>'relu'</td>
<td>'reluLayer', 'clippedReluLayer'</td>
</tr>
<tr>
<td>'linear'</td>
<td>None</td>
</tr>
<tr>
<td>'softmax'</td>
<td>'softmaxLayer'</td>
</tr>
<tr>
<td>'sigmoid'</td>
<td>'nnet.keras.layer.SigmoidLayer'</td>
</tr>
<tr>
<td>'tanh'</td>
<td>'nnet.cnn.layer.TanhLayer'</td>
</tr>
<tr>
<td>Advanced activations:</td>
<td>Layers:</td>
</tr>
<tr>
<td>Softmax</td>
<td>'softmaxLayer'</td>
</tr>
<tr>
<td>ReLU</td>
<td>'reluLayer', 'clippedReluLayer'</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td>'leakyReluLayer'</td>
</tr>
<tr>
<td>AveragePooling2D</td>
<td>averagePooling2dLayer</td>
</tr>
<tr>
<td>BatchNormalization</td>
<td>batchNormalizationLayer</td>
</tr>
<tr>
<td>Bidirectional(LSTM(__))</td>
<td>bilstmLayer</td>
</tr>
<tr>
<td>Concatenate</td>
<td>depthConcatenationLayer</td>
</tr>
<tr>
<td>Conv2D</td>
<td>convolution2dLayer</td>
</tr>
<tr>
<td>Conv2DTranspose</td>
<td>transposedConv2dLayer</td>
</tr>
<tr>
<td>DepthwiseConv2D</td>
<td>groupedConvolution2dLayer</td>
</tr>
<tr>
<td>Dense</td>
<td>fullyConnectedLayer</td>
</tr>
<tr>
<td>Dropout</td>
<td>dropoutLayer</td>
</tr>
<tr>
<td>Embedding</td>
<td>wordEmbeddingLayer (Text Analytics Toolbox™)</td>
</tr>
<tr>
<td>Flatten</td>
<td>nnet.keras.layer.FlattenCStyleLayer</td>
</tr>
<tr>
<td>GlobalAveragePooling2D</td>
<td>nnet.cnn.layer.GlobalAveragePooling2DLayer</td>
</tr>
</tbody>
</table>
Keras Layer | Deep Learning Toolbox Layer
--- | ---
Input | imageInputLayer
LeakyReLU | leakyReluLayer
LSTM | lstmLayer
MaxPooling2D | maxPooling2dLayer
SeparableConv2D | groupedConvolution2dLayer and convolution2dLayer
ZeroPadding2D | nnet.keras.layer.ZeroPadding2DLayer

• importKerasNetwork supports the following Keras loss functions:
  • mean_squared_error
  • categorical_crossentropy
  • sparse_categorical_crossentropy
  • binary_crossentropy

• To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way as the images used to train the imported model were preprocessed. Resizing images, subtracting the average image, and converting the images from RGB to BGR format are the most common preprocessing operations.
  • To resize images, use imresize. For example, imresize(im,[227 227]).
  • To convert images from RGB to BGR format, use flip. For example, flip(im,3).

For more information on preprocessing images for training and prediction, see “Preprocess Images for Deep Learning”.

Compatibility Considerations

'ClassNames' option will be removed

*Not recommended starting in R2018b*

'ClassNames' will be removed. Use 'Classes' instead. To update your code, replace all instances of 'ClassNames' with 'Classes'. There are some differences between the
corresponding properties in classification output layers that require additional updates to your code.

The ClassNames property of a classification output layer is a cell array of character vectors. The Classes property is a categorical array. To use the value of Classes with functions that require cell array input, convert the classes using the cellstr function.

References


See Also

exportONNXNetwork | importCaffeLayers | importCaffeNetwork | importKerasLayers | importONNXLayers | importONNXNetwork

Topics

“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”

Introduced in R2017b
importKerasLayers

Import layers from Keras network

Syntax

layers = importKerasLayers(modelfile)
layers = importKerasLayers(modelfile,Name,Value)

Description

layers = importKerasLayers(modelfile) imports the layers of a TensorFlow-Keras network from a model file. The function returns the layers defined in the HDF5 (.h5) or JSON (.json) file given by the file name modelfile.

This function requires the Deep Learning Toolbox Importer for TensorFlow-Keras Models support package. If this support package is not installed, then the function provides a download link.

layers = importKerasLayers(modelfile,Name,Value) imports the layers from a TensorFlow-Keras network with additional options specified by one or more name-value pair arguments.

For example, importKerasLayers(modelfile,'ImportWeights',true) imports the network layers and the weights from the model file modelfile.

Examples


Type importKerasLayers at the command line.
importKerasLayers

If Deep Learning Toolbox Importer for TensorFlow-Keras Models support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install. Check that the installation is successful by importing the layers from the model file 'digitsDAGnet.h5' at the command line.

```matlab
modelfile = 'digitsDAGnet.h5';
layers = importKerasLayers(modelfile)
```

layers =

LayerGraph with properties:

    Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]

If the required support package is installed, then the function returns a LayerGraph object.

**Import Layers from Keras Network and Plot Architecture**

Import the network layers from the model file digitsDAGnet.h5.

```matlab
modelfile = 'digitsDAGnet.h5';
layers = importKerasLayers(modelfile)
```

layers =

LayerGraph with properties:

    Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]

Plot the network architecture.

```matlab
figure
plot(layers)
```
Import Keras Network Layers and Train Network

Specify the network file to import.

`modelfile = 'digitsDAGnet.h5';`

Import network layers.

`layers = importKerasLayers(modelfile)`

`layers =`  
LayerGraph with properties:
Layers: [13×1 nnet.cnn.layer.Layer]
Connections: [13×2 table]

Load a data set for training a classifier to recognize new digits.

```matlab
classifier = fullfile(toolboxdir('nnet'), 'nndemos', 'nndatasets', 'DigitDataset');
imds = imageDatastore(classifier, ...
    'IncludeSubfolders', true, ...
    'LabelSource', 'folder');
```

Partition the dataset into training and test sets.

```matlab
numTrainFiles = 750;
[imdsTrain, imdsTest] = splitEachLabel(imds, numTrainFiles, 'randomize');
```

Set the training options.

```matlab
options = trainingOptions('sgdm', ...
    'MaxEpochs', 10, ...
    'InitialLearnRate', 0.001);
```

Train network using training data.

```matlab
net = trainNetwork(imdsTrain, layers, options);
```

Training on single GPU.

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Iteration</th>
<th>Time Elapsed</th>
<th>Mini-batch Accuracy</th>
<th>Mini-batch Loss</th>
<th>Base Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>00:00:00</td>
<td>5.47%</td>
<td>6.4266</td>
<td>0.001</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>00:00:05</td>
<td>81.25%</td>
<td>0.6227</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>00:00:08</td>
<td>92.19%</td>
<td>0.2386</td>
<td>0.001</td>
</tr>
<tr>
<td>3</td>
<td>150</td>
<td>00:00:12</td>
<td>96.88%</td>
<td>0.0726</td>
<td>0.001</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>00:00:16</td>
<td>97.66%</td>
<td>0.0871</td>
<td>0.001</td>
</tr>
<tr>
<td>5</td>
<td>250</td>
<td>00:00:19</td>
<td>99.22%</td>
<td>0.0415</td>
<td>0.001</td>
</tr>
<tr>
<td>6</td>
<td>300</td>
<td>00:00:23</td>
<td>97.66%</td>
<td>0.0725</td>
<td>0.001</td>
</tr>
<tr>
<td>7</td>
<td>350</td>
<td>00:00:27</td>
<td>100.00%</td>
<td>0.0291</td>
<td>0.001</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>00:00:30</td>
<td>99.22%</td>
<td>0.0230</td>
<td>0.001</td>
</tr>
<tr>
<td>8</td>
<td>450</td>
<td>00:00:34</td>
<td>99.22%</td>
<td>0.0337</td>
<td>0.001</td>
</tr>
<tr>
<td>9</td>
<td>500</td>
<td>00:00:38</td>
<td>99.22%</td>
<td>0.0120</td>
<td>0.001</td>
</tr>
<tr>
<td>10</td>
<td>550</td>
<td>00:00:41</td>
<td>100.00%</td>
<td>0.0036</td>
<td>0.001</td>
</tr>
<tr>
<td>10</td>
<td>580</td>
<td>00:00:44</td>
<td>100.00%</td>
<td>0.0059</td>
<td>0.001</td>
</tr>
</tbody>
</table>
```
Run the trained network on the test set that was not used to train the network and predict the image labels (digits).

\[ Y_{\text{Pred}} = \text{classify}(\text{net}, \text{imdsTest}); \]
\[ Y_{\text{Test}} = \text{imdsTest}.\text{Labels}; \]

Calculate the accuracy.

\[ \text{accuracy} = \frac{\text{sum}(Y_{\text{Pred}} == Y_{\text{Test}})}{\text{numel}(Y_{\text{Test}})} \]

\[ \text{accuracy} = 0.9828 \]

**Import Keras Network Architecture and Weights from Same File**

Specify the network file to import layers and weights from.

\[ \text{modelfile} = '\text{digitsDAGnet.h5}'; \]

Import the network architecture and weights from the files you specified. To import the layer weights, specify 'ImportWeights' to be true. The function also imports the layers with their weights from the same HDF5 file.

\[ \text{layers} = \text{importKerasLayers}(\text{modelfile}, '\text{ImportWeights}', \text{true}) \]

\[ \text{layers} = \]
\[ \text{LayerGraph with properties:} \]
\[ \text{Layers: [13\times1 nnet.cnn.layer.Layer]} \]
\[ \text{Connections: [13\times2 table]} \]

View the size of the weights in the second layer.

\[ \text{weights} = \text{layers.Layers(2).Weights}; \]
\[ \text{size(weights)} \]

\[ \text{ans} = 1\times4 \]
\[ \begin{array}{cccc}
  7 & 7 & 1 & 20 \\
\end{array} \]

The function has imported the weights so the layer weights are non-empty.
Import Keras Network Architecture and Weights from Separate Files

Specify the network file to import layers from and the file containing weights.

```matlab
modelfile = 'digitsDAGnet.json';
weights = 'digitsDAGnet.weights.h5';
```

Import the network architecture and weights from the files you specified. The .json file does not include an output layer. Specify the output layer, so that importKerasLayers adds an output layer at the end of the networks architecture.

```matlab
layers = importKerasLayers(modelfile, ... 'ImportWeights',true, ... 'WeightFile',weights, ... 'OutputLayerType','classification')
```

```
layers =
    LayerGraph with properties:
        Layers: [13×1 nnet.cnn.layer.Layer]
        Connections: [13×2 table]
```

Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

**Import Keras Network**

Import the layers from a Keras network model. The network in 'digitsDAGnetwithnoise.h5' classifies images of digits.

```matlab
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

Warning: Unable to import some Keras layers, because they are not yet supported by the
The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using `plot`.

```matlab
figure
plot(lgraph)
title("Imported Network")
```

---

**Replace Placeholder Layers**

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`. 

---
placeholderLayers = findPlaceholderLayers(lgraph)

placeholderLayers = 
  2x1 PlaceholderLayer array with layers:

1  'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
2  'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer

Display the Keras configurations of these layers.

placeholderLayers.KerasConfiguration

ans = struct with fields:
  trainable: 1
    name: 'gaussian_noise_1'
    stddev: 1.5000

ans = struct with fields:
  trainable: 1
    name: 'gaussian_noise_2'
    stddev: 0.7000

Define a custom Gaussian noise layer. To create this layer, save the file
gaussianNoiseLayer.m in the current folder. Then, create two Gaussian noise layers
with the same configurations as the imported Keras layers.

gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');

Replace the placeholder layers with the custom layers using replaceLayer.

lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);

Plot the updated layer graph using plot.

figure
plot(lgraph)
title("Network with Replaced Layers")
Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to 1, 2, ..., \( N \), where \( N \) is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```matlab
lgraph.Layers
```

```plaintext
ans =
    15x1 Layer array with layers:
```

*importKerasLayers*
The classification layer has the name 'ClassificationLayer_activation_1'. View the classification layer and check the Classes property.

cLayer = lgraph.Layers(end)

```
cLayer = 
    ClassificationOutputLayer with properties:

        Name: 'ClassificationLayer_activation_1'
    Classes: 'auto'
        OutputSize: 'auto'

    Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the Classes property of the layer is 'auto', you must specify the classes manually. Set the classes to 0, 1, ..., 9, and then replace the imported classification layer with the new one.

cLayer.Classes = string(0:9)

```
cLayer = 
    ClassificationOutputLayer with properties:

        Name: 'ClassificationLayer_activation_1'
    Classes: [0 1 2 3 4 5 6 7 8 9]
        OutputSize: 10
```
Hyperparameters
   LossFunction: 'crossentropyex'

lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);

Assemble Network

Assemble the layer graph using assembleNetwork. The function returns a DAGNetwork object that is ready to use for prediction.

net = assembleNetwork(lgraph)

net =
    DAGNetwork with properties:
        Layers: [15×1 nnet.cnn.layer.Layer]
        Connections: [15×2 table]

Input Arguments

modelfile — Name of Keras model file
character vector | string scalar

Name of the model file containing the network architecture, and possibly the weights, specified as a character vector or a string scalar. The file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

If modelfile includes

- The network architecture and weights, then it must be in HDF5 (.h5) format.
- Only the network architecture, then it can be in HDF5 or JSON (.json) format.

If modelfile includes only the network architecture, then you can optionally supply the weights using the 'ImportWeights' and 'WeightFile' name-value pair arguments. If you supply the weights, then the weights file must be in HDF5 format.

Example: 'digitsnet.h5'

Data Types: char | string
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:
importKerasLayers(modelfile, 'OutputLayerType', 'classification')

imports the network layers from the model file modelfile and adds an output layer for a classification problem at the end of the Keras layers.

**OutputLayerType — Type of output layer**

'classification' | 'regression' | 'pixelclassification'

Type of the output layer that the function appends to the end of the imported network architecture when modelfile does not specify a loss function, specified as 'classification', 'regression', or 'pixelclassification'. Appending a pixelClassificationLayer object requires Computer Vision Toolbox.

Example: 'OutputLayerType', 'regression'

*ImageInputSize — Size of input images*

vector of two or three numerical values

Size of the input images for the network, specified as a vector of two or three numerical values corresponding to [height, width] for grayscale images and [height, width, channels] for color images, respectively. The network uses this information when the modelfile does not specify the input size.

Example: 'ImageInputSize', [28 28]

*ImportWeights — Indicator to import weights*

false (default) | true

Indicator to import weights as well as the network architecture, specified as either false or true.

- If 'ImportWeights' is true and modelfile includes the weights, then importKerasLayers imports the weights from modelfile, which must have HDF5 (.h5) format.
• If 'ImportWeights' is true and modelfile does not include the weights, then you must specify a separate file that includes weights, using the 'WeightFile' name-value pair argument.

Example: 'ImportWeights',true

Data Types: logical

WeightFile — Weight file name
character vector | string scalar

Weight file name, from which to import weights when modelfile does not include weights, specified as a character vector or a string scalar. To use this name-value pair argument, you also must set 'ImportWeights' to true.

Weight file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: 'WeightFile','weights.h5'

Data Types: char | string

Output Arguments

layers — Network architecture
Layer array object | LayerGraph object

Network architecture, returned as a Layer array object when the Keras network is of type Sequential, or returned as a LayerGraph object when the Keras network is of type Model.

Tips

• importKerasLayers supports the following Keras layer types, with some limitations. If the network contains any other type of layer, then the software inserts a placeholder layer in place of the unsupported layer. To find the names and indices of the unsupported layers in the network, use the findPlaceholderLayers function.
## Supported Keras Layers

<table>
<thead>
<tr>
<th>Keras Layer</th>
<th>Deep Learning Toolbox Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>additionLayer</td>
</tr>
<tr>
<td>Activation, with activation names:</td>
<td>Layers:</td>
</tr>
<tr>
<td>'relu'</td>
<td>• reluLayer, clippedReluLayer</td>
</tr>
<tr>
<td>'linear'</td>
<td>• None</td>
</tr>
<tr>
<td>'softmax'</td>
<td>• softmaxLayer</td>
</tr>
<tr>
<td>'sigmoid'</td>
<td>• nnet.keras.layer.SigmoidLayer</td>
</tr>
<tr>
<td>'tanh'</td>
<td>• nnet.cnn.layer.TanhLayer</td>
</tr>
<tr>
<td>Advanced activations:</td>
<td>Layers:</td>
</tr>
<tr>
<td>Softmax</td>
<td>• softmaxLayer</td>
</tr>
<tr>
<td>ReLU</td>
<td>• reluLayer, clippedReluLayer</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td>• leakyReluLayer</td>
</tr>
<tr>
<td>AveragePooling2D</td>
<td>averagePooling2dLayer</td>
</tr>
<tr>
<td>BatchNormalization</td>
<td>batchNormalizationLayer</td>
</tr>
<tr>
<td>Bidirectional(LSTM(__))</td>
<td>bilstmLayer</td>
</tr>
<tr>
<td>Concatenate</td>
<td>depthConcatenationLayer</td>
</tr>
<tr>
<td>Conv2D</td>
<td>convolution2dLayer</td>
</tr>
<tr>
<td>Conv2DTranspose</td>
<td>transposedConv2dLayer</td>
</tr>
<tr>
<td>DepthwiseConv2D</td>
<td>groupedConvolution2dLayer</td>
</tr>
<tr>
<td>Dense</td>
<td>fullyConnectedLayer</td>
</tr>
<tr>
<td>Dropout</td>
<td>dropoutLayer</td>
</tr>
<tr>
<td>Embedding</td>
<td>wordEmbeddingLayer (Text Analytics Toolbox)</td>
</tr>
<tr>
<td>Flatten</td>
<td>nnet.keras.layer.FlattenCStyleLayer</td>
</tr>
<tr>
<td>GlobalAveragePooling2D</td>
<td>nnet.cnn.layer.GlobalAveragePooling2DLayer</td>
</tr>
</tbody>
</table>
importKerasLayers

<table>
<thead>
<tr>
<th>Keras Layer</th>
<th>Deep Learning Toolbox Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>imageInputLayer</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td>leakyReluLayer</td>
</tr>
<tr>
<td>LSTM</td>
<td>lstmLayer</td>
</tr>
<tr>
<td>MaxPooling2D</td>
<td>maxPooling2dLayer</td>
</tr>
<tr>
<td>SeparableConv2D</td>
<td>groupedConvolution2dLayer and convolution2dLayer</td>
</tr>
<tr>
<td>ZeroPadding2D</td>
<td>nnet.keras.layer.ZeroPadding2DLayer</td>
</tr>
</tbody>
</table>

- You can replace a placeholder layer with a new layer that you define.
  - If the network is a series network, then replace the layer in the array directly. For example, `layer(2) = newLayer;`.
  - If the network is a DAG network, then replace the layer using `replaceLayer`. For an example, see “Assemble Network from Pretrained Keras Layers” on page 1-1398.
- `importKerasLayers` supports the following Keras loss functions:
  - `mean_squared_error`
  - `categorical_crossentropy`
  - `sparse_categorical_crossentropy`
  - `binary_crossentropy`

References


See Also

assembleNetwork | exportONNXNetwork | findPlaceholderLayers | importCaffeLayers | importCaffeNetwork | importKerasNetwork | importONNXLayers | importONNXNetwork | replaceLayer

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”

**Introduced in R2017b**
findPlaceholderLayers

Find placeholder layers in network architecture imported from Keras or ONNX

Syntax

`placeholderLayers = findPlaceholderLayers(importedLayers)`  
`[placeholderLayers,indices] = findPlaceholderLayers(importedLayers)`

Description

`placeholderLayers = findPlaceholderLayers(importedLayers)` returns all placeholder layers that exist in the network architecture `importedLayers` imported by the `importKerasLayers` or `importONNXLayers` functions, or created by the `functionToLayerGraph` function. Placeholder layers are the layers that these functions insert in place of layers that are not supported by Deep Learning Toolbox.

To use with an imported network, this function requires either the Deep Learning Toolbox Importer for TensorFlow-Keras Models support package or the Deep Learning Toolbox Converter for ONNX™ Model Format support package.

`[placeholderLayers,indices] = findPlaceholderLayers(importedLayers)` also returns the indices of the placeholder layers.

Examples

Find and Explore Placeholder Layers

Specify the Keras network file to import layers from.

`modelfile = 'digitsDAGnetwithnoise.h5';`

Import the network architecture. The network includes some layer types that are not supported by Deep Learning Toolbox. The `importKerasLayers` function replaces each unsupported layer with a placeholder layer and returns a warning message.
lgraph = importKerasLayers(modelfile)

Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learning Toolbox. They have been replaced by placeholder layers. To find these layers, call the function findPlaceholderLayers on the returned object.
> In nnet.internal.cnn.keras.importKerasLayers (line 26)
In importKerasLayers (line 102)

lgraph =

LayerGraph with properties:

    Layers: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]

Display the imported layers of the network. Two placeholder layers replace the Gaussian noise layers in the Keras network.

lgraph.Layers

ans =

15x1 Layer array with layers:

    1   'input_1'                            Image Input             28x28x1 images
    2   'conv2d_1'                           Convolution             20 7x7 convolutions with stride [1  1] and padding 'same'
    3   'conv2d_1_relu'                      ReLU                    ReLU
    4   'conv2d_2'                           Convolution             20 3x3 convolutions with stride [1  1] and padding 'same'
    5   'conv2d_2_relu'                      ReLU                    ReLU
    6   'gaussian_noise_1'                   PLACEHOLDER LAYER       Placeholder for 'GaussianNoise' Keras layer
    7   'gaussian_noise_2'                   PLACEHOLDER LAYER       Placeholder for 'GaussianNoise' Keras layer
    8   'max_pooling2d_1'                    Max Pooling             2x2 max pooling with stride [2  2] and padding 'same'
    9   'max_pooling2d_2'                    Max Pooling             2x2 max pooling with stride [2  2] and padding 'same'
   10   'flatten_1'                          Flatten C-style         Flatten activations into 1D assuming C-style (row-major) order
   11   'flatten_2'                          Flatten C-style         Flatten activations into 1D assuming C-style (row-major) order
   12   'concatenate_1'                      Depth concatenation     Depth concatenation of 2 inputs
   13   'dense_1'                            Fully Connected         10 fully connected layer
   14   'activation_1_softmax'               Softmax                 softmax
   15   'ClassificationLayer_activation_1'   Classification Output   crossentropyex

Find the placeholder layers using findPlaceholderLayers. The output argument contains the two placeholder layers that importKerasLayers inserted in place of the Gaussian noise layers of the Keras network.

placeholders = findPlaceholderLayers(lgraph)
placeholders =
    2x1 PlaceholderLayer array with layers:
    1  'gaussian_noise_1' PLACEHOLDER LAYER  Placeholder for 'GaussianNoise' Keras layer
    2  'gaussian_noise_2' PLACEHOLDER LAYER  Placeholder for 'GaussianNoise' Keras layer

Display the configuration of each placeholder layer.

    gaussian1.KerasConfiguration
    gaussian2.KerasConfiguration

ans =
    struct with fields:
        trainable: 1
        name: 'gaussian_noise_1'
        stddev: 1.5000

ans =
    struct with fields:
        trainable: 1
        name: 'gaussian_noise_2'
        stddev: 0.7000

Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in 'digitsDAGnetwithnoise.h5' classifies images of digits.

filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learning Toolbox. They have been replaced by placeholder layers. To find these layers, call the function findPlaceholderLayers on the returned object.

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The importKerasLayers function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using plot.

```matlab
figure
plot(lgraph)
title("Imported Network")
```
Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```matlab
placeholderLayers = findPlaceholderLayers(lgraph)
```

```matlab
placeholderLayers =
  2x1 PlaceholderLayer array with layers:
    1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
    2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.

```matlab
placeholderLayers.KerasConfiguration
```

```matlab
ans = struct with fields:
   trainable: 1
       name: 'gaussian_noise_1'
        stddev: 1.5000

ans = struct with fields:
   trainable: 1
       name: 'gaussian_noise_2'
        stddev: 0.7000
```

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

```matlab
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```matlab
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```matlab
figure
plot(lgraph)
title("Network with Replaced Layers")
```
Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to 1, 2, ..., N, where N is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```matlab
lgraph.Layers
```

```plaintext
ans =
    15x1 Layer array with layers:
```
The classification layer has the name 'ClassificationLayer_activation_1'. View the classification layer and check the Classes property.

cLayer = lgraph.Layers(end)

cLayer = ClassificationOutputLayer with properties:

    Name: 'ClassificationLayer_activation_1'
    Classes: 'auto'
    OutputSize: 'auto'

    Hyperparameters
    LossFunction: 'crossentropyex'

Because the Classes property of the layer is 'auto', you must specify the classes manually. Set the classes to 0, 1, ..., 9, and then replace the imported classification layer with the new one.

```matlab
cLayer.Classes = string(0:9)

    cLayer = ClassificationOutputLayer with properties:

    Name: 'ClassificationLayer_activation_1'
    Classes: [0 1 2 3 4 5 6 7 8 9]
    OutputSize: 10
```
Hyperparameters
LossFunction: 'crossentropyex'

lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);

Assemble Network
Assemble the layer graph using assembleNetwork. The function returns a DAGNetwork object that is ready to use for prediction.

net = assembleNetwork(lgraph)

net = DAGNetwork with properties:
    Layers: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]

Input Arguments

importedLayers — Network architecture imported from Keras or ONNX or created by functionToLayerGraph
Layer array | LayerGraph object

Network architecture imported from Keras or ONNX or created by functionToLayerGraph, specified as a Layer array or LayerGraph object.

Output Arguments

placeholderLayers — All placeholder layers in network architecture
array of PlaceholderLayer objects

All placeholder layers in the network architecture, returned as an array of PlaceholderLayer objects.

indices — Indices of placeholder layers
vector

Indices of placeholder layers, returned as a vector.
• If `importedLayers` is a layer array, then `indices` are the indices of the placeholder layers in `importedLayers`.

• If `importedLayers` is a `LayerGraph` object, then `indices` are the indices of the placeholder layers in `importedLayers.Layers`.

If you remove a layer from or add a layer to a `Layer` array or `LayerGraph` object, then the indices of the other layers in the object can change. You must use `findPlaceholderLayers` again to find the updated indices of the rest of the placeholder layers.

**Tips**

• If you have installed Deep Learning Toolbox Importer for TensorFlow-Keras Models and `findPlaceholderLayers` is unable to find placeholder layers created when importing an ONNX network, then try updating the Deep Learning Toolbox Importer for TensorFlow-Keras Models support package in the Add-On Explorer.

**See Also**

`PlaceholderLayer` | `assembleNetwork` | `functionToLayerGraph` | `importKerasLayers` | `importONNXLayers` | `replaceLayer`

**Topics**

“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”
“Assemble Network from Pretrained Keras Layers”

**Introduced in R2017b**
**PlaceholderLayer**

Layer replacing an unsupported Keras layer, ONNX layer, or unsupported functionality from functionToLayerGraph

**Description**

PlaceholderLayer is a layer that importKerasLayers and importONNXLayers insert into a layer array or layer graph in place of an unsupported Keras or ONNX layer. It can also represent unsupported functionality from functionToLayerGraph.

**Creation**

Importing layers from a Keras or ONNX network that has layers that are not supported by Deep Learning Toolbox creates PlaceholderLayer objects. Also, when you create a layer graph using functionToLayerGraph, unsupported functionality leads to PlaceholderLayer objects.

**Properties**

**Name — Layer name**

character vector | string scalar

Layer name, specified as a character vector or a string scalar.

Data Types: char | string

**Description — Layer description**

character vector | string scalar

Layer description, specified as a character vector or a string scalar.

Data Types: char | string

**Type — Layer type**

character vector | string scalar
Layer type, specified as a character vector or a string scalar.
Data Types: char | string

**KerasConfiguration — Keras configuration of layer**
structure

Keras configuration of a layer, specified as a structure. The fields of the structure depend on the layer type.

**Note** This property only exists if the layer was created when importing a Keras network.

Data Types: struct

**ONNXNode — ONNX configuration of layer**
structure

ONNX configuration of a layer, specified as a structure. The fields of the structure depend on the layer type.

**Note** This property only exists if the layer was created when importing an ONNX network.

Data Types: struct

**Weights — Imported weights**
structure

Imported weights, specified as a structure.
Data Types: struct

### Examples

**Find and Explore Placeholder Layers**

Specify the Keras network file to import layers from.
modelfile = 'digitsDAGnetwithnoise.h5';

Import the network architecture. The network includes some layer types that are not supported by Deep Learning Toolbox. The importKerasLayers function replaces each unsupported layer with a placeholder layer and returns a warning message.

lgraph = importKerasLayers(modelfile)

Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learning Toolbox. They have been replaced by placeholder layers. To find these layers, call the function findPlaceholderLayers on the returned object.
> In nnet.internal.cnn.keras.importKerasLayers (line 26)
  In importKerasLayers (line 102)

lgraph =

LayerGraph with properties:

  Layers: [15x1 nnet.cnn.layer.Layer]
  Connections: [15x2 table]

Display the imported layers of the network. Two placeholder layers replace the Gaussian noise layers in the Keras network.

lgraph.Layers

ans =

15x1 Layer array with layers:

1  'input_1' Image Input 28x28x1 images
2  'conv2d_1' Convolution 20 7x7 convolutions with stride [1 1] and padding 'same'
3  'conv2d_1_relu' ReLU ReLU
4  'conv2d_2' Convolution 20 3x3 convolutions with stride [1 1] and padding 'same'
5  'conv2d_2_relu' ReLU ReLU
6  'gaussian_noise_1' PLACEHOLDER LAYER Placeholder for 'GaussianNoise' Keras layer
7  'gaussian_noise_2' PLACEHOLDER LAYER Placeholder for 'GaussianNoise' Keras layer
8  'max_pooling2d_1' Max Pooling 2x2 max pooling with stride [2 2] and padding 'same'
9  'max_pooling2d_2' Max Pooling 2x2 max pooling with stride [2 2] and padding 'same'
10 'flatten_1' Flatten C-style Flatten activation
11 'flatten_2' Flatten C-style Flatten activation
12 'concatenate_1' Depth concatenation Depth concatenation
13 'dense_1' Fully Connected 10 fully connected layer
14 'activation_1_softmax' Softmax softmax
15 'ClassificationLayer_activation_1' Classification Output crossentropyex
Find the placeholder layers using `findPlaceholderLayers`. The output argument contains the two placeholder layers that `importKerasLayers` inserted in place of the Gaussian noise layers of the Keras network.

```matlab
placeholders = findPlaceholderLayers(lgraph)

placeholders =  
2x1 PlaceholderLayer array with layers:
  
1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer  
2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the configuration of each placeholder layer.

```matlab
gaussian1.KerasConfiguration
gaussian2.KerasConfiguration

ans =  
struct with fields:
  
  trainable: 1
  name: 'gaussian_noise_1'
  stddev: 1.5000

ans =  
struct with fields:
  
  trainable: 1
  name: 'gaussian_noise_2'
  stddev: 0.7000
```

**Assemble Network from Pretrained Keras Layers**

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.
Import Keras Network

Import the layers from a Keras network model. The network in 'digitsDAGnetwithnoise.h5' classifies images of digits.

```matlab
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learning Toolbox. The importKerasLayers function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using plot.

```matlab
figure
plot(lgraph)
title("Imported Network")
```
Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```plaintext
placeholderLayers = findPlaceholderLayers(lgraph)
```

```plaintext
placeholderLayers =
   2x1 PlaceholderLayer array with layers:

  1  'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
  2  'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.
Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

```matlab
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```matlab
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```matlab
figure
plot(lgraph)
title("Network with Replaced Layers")
```
Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to 1, 2, ..., N, where N is the number of classes.

Find the index of the classification layer by viewing the Layers property of the layer graph.

lgraph.Layers

ans =
15x1 Layer array with layers:
The classification layer has the name 'ClassificationLayer_activation_1'. View the classification layer and check the Classes property.

cLayer = lgraph.Layers(end)

cLayer =
    ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
               Classes: 'auto'
          OutputSize: 'auto'

     Hyperparameters
     LossFunction: 'crossentropyex'

Because the Classes property of the layer is 'auto', you must specify the classes manually. Set the classes to 0, 1, ..., 9, and then replace the imported classification layer with the new one.

cLayer.Classes = string(0:9)

cLayer =
    ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
               Classes: [0 1 2 3 4 5 6 7 8 9]
          OutputSize: 10
Hyperparameters
  LossFunction: 'crossentropyex'

lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);

**Assemble Network**

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

net = assembleNetwork(lgraph)

net = DAGNetwork with properties:
  Layers: [15×1 nnet.cnn.layer.Layer]
  Connections: [15×2 table]

**See Also**
assembleNetwork | findPlaceholderLayers | functionToLayerGraph | importKerasLayers | importONNXLayers

**Topics**
“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”
“Assemble Network from Pretrained Keras Layers”

**Introduced in R2017b**
**checkLayer**

Check validity of custom layer

**Syntax**

```matlab
checkLayer(layer,validInputSize)
checkLayer(layer,validInputSize,'ObservationDimension',dim)
```

**Description**

*checkLayer(layer,validInputSize)* checks the validity of a custom layer using generated data of the sizes in *validInputSize*. For layers with a single input, set *validInputSize* to a typical size of input data to the layer. For layers with multiple inputs, set *validInputSize* to a cell array of typical sizes, where each element corresponds to a layer input.

*checkLayer(layer,validInputSize,'ObservationDimension',dim)* specifies the dimension of the data that corresponds to observations. If you specify this parameter, then the function checks the layer for both a single observation and multiple observations.

**Examples**

**Check Layer Validity**

Check the validity of the example custom layer *preluLayer*.

Define a custom PReLU layer. To create this layer, save the file *preluLayer.m* in the current folder.

Create an instance of the layer and check that it is valid using *checkLayer*. Set the valid input size to the typical size of a single observation input to the layer. For a single input, the layer expects observations of size *h*-by-*w*-by-*, where *h*, *w*, and *c* are the height, width, and number of channels of the previous layer output, respectively.
Specify `validInputSize` as the typical size of an input array.

```matlab
layer = preluLayer(20,'prelu');
validInputSize = [5 5 20];
checkLayer(layer,validInputSize)
```

Skipping multi-observation tests. To enable tests with multiple observations, specify the `'ObservationDimension'` option in `checkLayer`. For 2-D image data, set `'ObservationDimension'` to 4. For 3-D image data, set `'ObservationDimension'` to 5. For sequence data, set `'ObservationDimension'` to 2.

Skipping GPU tests. No compatible GPU device found.

Running `nnet.checkLayer.TestLayerWithoutBackward`

```
........
```

Test Summary:
  9 Passed, 0 Failed, 0 Incomplete, 8 Skipped.
  Time elapsed: 0.25217 seconds.

The results show the number of passed, failed, and skipped tests. If you do not specify the `'ObservationsDimension'` option, or do not have a GPU, then the function skips the corresponding tests.

**Check Multiple Observations**

For multi-observation input, the layer expects an array of observations of size `h`-by-`w`-by-`c`-by-`N`, where `h`, `w`, and `c` are the height, width, and number of channels, respectively, and `N` is the number of observations.

To check the layer validity for multiple observations, specify the typical size of an observation and set `'ObservationDimension'` to 4.

```matlab
layer = preluLayer(20,'prelu');
validInputSize = [5 5 20];
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

Skipping GPU tests. No compatible GPU device found.

Running `nnet.checkLayer.TestLayerWithoutBackward`

```
........
```

Test Summary:
  9 Passed, 0 Failed, 0 Incomplete, 8 Skipped.
  Time elapsed: 0.25217 seconds.
Test Summary:
13 Passed, 0 Failed, 0 Incomplete, 4 Skipped.
Time elapsed: 0.20768 seconds.

In this case, the function does not detect any issues with the layer.

Input Arguments

layer — Custom layer
nnet.layer.Layer object | nnet.layer.ClassificationLayer object | nnet.layer.RegressionLayer object

Custom layer, specified as an nnet.layer.Layer object, nnet.layer.ClassificationLayer object, or nnet.layer.RegressionLayer object. For an example showing how to define your own custom layer, see “Define Custom Deep Learning Layer with Learnable Parameters”.

validInputSize — Valid input sizes
vector of positive integers | cell array of vectors of positive integers

Valid input sizes of the layer, specified as a vector of positive integers or cell array of vectors of positive integers.

- For layers with a single input, specify validInputSize as a vector of integers corresponding to the dimensions of the input data. For example, [5 5 10] corresponds to valid input data of size 5-by-5-by-10.
- For layers with multiple inputs, specify validInputSize as a cell array of vectors, where each vector corresponds to a layer input and the elements of the vectors correspond to the dimensions of the corresponding input data. For example, {[24 24 20],[24 24 10]} corresponds to the valid input sizes of two inputs, where 24-by-24-by-20 is a valid input size for the first input and 24-by-24-by-10 is a valid input size for the second input.

For more information, see “Layer Input Sizes” on page 1-1431.

For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

Example: [5 5 10]
Example: \{[24 24 20],[24 24 10]\}

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | cell

**dim — Observation dimension**
positive integer

Observation dimension, specified as a positive integer.

The observation dimension specifies which dimension of the layer input data corresponds to observations. For example, if the layer expects input data is of size \(h\)-by-\(w\)-by-\(c\)-by-\(N\), where \(h\), \(w\), and \(c\) correspond to the height, width, and number of channels of the input data, respectively, and \(N\) corresponds to the number of observations, then the observation dimension is 4. For more information, see “Layer Input Sizes” on page 1-1431.

If you specify the observation dimension, then the checkLayer function checks that the layer functions are valid using generated data with mini-batches of size 1 and 2. If you do not specify the observation dimension, then the function skips the corresponding tests.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**More About**

**Layer Input Sizes**

For each layer, the valid input size and the observation dimension depend on the output of the previous layer.

**Intermediate Layers**

For intermediate layers (layers of type nnet.layer.Layer), the valid input size and the observation dimension depend on the type of data input to the layer. For layers with a single input, specify `validInputSize` as a vector of integers corresponding to the dimensions of the input data. For layers with multiple inputs, specify `validInputSize` as a cell array of vectors, where each vector corresponds to a layer input and the elements of the vectors correspond to the dimensions of the corresponding input data. For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.
<table>
<thead>
<tr>
<th>Layer Input</th>
<th>Input Size</th>
<th>Observation Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D images</td>
<td>$h$-by-$w$-by-$c$-by-$N$, where $h$, $w$, and $c$ correspond to the height, width, and number of channels of the images respectively, and $N$ is the number of observations.</td>
<td>4</td>
</tr>
<tr>
<td>3-D images</td>
<td>$h$-by-$w$-by-$D$-by-$c$-by-$N$, where $h$, $w$, $D$, and $c$ correspond to the height, width, depth, and number of channels of the 3-D images respectively, and $N$ is the number of observations.</td>
<td>5</td>
</tr>
<tr>
<td>Vector sequences</td>
<td>$c$-by-$N$-by-$S$, where $c$ is the number of features of the sequences, $N$ is the number of observations, and $S$ is the sequence length.</td>
<td>2</td>
</tr>
<tr>
<td>2-D image sequences</td>
<td>$h$-by-$w$-by-$c$-by-$N$-by-$S$, where $h$, $w$, and $c$ correspond to the height, width, and number of channels of the images respectively, $N$ is the number of observations, and $S$ is the sequence length.</td>
<td>4</td>
</tr>
<tr>
<td>3-D image sequences</td>
<td>$h$-by-$w$-by-$d$-by-$c$-by-$N$-by-$S$, where $h$, $w$, $d$, and $c$ correspond to the height, width, depth, and number of channels of the 3-D images respectively, $N$ is the number of observations, and $S$ is the sequence length.</td>
<td>5</td>
</tr>
</tbody>
</table>
For example, for 2-D image classification problems, set `validInputSize` to `[h w c]`, where `h`, `w`, and `c` correspond to the height, width, and number of channels of the images, respectively, and `'ObservationDimension'` to 4.

**Output Layers**

For output layers (layers of type `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`), set `validInputSize` to the typical size of a single input observation `Y` to the layer.

For classification problems, the valid input size and the observation dimension of `Y` depend on the type of problem:

<table>
<thead>
<tr>
<th>Classification Task</th>
<th>Input Size</th>
<th>Observation Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D image classification</td>
<td>1-by-1-by-<code>K</code>-by-<code>N</code>, where <code>K</code> is the number of classes and <code>N</code> is the number of observations.</td>
<td>4</td>
</tr>
<tr>
<td>3-D image classification</td>
<td>1-by-1-by-1-by-<code>K</code>-by-<code>N</code>, where <code>K</code> is the number of classes and <code>N</code> is the number of observations.</td>
<td>5</td>
</tr>
<tr>
<td>Sequence-to-label classification</td>
<td><code>K</code>-by-<code>N</code>, where <code>K</code> is the number of classes and <code>N</code> is the number of observations.</td>
<td>2</td>
</tr>
<tr>
<td>Sequence-to-sequence classification</td>
<td><code>K</code>-by-<code>N</code>-by-<code>S</code>, where <code>K</code> is the number of classes, <code>N</code> is the number of observations, and <code>S</code> is the sequence length.</td>
<td>2</td>
</tr>
</tbody>
</table>

For example, for 2-D image classification problems, set `validInputSize` to `[1 1 K]`, where `K` is the number of classes, and `'ObservationDimension'` to 4.

For regression problems, the dimensions of `Y` also depend on the type of problem. The following table describes the dimensions of `Y`. 

1-1433
<table>
<thead>
<tr>
<th>Regression Task</th>
<th>Input Size</th>
<th>Observation Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D image regression</td>
<td>1-by-1-by-(R)-by-(N), where (R) is the number of responses and (N) is the number of observations.</td>
<td>4</td>
</tr>
<tr>
<td>2-D Image-to-image regression</td>
<td>(h)-by-(w)-by-(c)-by-(N), where (h), (w), and (c) are the height, width, and number of channels of the output respectively, and (N) is the number of observations.</td>
<td>4</td>
</tr>
<tr>
<td>3-D image regression</td>
<td>1-by-1-by-1-by-(R)-by-(N), where (R) is the number of responses and (N) is the number of observations.</td>
<td>5</td>
</tr>
<tr>
<td>3-D Image-to-image regression</td>
<td>(h)-by-(w)-by-(d)-by-(c)-by-(N), where (h), (w), (d), and (c) are the height, width, depth, and number of channels of the output respectively, and (N) is the number of observations.</td>
<td>5</td>
</tr>
<tr>
<td>Sequence-to-one regression</td>
<td>(R)-by-(N), where (R) is the number of responses and (N) is the number of observations.</td>
<td>2</td>
</tr>
<tr>
<td>Sequence-to-sequence regression</td>
<td>(R)-by-(N)-by-(S), where (R) is the number of responses, (N) is the number of observations, and (S) is the sequence length.</td>
<td>2</td>
</tr>
</tbody>
</table>

For example, for 2-D image regression problems, set `validInputSize` to `[1 1 \(R\)]`, where \(R\) is the number of responses, and 'ObservationDimension' to 4.
Algorithms

List of Tests

The `checkLayer` function checks the validity of a custom layer by performing a series of tests, described in these tables. For more information on the tests used by `checkLayer`, see “Check Custom Layer Validity”.

Intermediate Layers

The `checkLayer` function uses these tests to check the validity of custom intermediate layers (layers of type `nnet.layer.Layer`).

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>functionsSyntaxesAreCorrect</code></td>
<td>The syntaxes of the layer functions are correctly defined.</td>
</tr>
<tr>
<td><code>predictDoesNotError</code></td>
<td><code>predict</code> does not error.</td>
</tr>
<tr>
<td><code>forwardDoesNotError</code></td>
<td>When specified, <code>forward</code> does not error.</td>
</tr>
<tr>
<td><code>forwardPredictAreConsistentInSize</code></td>
<td>When <code>forward</code> is specified, <code>forward</code> and <code>predict</code> output values of the same size.</td>
</tr>
<tr>
<td><code>backwardDoesNotError</code></td>
<td>When specified, <code>backward</code> does not error.</td>
</tr>
<tr>
<td><code>backwardIsConsistentInSize</code></td>
<td>When backward is specified, the outputs of <code>backward</code> are consistent in size:</td>
</tr>
<tr>
<td></td>
<td>• The derivatives with respect to each input are the same size as the corresponding input.</td>
</tr>
<tr>
<td></td>
<td>• The derivatives with respect to each learnable parameter are the same size as the corresponding learnable parameter.</td>
</tr>
<tr>
<td><code>predictIsConsistentInType</code></td>
<td>The outputs of <code>predict</code> are consistent in type with the inputs.</td>
</tr>
<tr>
<td><code>forwardIsConsistentInType</code></td>
<td>When <code>forward</code> is specified, the outputs of <code>forward</code> are consistent in type with the inputs.</td>
</tr>
</tbody>
</table>
When backward is specified, the outputs of backward are consistent in type with the inputs.

When backward is specified, the gradients computed in backward are consistent with the numerical gradients.

When backward is not specified, the derivatives can be computed using automatic differentiation.

The tests predictIsConsistentInType, forwardIsConsistentInType, and backwardIsConsistentInType also check for GPU compatibility. To execute the layer functions on a GPU, the functions must support inputs and outputs of type gpuArray with the underlying data type single.

Output Layers

The checkLayer function uses these tests to check the validity of custom output layers (layers of type nnet.layer.ClassificationLayer or nnet.layer.RegressionLayer).

When backwardLoss is specified, the output of backwardLoss is consistent in size: dLdY is the same size as the predictions Y.

The output of forwardLoss is consistent in type: loss is the same type as the predictions Y.
<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backwardLossIsConsistentInType</td>
<td>When <code>backwardLoss</code> is specified, the output of <code>backwardLoss</code> is consistent in type: <code>dLdY</code> must be the same type as the predictions <code>Y</code>.</td>
</tr>
<tr>
<td>gradientsAreNumericallyCorrect</td>
<td>When <code>backwardLoss</code> is specified, the gradients computed in <code>backwardLoss</code> are numerically correct.</td>
</tr>
<tr>
<td>backwardPropagationDoesNotError</td>
<td>When <code>backwardLoss</code> is not specified, the derivatives can be computed using automatic differentiation.</td>
</tr>
</tbody>
</table>

The `forwardLossIsConsistentInType` and `backwardLossIsConsistentInType` tests also check for GPU compatibility. To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

### See Also
- `analyzeNetwork`
- `trainNetwork`
- `trainingOptions`

### Topics
- “Check Custom Layer Validity”
- “Define Custom Deep Learning Layers”
- “Define Custom Deep Learning Layer with Learnable Parameters”
- “Define Custom Deep Learning Layer with Multiple Inputs”
- “Define Custom Classification Output Layer”
- “Define Custom Weighted Classification Layer”
- “Define Custom Regression Output Layer”
- “List of Deep Learning Layers”
- “Deep Learning Tips and Tricks”

### Introduced in R2018a
replaceLayer

Replace layer in layer graph

Syntax

newlgraph = replaceLayer(lgraph,layerName,larray)
newlgraph = replaceLayer(lgraph,layerName,larray,'ReconnectBy',mode)

Description

newlgraph = replaceLayer(lgraph,layerName,larray) replaces the layer layerName in the layer graph lgraph with the layers in larray.

replaceLayer connects the layers in larray sequentially and connects larray into the layer graph.

newlgraph = replaceLayer(lgraph,layerName,larray,'ReconnectBy',mode) additionally specifies the method of reconnecting layers.

Examples

Replace Layer in Layer Graph

Define a simple network architecture and plot it.

layers = [  
    imageInputLayer([28 28 1],'Name','input')  
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')  
    reluLayer('Name','relu_1')  
    additionLayer(2,'Name','add')  
    fullyConnectedLayer(10,'Name','fc')  
    softmaxLayer('Name','softmax')  
    classificationLayer('Name','classoutput')];
Replace the ReLU layer in the network with a batch normalization layer followed by a leaky ReLU layer.

```matlab
larray = [batchNormalizationLayer('Name','BN1')
    leakyReluLayer('Name','leakyRelu_1','Scale',0.1)];
lgraph = replaceLayer(lgraph,'relu_1',larray);
plot(lgraph)
```
Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in 'digitsDAGnetwithnoise.h5' classifies images of digits.
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);

Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learning Toolbox. The importKerasLayers function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using plot.

figure
plot(lgraph)
title("Imported Network")
Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using findPlaceholderLayers.

```matlab
placeholderLayers = findPlaceholderLayers(lgraph)
```

**placeholderLayers** =

2x1 PlaceholderLayer array with layers:

1  'gaussian_noise_1'  PLACEHOLDER LAYER  Placeholder for 'GaussianNoise' Keras layer
2  'gaussian_noise_2'  PLACEHOLDER LAYER  Placeholder for 'GaussianNoise' Keras layer

Display the Keras configurations of these layers.

```matlab
placeholderLayers.KerasConfiguration
```

ans = struct with fields:

- `trainable`: 1
- `name`: 'gaussian_noise_1'
- `stddev`: 1.5000

ans = struct with fields:

- `trainable`: 1
- `name`: 'gaussian_noise_2'
- `stddev`: 0.7000

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

```matlab
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using replaceLayer.

```matlab
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using plot.

```matlab
figure
plot(lgraph)
title("Network with Replaced Layers")
```
Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to 1, 2, ..., N, where N is the number of classes.

Find the index of the classification layer by viewing the Layers property of the layer graph.

lgraph.Layers

ans =
15x1 Layer array with layers:
The classification layer has the name 'ClassificationLayer_activation_1'. View the classification layer and check the Classes property.

cLayer = lgraph.Layers(end)

cLayer =
    ClassificationOutputLayer with properties:

    Name: 'ClassificationLayer_activation_1'
    Classes: 'auto'
    OutputSize: 'auto'

    Hyperparameters
    LossFunction: 'crossentropyex'

Because the Classes property of the layer is 'auto', you must specify the classes manually. Set the classes to 0, 1, ..., 9, and then replace the imported classification layer with the new one.

cLayer.Classes = string(0:9)

cLayer =
    ClassificationOutputLayer with properties:

    Name: 'ClassificationLayer_activation_1'
    Classes: [0 1 2 3 4 5 6 7 8 9]
    OutputSize: 10
Hyperparameters

LossFunction: 'crossentropyex'

lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);

**Assemble Network**

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)
```

```
net = DAGNetwork with properties:
    Layers: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]
```

**Input Arguments**

- `lgraph` — *Layer graph*
  LayerGraph object

  Layer graph, specified as a `LayerGraph` object. To create a layer graph, use `layerGraph`.

- `layerName` — *Name of layer to replace*
  string scalar | character vector

  Name of the layer to replace, specified as a string scalar or a character vector.

- `larray` — *Network layers*
  Layer array

  Network layers, specified as a `Layer` array.

  For a list of built-in layers, see “List of Deep Learning Layers”.

- `mode` — *Method to reconnect layers*
  'name' (default) | 'order'

  Method to reconnect layers.
Method to reconnect layers specified as one of the following:

- `'name'` – Reconnect `larray` using the input and output names of the replaced layer. For each layer connected to an input of the replaced layer, reconnect the layer to the input of the same input name of `larray(1)`. For each layer connected to an output of the replaced layer, reconnect the layer to the output of the same output name of `larray(end)`.

- `'order'` – Reconnect `larray` using the order of the input names of `larray(1)` and the output names of `larray(end)`. Reconnect the layer connected to the `i`th input of the replaced layer to the `i`th input of `larray(1)`. Reconnect the layer connected to the `j`th output of the replaced layer to the `j`th output of `larray(end)`.

Data Types: `char` | `string`

Output Arguments

`newlgraph` — Output layer graph
`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`PlaceholderLayer` | `addLayers` | `assembleNetwork` | `connectLayers` | `disconnectLayers` | `findPlaceholderLayers` | `layerGraph` | `removeLayers`

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“Train Residual Network for Image Classification”
“Train Deep Learning Network to Classify New Images”

Introduced in R2018b
analyzeNetwork

Analyze deep learning network architecture

Syntax

analyzeNetwork(layers)

Description

analyzeNetwork(layers) analyzes the deep learning network architecture specified by layers. The analyzeNetwork function displays an interactive visualization of the network architecture, detects errors and issues in the network, and provides detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

Use the network analyzer to visualize and understand the network architecture, check that you have defined the architecture correctly, and detect problems before training. Problems that analyzeNetwork detects include missing or unconnected layers, incorrectly sized layer inputs, an incorrect number of layer inputs, and invalid graph structures.

Examples

Analyze Trained Network

Load a pretrained GoogLeNet convolutional neural network.

net = googlenet

net =

DAGNetwork with properties:

Layers: [144×1 nnet.cnn.layer.Layer]
Analyze the network. `analyzeNetwork` displays an interactive plot of the network architecture and a table containing information about the network layers.

Investigate the network architecture using the plot to the left. Select a layer in the plot. The selected layer is highlighted in the plot and in the layer table.

In the table, view layer information such as layer properties, layer type, and sizes of the layer activations and learnable parameters. The activations of a layer are the outputs of that layer.

Select a deeper layer in the network. Notice that activations in deeper layers are smaller in the spatial dimensions (the first two dimensions) and larger in the channel dimension (the last dimension). Using this structure enables convolutional neural networks to gradually increase the number of extracted image features while decreasing the spatial resolution.

Show the total number of learnable parameters in each layer by clicking the arrow in the top-right corner of the layer table and select **Total Learnables**. To sort the layer table by column value, hover the mouse over the column heading and click the arrow that appears. For example, you can determine which layer contains the most parameters by sorting the layers by the total number of learnable parameters.

`analyzeNetwork(net)`
Fix Errors in Network Architecture

Create a simple convolutional network with shortcut connections. Create the main branch of the network as an array of layers and create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially.

```matlab
layers = [
    imageInputLayer([32 32 3], 'Name', 'input')
    convolution2dLayer(5, 16, 'Padding', 'same', 'Name', 'conv_1')
    reluLayer('Name', 'relu_1')
    convolution2dLayer(3, 16, 'Padding', 'same', 'Stride', 2, 'Name', 'conv_2')
    reluLayer('Name', 'relu_2')
];
```
additionLayer(2,'Name','add1')

convolution2dLayer(3,16,'Padding','same','Stride',2,'Name','conv_3')
reluLayer('Name','relu_3')
additionLayer(3,'Name','add2')

fullyConnectedLayer(10,'Name','fc')
classificationLayer('Name','output')];

lgraph = layerGraph(layers);

Create the shortcut connections. One of the shortcut connections contains a single 1-by-1 convolutional layer skipConv.

skipConv = convolution2dLayer(1,16,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
lgraph = connectLayers(lgraph,'relu_1','add1/in2');
lgraph = connectLayers(lgraph,'add1','add2/in2');

Analyze the network architecture. `analyzeNetwork` finds four errors in the network.

analyzeNetwork(lgraph)
Investigate and fix the errors in the network. In this example, the following issues cause the errors:

- A softmax layer, which outputs class probabilities, must precede the classification layer. To fix the error in the output classification layer, add a softmax layer before the classification layer.
- The skipConv layer is not connected to the rest of the network. It should be a part of the shortcut connection between the add1 and add2 layers. To fix this error, connect add1 to skipConv and skipConv to add2.
- The add2 layer is specified to have three inputs, but the layers only has two inputs. To fix the error, specify the number of inputs as 2.
- All the inputs to an addition layer must have the same size, but the add1 layer has two inputs with different sizes. Because the conv_2 layer has a 'Stride' value of 2, this layer downsamples the activations by a factor of two in the first two dimensions (the spatial dimensions). To resize the input from the relu2 layer so that it has the same
size as the input from relu1, remove the downsampling by setting the 'Stride' value of the conv_2 layer to 1.

Apply these modifications to the layer graph construction from the beginning of this example and create a new layer graph.

layers = [
    imageInputLayer([32 32 3],'Name','input')
    convolution2dLayer(5,16,'Padding','same','Name','conv_1')
    reluLayer('Name','relu_1')
    convolution2dLayer(3,16,'Padding','same','Stride',1,'Name','conv_2')
    reluLayer('Name','relu_2')
    additionLayer(2,'Name','add1')
    convolution2dLayer(3,16,'Padding','same','Stride',2,'Name','conv_3')
    reluLayer('Name','relu_3')
    additionLayer(2,'Name','add2')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax');
    classificationLayer('Name','output')
];

lgraph = layerGraph(layers);

skipConv = convolution2dLayer(1,16,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
lgraph = connectLayers(lgraph,'relu_1','add1/in2');
lgraph = connectLayers(lgraph,'add1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add2/in2');

Analyze the new architecture. The new network does not contain any errors and is ready to be trained.

analyzeNetwork(lgraph)
Input Arguments

layers — Network architecture
Layer array | LayerGraph object | SeriesNetwork object | DAGNetwork object

Network architecture, specified as a Layer array, LayerGraph object, SeriesNetwork object, or DAGNetwork object.

See Also
DAGNetwork | Deep Network Designer | LayerGraph | SeriesNetwork | assembleNetwork | plot | trainNetwork
Topics
“Create Simple Deep Learning Network for Classification”
“Transfer Learning with Deep Network Designer”
“Build Networks with Deep Network Designer”
“Train Deep Learning Network to Classify New Images”
“Pretrained Deep Neural Networks”
“Visualize Activations of a Convolutional Neural Network”
“Deep Learning in MATLAB”

Introduced in R2018a
assembleNetwork

Assemble deep learning network from pretrained layers

Syntax

assembledNet = assembleNetwork(layers)

Description

assembleNetwork creates deep learning networks from layers without training.

Use assembleNetwork for the following tasks:

• Convert a layer array or layer graph to a network ready for prediction.
• Assemble networks from imported layers.
• Modify the weights of a trained network.

To train a network from scratch, use trainNetwork.

assembledNet = assembleNetwork(layers) assembles the layer array or layer graph layers into a deep learning network ready to use for prediction.

Examples

Assemble Network from Pretrained Keras Layers

Import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in 'digitsDAGnetwithnoise.h5' classifies images of digits.
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);

Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learning Toolbox™. The importKerasLayers function displays a warning and replaces the unsupported layers with placeholder layers.

Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using findPlaceholderLayers and display their Keras configurations.

placeholderLayers = findPlaceholderLayers(lgraph);
placeholderLayers.KerasConfiguration

ans = struct with fields:
    trainable: 1
        name: 'gaussian_noise_1'
        stddev: 1.5000

ans = struct with fields:
    trainable: 1
        name: 'gaussian_noise_2'
        stddev: 0.7000

Define a custom Gaussian noise layer by saving the file gaussianNoiseLayer.m in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');

Replace the placeholder layers with the custom layers using replaceLayer.

lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
Specify Class Names

The imported classification layer does not contain the classes, so you must specify these before assembling the network. If you do not specify the classes, then the software automatically sets the classes to 1, 2, ..., N, where N is the number of classes.

The classification layer has the name 'ClassificationLayer_activation_1'. Set the classes to 0, 1, ..., 9, and then replace the imported classification layer with the new one.

cLayer = lgraph.Layers(end);
cLayer.Classes = string(0:9);
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);

Assemble Network

Assemble the layer graph using assembleNetwork. The function returns a DAGNetwork object that is ready to use for prediction.

net = assembleNetwork(lgraph)

net = DAGNetwork with properties:

    Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]

Input Arguments

layers — Network layers
Layer array | LayerGraph object

Network layers, specified as a Layer array or a LayerGraph object.

To create a network with all layers connected sequentially, you can use a Layer array as the input argument. In this case, the returned network is a SeriesNetwork object.

A directed acyclic graph (DAG) network has a complex structure in which layers can have multiple inputs and outputs. To create a DAG network, specify the network architecture as a LayerGraph object and then use that layer graph as the input argument to assembleNetwork.
For a list of built-in layers, see “List of Deep Learning Layers”.

**Output Arguments**

**assembledNet — Assembled network**  
*SeriesNetwork object | DAGNetwork object*

Assembled network ready for prediction, returned as a `SeriesNetwork` object or a `DAGNetwork` object. The returned network depends on the `layers` input argument:

- If `layers` is a `Layer` array, then `assembledNet` is a `SeriesNetwork` object.
- If `layers` is a `LayerGraph` object, then `assembledNet` is a `DAGNetwork` object.

**See Also**

- `findPlaceholderLayers`
- `importKerasLayers`
- `importKerasNetwork`
- `replaceLayer`
- `trainNetwork`

**Topics**

- “Assemble Network from Pretrained Keras Layers”
- “Deep Learning in MATLAB”
- “Pretrained Deep Neural Networks”
- “Define Custom Deep Learning Layers”

**Introduced in R2018b**
**exportONNXNetwork**

Export network to ONNX model format

**Syntax**

```matlab
exportONNXNetwork(net,filename)
exportONNXNetwork(net,filename,Name,Value)
```

**Description**

Export a trained Deep Learning Toolbox network to the ONNX (Open Neural Network Exchange) model format.

`exportONNXNetwork(net,filename)` exports the deep learning network `net` with weights to the ONNX format file `filename`. If `filename` exists, then `exportONNXNetwork` overwrites the file.

This function requires the Deep Learning Toolbox Converter for ONNX Model Format support package. If this support package is not installed, then the function provides a download link.

`exportONNXNetwork(net,filename,Name,Value)` exports a network using additional options specified by one or more name-value pair arguments.

**Examples**

**Export Network in ONNX Format**

Load a pretrained SqueezeNet convolutional neural network. If Deep Learning Toolbox Model for SqueezeNet Network is not installed, then the `squeezenet` function provides a download link.

```matlab
net = squeezenet
```
net = DAGNetwork with properties:

    Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]

Export the network as an ONNX format file in the current folder called squeezenet.onnx. If the Deep Learning Toolbox Converter for ONNX Model Format support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install.

filename = 'squeezenet.onnx';
exportONNXNetwork(net, filename)

Now, you can import the squeezenet.onnx file into any deep learning framework that supports ONNX import.

**Input Arguments**

- **net** — Trained network
  SeriesNetwork object | DAGNetwork object
  Trained network, specified as a SeriesNetwork or a DAGNetwork object. You can get a trained network by importing a pretrained network (for example, by using the googlenet function) or by training your own network using trainNetwork.

- **filename** — Name of file
  character vector | string scalar
  Name of file, specified as a character vector or string scalar.
  Example: 'network.onnx'

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.
Example: `exportONNXNetwork(net, filename, 'NetworkName', 'my_net')` exports a network and specifies 'my_net' as the network name in the saved ONNX network.

**NetworkName — Name of ONNX network**

'Network' (default) | character vector | string scalar

Name of ONNX network to store in the saved file, specified as a character vector or a string scalar.

Example: 'my_squeezenet'

**OpsetVersion — Version of ONNX operator set**

8 (default) | 6 | 7 | 9

Version of ONNX operator set to use in the exported model. If the default operator set does not support the network you are trying to export, then try using a later version. If you import the exported network to another framework and you used an operator set during export that the importer does not support, then the import can fail.

Example: 6

**Tips**

- `exportONNXNetwork` does not export settings or properties related to network training such as training options, learning rate factors, or regularization factors.

- If you export a network that contains a layer that the ONNX format does not support, then `exportONNXNetwork` saves a placeholder ONNX operator in place of the unsupported layer and returns a warning. You cannot import an ONNX network with a placeholder operator into other deep learning frameworks.

- `exportONNXNetwork` can export the following:
  - Networks that have both convolutional and LSTM layers, for example, for video classification applications.
  - All custom layers that are created when importing networks from ONNX or TensorFlow-Keras using Deep Learning Toolbox Converter for ONNX Model Format or Deep Learning Toolbox Importer for TensorFlow-Keras Models as in the below table.
  - The following layers:
## ONNX Exporter Supported Layers

### Deep Learning Toolbox Layers
- additionLayer
- averagePooling2dLayer
- averagePooling3dLayer
- batchNormalizationLayer
- bilstmLayer
- classificationLayer
- clippedReluLayer
- convolution2dLayer
- concatenationLayer
- convolution3dLayer
- crop2dLayer
- CrossChannelNormalizationLayer
- depthConcatenationLayer
- dropoutLayer
- fullyConnectedLayer
- flattenLayer
- globalAveragePooling2dLayer
- groupedConvolution2dLayer
- imageInputLayer
- image3dInputLayer
- leakyReluLayer
- lstmLayer
- maxPooling2dLayer
- maxPooling3dLayer
- maxUnpooling2dLayer
- regressionLayer
- reluLayer
<table>
<thead>
<tr>
<th>ONNX Exporter Supported Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sequenceInputLayer</td>
<td></td>
</tr>
<tr>
<td>softmaxLayer</td>
<td></td>
</tr>
<tr>
<td>transposedConv2dLayer</td>
<td></td>
</tr>
<tr>
<td>transposedConv3dLayer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ONNX Importer Custom Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nnet.onnx.layer.ClipLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.onnx.layer.FlattenLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.onnx.layer.GlobalAveragePooling2dLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.onnx.layer.IdentityLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.onnx.layer.PReluLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.onnx.layer.ElementwiseAffineLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.onnx.layer.TanhLayer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keras Importer Custom Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nnet.keras.layer.FlattenCStyleLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.keras.layer.GlobalAveragePooling2dLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.keras.layer.SigmoidLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.keras.layer.TanhLayer</td>
<td></td>
</tr>
<tr>
<td>nnet.keras.layer.ZeroPadding2dLayer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Caffe Importer Custom Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nnet.caffe.layer.TanhLayer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computer Vision Toolbox Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pixelClassificationLayer</td>
<td></td>
</tr>
<tr>
<td>rcnnBoxRegressionLayer</td>
<td></td>
</tr>
<tr>
<td>regionProposalLayer</td>
<td></td>
</tr>
</tbody>
</table>
ONNX Exporter Supported Layers

roiInputLayer
roiMaxPooling2dLayer
rpnClassificationLayer
rpnSoftmaxLayer

Text Analytics Toolbox Layers

wordEmbeddingLayer

References


See Also

importCaffeLayers | importCaffeNetwork | importKerasLayers | importKerasNetwork | importONNXLayers | importONNXNetwork

Topics

“Pretrained Deep Neural Networks”
“Deep Learning in MATLAB”

Introduced in R2018a
importONNXNetwork

Import pretrained ONNX network

Syntax

net = importONNXNetwork(modelfile,'OutputLayerType',outputtype)
net = importONNXNetwork(modelfile,'OutputLayerType',
outputtype,'Classes',classes)

Description

Import a pretrained network from ONNX (Open Neural Network Exchange).

net = importONNXNetwork(modelfile,'OutputLayerType',outputtype)
imports a pretrained network from the ONNX (Open Neural Network Exchange) file
modelfile and specifies the output layer type of the imported network.

This function requires the Deep Learning Toolbox Converter for ONNX Model Format
support package. If this support package is not installed, then the function provides a
download link.

net = importONNXNetwork(modelfile,'OutputLayerType',
outputtype,'Classes',classes) additionally specifies the classes for a classification
network.

Examples

Download and Install Deep Learning Toolbox Converter for ONNX Model Format

Download and install the Deep Learning Toolbox Converter for ONNX Model Format
support package.

Type importONNXNetwork at the command line.
importONNXNetwork

If Deep Learning Toolbox Converter for ONNX Model Format is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file 'cifarResNet.onnx' at the command line. If the support package is installed, then the function returns a DAGNetwork object.

```matlab
modelfile = 'cifarResNet.onnx';
classes = ["airplane" "automobile" "bird" "cat" "deer" "dog" "frog" "horse" "ship" "truck"];
net = importONNXNetwork(modelfile, 'OutputLayerType', 'classification', 'Classes', classes);
```

```
net = 
    DAGNetwork with properties:
        Layers: [77x1 nnet.cnn.layer.Layer]
        Connections: [85x2 table]
```

**Import ONNX Network**

Import a residual neural network trained on the CIFAR-10 data set. Specify the file containing the ONNX network, its output type, and its output classes.

```matlab
modelfile = 'cifarResNet.onnx';
classes = ["airplane" "automobile" "bird" "cat" "deer" "dog" "frog" "horse" "ship" "truck"];
net = importONNXNetwork(modelfile, ...
               'OutputLayerType', 'classification', ...
               'Classes', classes)
```

```
net = 
    DAGNetwork with properties:
        Layers: [77x1 nnet.cnn.layer.Layer]
        Connections: [85x2 table]
```

Analyze the imported network.

```matlab
analyzeNetwork(net)
```
Input Arguments

`modelfile` — Name of ONNX model file
character vector | string scalar

Name of ONNX model file containing the network, specified as a character vector or a string scalar. The file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: 'cifarResNet.onnx'

`outputtype` — Type of output layer
'classification' | 'regression' | 'pixelclassification'

1-1467
Type of the output layer that the function appends to the end of the imported network, specified as 'classification', 'regression', or 'pixelclassification'. Using 'pixelclassification' appends a pixelClassificationLayer object (requires Computer Vision Toolbox).

Example: 'regression'

classes — Classes of the output layer

'auto' (default) | categorical vector | string array | cell array of character vectors

Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If Classes is 'auto', then the software sets the classes to categorical(1:N), where N is the number of classes. If you specify a string array or cell array of character vectors str, then the software sets the classes of the output layer to categorical(str,str).

Data Types: char | categorical | string | cell

Output Arguments

net — Pretrained network

DAGNetwork object

Pretrained network, returned as DAGNetwork object.

Tips

• If the ONNX network contains a layer that Deep Learning Toolbox Converter for ONNX Model Format does not support, then the function returns an error message. In this case, you can still use importONNXLayers to import the network architecture and weights.

importONNXNetwork supports the following ONNX operators, with some limitations:

• Add
• AveragePool
• BatchNormalization
• Clip
- Concat
- Conv
- ConvTranspose
- Div
- Dropout
- Flatten
- Gemm
- GlobalAveragePool
- Identity
- LRN
- LSTM
- LeakyRelu
- MaxPool
- MatMul
- Mul
- PRelu
- Relu
- Reshape
- Sigmoid
- Softmax
- Sub
- Sum
- Tanh
- ImageScaler

- To use a pretrained network for prediction or transfer learning on new images, you must preprocess your images in the same way the images that were used to train the imported model were preprocessed. Most common preprocessing steps are resizing images, subtracting image average values, and converting the images from BGR images to RGB.

- To resize images, use \texttt{imresize}. For example, \texttt{imresize(image,[227,227,3])}.
- To convert images from RGB to BGR format, use \texttt{flip}. For example, \texttt{flip(image,3)}.  

1-1469
For more information on preprocessing images for training and prediction, see “Preprocess Images for Deep Learning”.

**Compatibility Considerations**

'ClassNames' option will be removed

*Not recommended starting in R2018b*

'ClassNames' will be removed. Use 'Classes' instead. To update your code, replace all instances of 'ClassNames' with 'Classes'. There are some differences between the corresponding properties in classification output layers that require additional updates to your code.

The `ClassNames` property of a classification output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.

**References**


**See Also**

`exportONNXNetwork` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `importONNXLayers`

**Topics**

“Preprocess Images for Deep Learning”

“Deep Learning in MATLAB”

“Pretrained Deep Neural Networks”

**Introduced in R2018a**
importONNXLayers

Import layers from ONNX network

Syntax

```matlab
layers = importONNXLayers(modelfile)
layers = importONNXLayers(modelfile,Name,Value)
```

Description

Import layers from an ONNX (Open Neural Network Exchange) network.

```
layers = importONNXLayers(modelfile) imports the layers of an ONNX (Open Neural Network Exchange) network from the file `modelfile`. You can train the imported layers on a new data set or assemble the layers into a network ready for prediction. For an example of the workflow of assembling a network, see “Assemble Network from Pretrained Keras Layers”.
```

This function requires the Deep Learning Toolbox Converter for ONNX Model Format support package. If this support package is not installed, then the function provides a download link.

```
layers = importONNXLayers(modelfile,Name,Value) imports the layers from an ONNX network with additional options specified by one or more name-value pair arguments.
```

For example, `importONNXLayers(modelfile,'ImportWeights',false)` imports the network architecture without weights from the file `modelfile`.

Examples
Download and Install Deep Learning Toolbox Converter for ONNX Model Format

Download and install the Deep Learning Toolbox Converter for ONNX Model Format support package.

Type `importONNXLayers` at the command line.

`importONNXLayers`

If Deep Learning Toolbox Converter for ONNX Model Format is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by importing the network from the model file `'cifarResNet.onnx'` at the command line. If the support package is installed, then the function returns a `DAGNetwork` object.

```matlab
modelfile = 'cifarResNet.onnx';
layers = importONNXLayers(modelfile, 'OutputLayerType', 'classification')
```

```matlab
layers = 
    LayerGraph with properties:
        Layers: [77×1 nnet.cnn.layer.Layer]
        Connections: [85×2 table]
```

Import ONNX Network Architecture

Import the architecture and weights of a residual neural network trained on the CIFAR-10 data set. Specify the file containing the ONNX network and the type of the output layer to add to the imported network.

```matlab
modelfile = 'cifarResNet.onnx';
lgraph = importONNXLayers(modelfile, ...  
    'OutputLayerType', 'classification', ...  
    'ImportWeights', true)
```

```matlab
lgraph = 
    LayerGraph with properties:
        Layers: [77×1 nnet.cnn.layer.Layer]
        Connections: [85×2 table]
```
Analyze the imported network architecture.

`analyzeNetwork(lgraph);`

### Input Arguments

**modelfile** — Name of ONNX model file  
character vector | string scalar

Name of ONNX model file containing the network, specified as a character vector or a string scalar. The file must be in the current folder, in a folder on the MATLAB path, or you must include a full or relative path to the file.

Example: `'cifarResNet.onnx'`
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:
importONNXLayers(modelFile, 'OutputLayerType', 'classification') imports the network layers from modelFile and adds an output layer for a classification output layer at the end of the imported layers.

OutputLayerType — Type of output layer
'classification' | 'regression' | 'pixelclassification'

Type of the output layer that the function appends to the end of the imported network architecture, specified as 'classification', 'regression', or 'pixelclassification'. Using 'pixelclassification' appends a pixelClassificationLayer object (requires Computer Vision Toolbox).

Example: 'OutputLayerType', 'regression'

ImportWeights — Indicator to import weights
false (default) | true

Indicator to import weights as well as the network architecture, specified as either false or true.

Example: 'ImportWeights', true

Data Types: logical

Output Arguments

layers — Network architecture
LayerGraph object

Network architecture, returned as a LayerGraph object.
Tips

• If the ONNX network contains a layer that Deep Learning Toolbox Converter for ONNX Model Format does not support, then `importONNXLayers` inserts a placeholder layer in place of the unsupported layer. To find the names and indices of the unsupported layers in the network, use the `findPlaceholderLayers` function. You then can replace a placeholder layer with a new layer that you define. To replace a layer, use `replaceLayer`.

`importONNXLayers` supports the following ONNX operators, with some limitations:

• Add
• AveragePool
• BatchNormalization
• Clip
• Concat
• Conv
• ConvTranspose
• Div
• Dropout
• Flatten
• Gemm
• GlobalAveragePool
• Identity
• LRN
• LSTM
• LeakyRelu
• MaxPool
• MatMul
• Mul
• PReLU
• ReLU
• Reshape
• Sigmoid
• Softmax
• Sub
• Sum
• Tanh
• ImageScaler

• The workflow for assembling layers imported from ONNX into a network ready for prediction is the same as assembling layers imported from Keras. For an example of this workflow, see “Assemble Network from Pretrained Keras Layers”.

References


See Also

assembleNetwork | exportONNXNetwork | findPlaceholderLayers | importCaffeLayers | importCaffeNetwork | importKerasLayers | importKerasNetwork | importONNXNetwork | replaceLayer

Topics

“Deep Learning in MATLAB”
“Pretrained Deep Neural Networks”
“List of Deep Learning Layers”
“Define Custom Deep Learning Layers”
“Define Custom Deep Learning Layer with Learnable Parameters”
“Check Custom Layer Validity”
“Assemble Network from Pretrained Keras Layers”

Introduced in R2018a
confusionchart

Create confusion matrix chart for classification problem

Syntax

`cm = confusionchart(trueLabels,predictedLabels)`  
`cm = confusionchart(m)`  
`cm = confusionchart(m,classLabels)`  
`cm = confusionchart(parent,___)`  
`cm = confusionchart(___,Name,Value)`

Description

`cm = confusionchart(trueLabels,predictedLabels)` creates a confusion matrix chart from true labels `trueLabels` and predicted labels `predictedLabels` and returns a `ConfusionMatrixChart` object. The rows of the confusion matrix correspond to the true class and the columns correspond to the predicted class. Diagonal and off-diagonal cells correspond to correctly and incorrectly classified observations, respectively. Use `cm` to modify the confusion matrix chart after it is created. For a list of properties, see `ConfusionMatrixChart` Properties.

`cm = confusionchart(m)` creates a confusion matrix chart from the numeric confusion matrix `m`. Use this syntax if you already have a numeric confusion matrix in the workspace.

`cm = confusionchart(m,classLabels)` specifies class labels that appear along the x-axis and y-axis. Use this syntax if you already have a numeric confusion matrix and class labels in the workspace.

`cm = confusionchart(parent,___)` creates the confusion chart in the figure, panel, or tab specified by `parent`.

`cm = confusionchart(___,Name,Value)` specifies additional `ConfusionMatrixChart` properties using one or more name-value pair arguments. Specify the properties after all other input arguments. For a list of properties, see `ConfusionMatrixChart` Properties.
Examples

Create Confusion Matrix Chart

Load a sample of predicted and true labels for a classification problem. trueLabels is the true labels for an image classification problem and predictedLabels is the predictions of a convolutional neural network.

load('Cifar10Labels.mat','trueLabels','predictedLabels');

Create a confusion matrix chart.

figure

cm = confusionchart(trueLabels,predictedLabels);
Modify the appearance and behavior of the confusion matrix chart by changing property values. Add column and row summaries and a title. A column-normalized column summary displays the number of correctly and incorrectly classified observations for each predicted class as percentages of the number of observations of the corresponding predicted class. A row-normalized row summary displays the number of correctly and incorrectly classified observations for each true class as percentages of the number of observations of the corresponding true class.

```matlab
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
cm.Title = 'CIFAR-10 Confusion Matrix';
```
Create Confusion Matrix Chart from Numeric Confusion Matrix

You can use `confusionchart` to create a confusion matrix chart from a numeric confusion matrix.

Load a sample confusion matrix `m` and the associated class labels `classLabels`.

```matlab
load('Cifar10ConfusionMat.mat','m','classLabels');
m
```

```
m = 10×10
```

```
airplane 4 21 8 4 1 5 5 23 6
automobile 5 972 2 1 5 15
bird 26 2 892 30 13 8 17 5 4 3
cat 12 4 32 826 24 48 30 12 5 7
deer 5 1 28 24 898 13 14 14 2 1
dog 7 2 28 111 18 801 13 17 3
frog 5 16 27 3 4 943 1 1
horse 9 14 13 22 17 3 915 2 4
ship 37 10 4 4 1 2 1 931 10
truck 20 39 3 3 2 1 9 923

```

Confusion Matrix Chart

```
CIFAR-10 Confusion Matrix

Predicted Class

True Class
```
Create a confusion matrix chart from the numeric confusion matrix and the class labels.

```matlab
cm = confusionchart(m, classLabels);
```
Sort Classes by Precision or Recall

Load a sample of predicted and true labels for a classification problem. `trueLabels` are the true labels for an image classification problem and `predictedLabels` are the predictions of a convolutional neural network. Create a confusion matrix chart with column and row summaries:

```matlab
load('Cifar10Labels.mat','trueLabels','predictedLabels');
figure
cm = confusionchart(trueLabels,predictedLabels, ...)
```
To sort the classes of the confusion matrix by class-wise recall (true positive rate), normalize the cell values across each row, that is, by the number of observations that have the same true class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the row summaries to the right are decreasing.

```matlab
cm.Normalization = 'row-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```
To sort the classes by class-wise precision (positive predictive value), normalize the cell values across each column, that is, by the number of observations that have the same predicted class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the column summaries at the bottom are decreasing.

```matlab
cm.Normalization = 'column-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```
**Input Arguments**

*trueLabels — True labels of classification problem*

categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

True labels of classification problem, specified as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. If `trueLabels` is a vector, then each element corresponds to one observation. If `trueLabels` is a character array, then it must be two-dimensional with each row corresponding to the label of one observation.
**predictedLabels** — Predicted labels of classification problem  
categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

Predicted labels of classification problem, specified as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. If `predictedLabels` is a vector, then each element corresponds to one observation. If `predictedLabels` is a character array, then it must be two-dimensional with each row corresponding to the label of one observation.

**m** — Confusion matrix  
matrix

Confusion matrix, specified as a matrix. `m` must be square and its elements must be positive integers. The element `m(i,j)` is the number of times an observation of the `i`th true class was predicted to be of the `j`th class. Each colored cell of the confusion matrix chart corresponds to one element of the confusion matrix `m`.

**classLabels** — Class labels  
categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

Class labels of the confusion matrix chart, specified as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. If `classLabels` is a vector, then it must have the same number of elements as the confusion matrix has rows and columns. If `classLabels` is a character array, then it must be two-dimensional with each row corresponding to the label of one class.

**parent** — Parent container  
Figure object | Panel object | Tab object

Parent container in which to plot, specified as a `Figure`, `Panel`, or `Tab` object.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `cm = confusionchart(trueLabels,predictedLabels,'Title','My Title Text','ColumnSummary','column-normalized')`
The properties listed here are only a subset. For a complete list, see ConfusionMatrixChart Properties.

**Title — Title**

' '(default) | character vector | string scalar

Title of the confusion matrix chart, specified as a character vector or string scalar.

Example: `cm = confusionchart(__,'Title','My Title Text')`

Example: `cm.Title = 'My Title Text'`

**ColumnSummary — Column summary**

'off' (default) | 'absolute' | 'column-normalized' | 'total-normalized'

Column summary of the confusion matrix chart, specified as one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'off'</td>
<td>Do not display a column summary.</td>
</tr>
<tr>
<td>'absolute'</td>
<td>Display the total number of correctly and incorrectly classified observations for each predicted class.</td>
</tr>
<tr>
<td>'column-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the number of observations of the corresponding predicted class. The percentages of correctly classified observations can be thought of as class-wise precisions (or positive predictive values).</td>
</tr>
<tr>
<td>'total-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the total number of observations.</td>
</tr>
</tbody>
</table>

Example: `cm = confusionchart(__,'ColumnSummary','column-normalized')`

Example: `cm.ColumnSummary = 'column-normalized'`
**RowSummary — Row summary**

'off' (default) | 'absolute' | 'row-normalized' | 'total-normalized'

Row summary of the confusion matrix chart, specified as one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'off'</td>
<td>Do not display a row summary.</td>
</tr>
<tr>
<td>'absolute'</td>
<td>Display the total number of correctly and incorrectly classified observations for each true class.</td>
</tr>
<tr>
<td>'row-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each true class as percentages of the number of observations of the corresponding true class. The percentages of correctly classified observations can be thought of as class-wise recalls (or true positive rates).</td>
</tr>
<tr>
<td>'total-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each true class as percentages of the total number of observations.</td>
</tr>
</tbody>
</table>

Example: `cm = confusionchart(__,'RowSummary','row-normalized')`

Example: `cm.RowSummary = 'row-normalized'`

**Normalization — Normalization of cell values**

'absolute' (default) | 'column-normalized' | 'row-normalized' | 'total-normalized'

Normalization of cell values, specified as one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'absolute'</td>
<td>Display the total number of observations in each cell.</td>
</tr>
<tr>
<td>'column-normalized'</td>
<td>Normalize each cell value by the number of observations that has the same predicted class.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>'row-normalized'</td>
<td>Normalize each cell value by the number of observations that has the same true class.</td>
</tr>
<tr>
<td>'total-normalized'</td>
<td>Normalize each cell value by the total number of observations.</td>
</tr>
</tbody>
</table>

Modifying the normalization of cell values also affects the colours of the cells.

Example: `cm = confusionchart(__,'Normalization','total-normalized')`
Example: `cm.Normalization = 'total-normalized'`

**Tips**

- If you have Statistics and Machine Learning Toolbox™, you can create a confusion matrix chart for tall arrays. For details, see `confusionchart` and “Confusion Matrix for Classification Using Tall Arrays” (Statistics and Machine Learning Toolbox).

**See Also**

**Functions**
categorical | classify | confusionmat | sortClasses

**Properties**
ConfusionMatrixChart Properties

**Topics**
“Deep Learning in MATLAB”

**Introduced in R2018b**
ConfusionMatrixChart Properties

Confusion matrix chart appearance and behavior

Description

ConfusionMatrixChart properties control the appearance and behavior of a ConfusionMatrixChart object. By changing property values, you can modify certain aspects of the confusion matrix chart. For example, you can add a title:

```matlab
cm = confusionchart([1 3 5; 2 4 6; 11 7 3]);
cm.Title = 'My Confusion Matrix Title';
```

Properties

Labels

**Title — Title**

' ' (default) | character vector | string scalar

Title of the confusion matrix chart, specified as a character vector or string scalar.

Example: `cm = confusionchart(__,'Title','My Title Text')`

Example: `cm.Title = 'My Title Text'`

**XLabel — Label for x-axis**

'Predicted class' (default) | string scalar | character vector

Label for the x-axis, specified as a string scalar or character vector.

Example: `cm = confusionchart(__,'XLabel','My Label')`

Example: `cm.XLabel = 'My Label'`

**YLabel — Label for y-axis**

'True class' (default) | string scalar | character vector

Label for the x-axis, specified as a string scalar or character vector.
Example: `cm = confusionchart(__,'YLabel','My Label')`
Example: `cm.YLabel = 'My Label'

**ClassLabels — Class labels**
categorical vector | numeric vector | string vector | character array | cell array of character vectors | logical vector

This property is read-only.

Class labels of the confusion matrix chart, stored as a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector.

**Row and Column Summaries**

**ColumnSummary — Column summary**
'off' (default) | 'absolute' | 'column-normalized' | 'total-normalized'

Column summary of the confusion matrix chart, specified as one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'off'</td>
<td>Do not display a column summary.</td>
</tr>
<tr>
<td>'absolute'</td>
<td>Display the total number of correctly and incorrectly classified observations for each predicted class.</td>
</tr>
<tr>
<td>'column-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the number of observations of the corresponding predicted class. The percentages of correctly classified observations can be thought of as class-wise precisions (or positive predictive values).</td>
</tr>
<tr>
<td>'total-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each predicted class as percentages of the total number of observations.</td>
</tr>
</tbody>
</table>

Example: `cm = confusionchart(__,'ColumnSummary','column-normalized')"
Example: `cm.ColumnSummary = 'column-normalized'`

**RowSummary — Row summary**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'off'</td>
<td>Do not display a row summary.</td>
</tr>
<tr>
<td>'absolute'</td>
<td>Display the total number of correctly and incorrectly classified observations for each true class.</td>
</tr>
<tr>
<td>'row-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each true class as percentages of the number of observations of the corresponding true class. The percentages of correctly classified observations can be thought of as class-wise recalls (or true positive rates).</td>
</tr>
<tr>
<td>'total-normalized'</td>
<td>Display the number of correctly and incorrectly classified observations for each true class as percentages of the total number of observations.</td>
</tr>
</tbody>
</table>

Example: `cm = confusionchart(__,'RowSummary','row-normalized')`
Example: `cm.RowSummary = 'row-normalized'`

**Data**

**NormalizedValues — Values of the confusion matrix**

numeric matrix

This property is read-only.

Values of the confusion matrix, stored as a numeric matrix. This property equals the values of the confusion matrix normalized using the method of the Normalization property. The software recalculates the normalized values of the confusion matrix each time you modify the Normalization property.
Normalization — Normalization of cell values
'absolute' (default) | 'column-normalized' | 'row-normalized' | 'total-normalized'

Normalization of cell values, specified as one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'absolute'</td>
<td>Display the total number of observations in each cell.</td>
</tr>
<tr>
<td>'column-normalized'</td>
<td>Normalize each cell value by the number of observations that has the same predicted class.</td>
</tr>
<tr>
<td>'row-normalized'</td>
<td>Normalize each cell value by the number of observations that has the same true class.</td>
</tr>
<tr>
<td>'total-normalized'</td>
<td>Normalize each cell value by the total number of observations.</td>
</tr>
</tbody>
</table>

Modifying the normalization of cell values also affects the colours of the cells.

Example: `cm = confusionchart(__,'Normalization','total-normalized')`
Example: `cm.Normalization = 'total-normalized'

Color and Styling

GridVisible — State of grid visibility
'on' (default) | 'off'

State of grid visibility, specified as one of the following:

- 'on' — Display grid lines between the chart cells.
- 'off' — Do not display grid lines between the chart cells.

Example: `cm = confusionchart(__,'GridVisible','off')`
Example: `cm.GridVisible = 'off'

DiagonalColor — Color for diagonal cells
[0 0.4471 0.7412] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Color for diagonal cells, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color of each diagonal cell is proportional to the cell value
and the `DiagonalColor` property, normalized to the largest cell value of the confusion matrix chart. Cells with positive values are colored with a minimum amount of color, proportional to the `DiagonalColor` property.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range $[0,1]$; for example, $[0.4 \ 0.6 \ 0.7]$.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

<table>
<thead>
<tr>
<th>Color Name</th>
<th>Short Name</th>
<th>RGB Triplet</th>
<th>Hexadecimal Color Code</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>'red'</td>
<td>'r'</td>
<td>[1 0 0]</td>
<td>'#FF0000'</td>
<td></td>
</tr>
<tr>
<td>'green'</td>
<td>'g'</td>
<td>[0 1 0]</td>
<td>'#00FF00'</td>
<td></td>
</tr>
<tr>
<td>'blue'</td>
<td>'b'</td>
<td>[0 0 1]</td>
<td>'#0000FF'</td>
<td></td>
</tr>
<tr>
<td>'cyan'</td>
<td>'c'</td>
<td>[0 1 1]</td>
<td>'#00FFFF'</td>
<td></td>
</tr>
<tr>
<td>'magenta'</td>
<td>'m'</td>
<td>[1 0 1]</td>
<td>'#FF00FF'</td>
<td></td>
</tr>
<tr>
<td>'yellow'</td>
<td>'y'</td>
<td>[1 1 0]</td>
<td>'#FFFF00'</td>
<td></td>
</tr>
<tr>
<td>'black'</td>
<td>'k'</td>
<td>[0 0 0]</td>
<td>'#000000'</td>
<td></td>
</tr>
<tr>
<td>'white'</td>
<td>'w'</td>
<td>[1 1 1]</td>
<td>'#FFFFFF'</td>
<td></td>
</tr>
</tbody>
</table>

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

<table>
<thead>
<tr>
<th>RGB Triplet</th>
<th>Hexadecimal Color Code</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 0.4470 0.7410]</td>
<td>'#0072BD'</td>
<td></td>
</tr>
<tr>
<td>[0.8500 0.3250 0.0980]</td>
<td>'#D95319'</td>
<td></td>
</tr>
<tr>
<td>[0.9290 0.6940 0.1250]</td>
<td>'#EDB120'</td>
<td></td>
</tr>
</tbody>
</table>
The software chooses an appropriate text color for cell labels automatically, depending on the color of the chart cells.

Example: cm = confusionchart(___,'DiagonalColor','blue')
Example: cm.DiagonalColor = 'blue'

**OffDiagonalColor — Color for off-diagonal cells**

```
[0.8510 0.3255 0.0980] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...
```

Color for off-diagonal cells, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color of each diagonal cell is proportional to the cell value and the OffDiagonalColor property, normalized to the largest cell value of the confusion matrix chart. Cells with positive values are colored with a minimum amount of color, proportional to the OffDiagonalColor property.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

<table>
<thead>
<tr>
<th>Color Name</th>
<th>Short Name</th>
<th>RGB Triplet</th>
<th>Hexadecimal Color Code</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>'red'</td>
<td>'r'</td>
<td>[1 0 0]</td>
<td>'#FF0000'</td>
<td></td>
</tr>
</tbody>
</table>
Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

<table>
<thead>
<tr>
<th>RGB Triplet</th>
<th>Hexadecimal Color Code</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 0.4470 0.7410]</td>
<td>'#0072BD'</td>
<td>![](color#![0 0.4470 0.7410])</td>
</tr>
<tr>
<td>[0.8500 0.3250 0.0980]</td>
<td>'#D95319'</td>
<td>![](color#![0.8500 0.3250 0.0980])</td>
</tr>
<tr>
<td>[0.9290 0.6940 0.1250]</td>
<td>'#EDB120'</td>
<td>![](color#![0.9290 0.6940 0.1250])</td>
</tr>
<tr>
<td>[0.4940 0.1840 0.5560]</td>
<td>'#7E2F8E'</td>
<td>![](color#![0.4940 0.1840 0.5560])</td>
</tr>
<tr>
<td>[0.4660 0.6740 0.1880]</td>
<td>'#77AC30'</td>
<td>![](color#![0.4660 0.6740 0.1880])</td>
</tr>
<tr>
<td>[0.3010 0.7450 0.9330]</td>
<td>'#4DBEEE'</td>
<td>![](color#![0.3010 0.7450 0.9330])</td>
</tr>
<tr>
<td>[0.6350 0.0780 0.1840]</td>
<td>'#A2142F'</td>
<td>![](color#![0.6350 0.0780 0.1840])</td>
</tr>
</tbody>
</table>

The software chooses an appropriate text color for cell labels automatically, depending on the color of the chart cells.

Example: `cm = confusionchart(__,'OffDiagonalColor','blue')`

Example: `cm.OffDiagonalColor = 'blue'`

**FontColor — Text color for title, axis labels, and class labels**

```
[0.1500 0.1500 0.1500] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...
```

Text color for title, axis labels, and class labels, specified as a color name, an RGB triplet, a hexadecimal color code, a color name, or a short name.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.
• An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].

• A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

<table>
<thead>
<tr>
<th>Color Name</th>
<th>Short Name</th>
<th>RGB Triplet</th>
<th>Hexadecimal Color Code</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>'red'</td>
<td>'r'</td>
<td>[1 0 0]</td>
<td>'#FF0000'</td>
<td></td>
</tr>
<tr>
<td>'green'</td>
<td>'g'</td>
<td>[0 1 0]</td>
<td>'#00FF00'</td>
<td></td>
</tr>
<tr>
<td>'blue'</td>
<td>'b'</td>
<td>[0 0 1]</td>
<td>'#0000FF'</td>
<td></td>
</tr>
<tr>
<td>'cyan'</td>
<td>'c'</td>
<td>[0 1 1]</td>
<td>'#00FFFF'</td>
<td></td>
</tr>
<tr>
<td>'magenta'</td>
<td>'m'</td>
<td>[1 0 1]</td>
<td>'#FF00FF'</td>
<td></td>
</tr>
<tr>
<td>'yellow'</td>
<td>'y'</td>
<td>[1 1 0]</td>
<td>'#FFFF00'</td>
<td></td>
</tr>
<tr>
<td>'black'</td>
<td>'k'</td>
<td>[0 0 0]</td>
<td>'#000000'</td>
<td></td>
</tr>
<tr>
<td>'white'</td>
<td>'w'</td>
<td>[1 1 1]</td>
<td>'#FFFFFF'</td>
<td></td>
</tr>
</tbody>
</table>

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

<table>
<thead>
<tr>
<th>RGB Triplet</th>
<th>Hexadecimal Color Code</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0 0.4470 0.7410]</td>
<td>'#0072BD'</td>
<td></td>
</tr>
<tr>
<td>[0.8500 0.3250 0.0980]</td>
<td>'#D95319'</td>
<td></td>
</tr>
<tr>
<td>[0.9290 0.6940 0.1250]</td>
<td>'#EDB120'</td>
<td></td>
</tr>
<tr>
<td>[0.4940 0.1840 0.5560]</td>
<td>'#7E2F8E'</td>
<td></td>
</tr>
<tr>
<td>[0.4660 0.6740 0.1880]</td>
<td>'#77AC30'</td>
<td></td>
</tr>
<tr>
<td>[0.3010 0.7450 0.9330]</td>
<td>'#4DBEEE'</td>
<td></td>
</tr>
<tr>
<td>[0.6350 0.0780 0.1840]</td>
<td>'#A2142F'</td>
<td></td>
</tr>
</tbody>
</table>
The software chooses an appropriate text color for cell labels automatically, depending on the color of the chart cells.

Example: \( cm = \text{confusionchart}(\_, \text{'FontColor','blue'}) \)

Example: \( cm.\text{FontColor} = \text{'blue'} \)

**Font**

**FontName** — Font name

System supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

Example: \( cm = \text{confusionchart}(\_, \text{'FontName','Cambria'}) \)

Example: \( cm.\text{FontName} = \text{'Cambria'} \)

**FontSize** — Font size

Positive scalar

Font size used for the title, axis labels, class labels, and cell labels, specified as a positive scalar. The default font depends on the specific operating system and locale.

The title and axis labels use a slightly larger font size (scaled up by 10%). If there is not enough room to display the cell labels within the cells, then the cell labels use a smaller font size. If the cell labels become too small, then they are hidden.

Example: \( cm = \text{confusionchart}(\_, \text{'FontSize',12}) \)

Example: \( cm.\text{FontSize} = 12 \)

**Position**

**ActivePositionProperty** — Position property to hold constant

'outerposition' (default) | 'innerposition'

Position property to hold constant during resize operations, specified as 'outerposition' or 'innerposition'. The default value of 'outerposition' means that the OuterPosition property remains constant. The InnerPosition property value can change when the parent container changes size, the data changes, or the labels change.

Example: \( cm = \text{confusionchart}(\_, \text{'ActivePositionProperty','innerposition'}) \)
Example: `cm.ActivePositionProperty = 'innerposition'`

**OuterPosition — Outer size and position**

\[ \begin{array}{cccc}
0 & 0 & 1 & 1 \\
\end{array} \] (default) \ | four-element vector

Outer size and position within the parent container (a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`. The outer position includes the title, axis labels, and class labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the chart.
- The `width` and `height` elements are the chart dimensions, which include the chart cells, plus a margin for the surrounding text.

The default value of \[ \begin{array}{cccc}
0 & 0 & 1 & 1 \\
\end{array} \] is the whole interior of the container.

By default, the values are normalized to the container. To change the units, set the `Units` property.

Example: `cm = confusionchart(__,'OuterPosition',[0.1 0.1 0.8 0.8])`
Example: `cm.OuterPosition = [0.1 0.1 0.8 0.8]`

**InnerPosition — Inner size and position**

\[ \begin{array}{cccc}
0.1300 & 0.1100 & 0.7750 & 0.8150 \\
\end{array} \] (default) \ | four-element vector

Inner size and position of the chart within the parent container (a figure, panel, or tab) returned as a four-element vector of the form `[left bottom width height]`. The inner position does not include the title, axis labels, or class labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the chart.
- The `width` and `height` elements are the chart dimensions, which include only the chart cells.

Example: `cm = confusionchart(__,'InnerPosition',[0.1 0.1 0.8 0.8])`
Example: `cm.InnerPosition = [0.1 0.1 0.8 0.8]`

**Position — Inner size and position**

four-element vector
Inner size and position of the chart within the parent container (a figure, panel, or tab) returned as a four-element vector of the form \([\text{left} \ \text{bottom} \ \text{width} \ \text{height}]\). This property is equivalent to the \text{InnerPosition} property.

**Units — Position units**

'normalized' (default) | 'inches' | 'centimeters' | 'points' | 'pixels' | 'characters'

Position units, specified as one of these values:

<table>
<thead>
<tr>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'normalized'</td>
<td>Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to ((0,0)), and the upper right corner maps to ((1,1)).</td>
</tr>
<tr>
<td>'inches'</td>
<td>Inches.</td>
</tr>
<tr>
<td>'centimeters'</td>
<td>Centimeters.</td>
</tr>
</tbody>
</table>
| 'characters'| Based on the default uicontrol font of the graphics root object:  
  - Character width = width of letter \(x\).  
  - Character height = distance between the baselines of two lines of text. |
<p>| 'points'    | Typography points. One point equals (1/72) inch. |</p>
<table>
<thead>
<tr>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pixels'</td>
<td>Pixels.</td>
</tr>
</tbody>
</table>

Starting in R2015b, distances in pixels are independent of your system resolution on Windows® and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.

On Linux® systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the **Units** property before specifying the properties that you want to use these units for, such as **OuterPosition**.

**Visible — State of visibility**

* 'on' (default) | 'off'*

State of visibility, specified as one of these values:

- 'on' — Display the chart.
- 'off' — Hide the chart without deleting it. You still can access the properties of an invisible chart.

**Parent/Child**

**Parent — Parent container**

*figure object | panel object | tab object*

Parent container, specified as a figure, panel, or tab object.

**HandleVisibility — Visibility of object handle**

* 'on' (default) | 'off' | 'callback'*

Visibility of the chart object handle in the Children property of the parent, specified as one of these values:
• 'on' — Object handle is always visible.
• 'off' — Object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. To temporarily hide the handle during the execution of that function, set the HandleVisibility to 'off'.
• 'callback' — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but allows callback functions to access it.

If the object is not listed in the Children property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to 'on' to list all object handles, regardless of their HandleVisibility property setting.

See Also

Functions
categorical | confusionchart | sortClasses

Topics
“Deep Learning in MATLAB”

Introduced in R2018b
sortClasses

**Package**: mlearnlib.graphics.chart

Sort classes of confusion matrix chart

**Syntax**

sortClasses(cm,order)

**Description**

sortClasses(cm,order) sorts the classes of the confusion matrix chart cm in the order specified by order. You can sort the classes in their natural order, by the values along the diagonal of the confusion matrix, or in fixed order that you specify.

**Examples**

**Sort Classes in a Fixed Order**

Load a sample of predicted and true labels for a classification problem. trueLabels are the true labels for an image classification problem and predictedLabels are the predictions of a convolutional neural network. Create a confusion matrix chart.

```matlab
load('Cifar10Labels.mat','trueLabels','predictedLabels');
figure
cm = confusionchart(trueLabels,predictedLabels);
```
Reorder the classes of the confusion matrix chart so that the classes are in a fixed order.

```
sortClasses(cm, ...
    ["cat" "dog" "horse" "dear" "bird" "frog", ...
    "airplane" "ship" "automobile" "truck"]
```
Sort Classes by Precision or Recall

Load a sample of predicted and true labels for a classification problem. `trueLabels` are the true labels for an image classification problem and `predictedLabels` are the predictions of a convolutional neural network. Create a confusion matrix chart with column and row summaries.

```matlab
load('Cifar10Labels.mat','trueLabels','predictedLabels');
figure
cm = confusionchart(trueLabels,predictedLabels);
```
To sort the classes of the confusion matrix by class-wise recall (true positive rate), normalize the cell values across each row, that is, by the number of observations that have the same true class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the row summaries to the right are decreasing.

```
cm.Normalization = 'row-normalized';
sortClasses(cm,'descending-diagonal');
```

```
cm.Normalization = 'absolute';
```
To sort the classes by class-wise precision (positive predictive value), normalize the cell values across each column, that is, by the number of observations that have the same predicted class. Sort the classes by the corresponding diagonal cell values and reset the normalization of the cell values. The classes are now sorted such that the percentages in the blue cells in the column summaries at the bottom are decreasing.

```matlab
cm.Normalization = 'column-normalized';
sortClasses(cm,'descending-diagonal');
cm.Normalization = 'absolute';
```
**Input Arguments**

**cm — Confusion matrix chart**

ConfusionMatrixChart object

Confusion matrix chart, specified as a ConfusionMatrixChart object. To create a confusion matrix chart, use confusionchart,

**order — Order in which to sort classes**

'auto' | 'ascending-diagonal' | 'descending-diagonal' | array
Order in which to sort the classes of the confusion matrix chart, specified as one of these values:

- 'auto' — Sorts the classes into their natural order as defined by the `sort` function. For example, if the class labels of the confusion matrix chart are a string vector, then sort alphabetically. If the class labels are an ordinal categorical vector, then use the order of the class labels.
- 'ascending-diagonal' — Sort the classes so that the values along the diagonal of the confusion matrix increase from top left to bottom right.
- 'descending-diagonal' — Sort the classes so that the values along the diagonal of the confusion matrix decrease from top left to bottom right.
- 'cluster' (Requires Statistics and Machine Learning Toolbox) — Sort the classes to cluster similar classes. You can customize clustering by using the `pdist`, `linkage`, and `optimalleaforder` functions. For details, see “Sort Classes to Cluster Similar Classes” (Statistics and Machine Learning Toolbox).
- Array — Sort the classes in a unique order specified by a categorical vector, numeric vector, string vector, character array, cell array of character vectors, or logical vector. The array must be a permutation of the `ClassLabels` property of the confusion matrix chart.

Example: `sortClasses(cm,'ascending-diagonal')`
Example: `sortClasses(cm,['owl','cat','toad'])`

See Also

Functions
categorical | confusionchart

Properties
ConfusionMatrixChart Properties

Topics
“Deep Learning in MATLAB”

Introduced in R2018b
confusionmat

Compute confusion matrix for classification problem

Syntax

C = confusionmat(group,grouphat)
C = confusionmat(group,grouphat,'Order',grouporder)
[C,order] = confusionmat(___)

Description

C = confusionmat(group,grouphat) returns the confusion matrix C determined by the known and predicted groups in group and grouphat, respectively.

C = confusionmat(group,grouphat,'Order',grouporder) uses grouporder to order the rows and columns of C.

[C,order] = confusionmat(___) also returns the order of the rows and columns of C in the variable order using any of the input arguments in previous syntaxes.

Examples

Calculate Confusion Matrix

Load a sample of predicted and true labels for a classification problem. trueLabels are the true labels for an image classification problem and predictedLabels are the predictions of a convolutional neural network.

load('Cifar10Labels.mat','trueLabels','predictedLabels');

Calcualte the numeric confusion matrix. order is the order of the classes in the confusion matrix.

[m,order] = confusionmat(trueLabels,predictedLabels)
m = 10×10

923  4  21  8  4  1  5  5  23  6
 5  972  2  0  0  0  0  1  5  15
26  2  892 30 13  8 17  5  4  3
12  4  32 826 24 48 30 12  5  7
 5  1  28 24 898 13 14 14  2  1
 7  2  28 111 18 801 13 17  0  3
 5  0  16 27  3  4 943  1  1  0
 9  1  14 13 22 17  3 915  2  4
37 10  4  4  0  1  2  1  931 10
20 39  3  3  0  0  2  1  9 923

order = 10x1 categorical array
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

You can use confusionchart to plot a the confusion matrix as a confusion matrix chart.

figure
cm = confusionchart(m,order);
You do not need to calculate the confusion matrix first and then plot it. Instead, plot a confusion matrix chart directly from the true and predicted labels. You can also add column and row summaries and a title.

```matlab
figure
cm = confusionchart(trueLabels,predictedLabels, ...
    'Title','My Title', ...)
    'RowSummary','row-normalized', ...
    'ColumnSummary','column-normalized');
```
The ConfusionMatrixChart object stores the numeric confusion matrix in the NormalizedValues property and classes in the ClassLabels property.

```
cm.NormalizedValues
ans = 10×10

923     4    21     8     4     1     5     5    23     6
 5    972     2     0     0     0     1     5    15
26     2    892    30    13     8    17     5     4    3
12     3     2    826    24    48    30    12     5     7
 5     1     2    28    898    13    14    14     2     1
 7     2     8    111    801    13    17     0     3
 5     0     1    27     3     4    943     1     1     0
```
Input Arguments

group — Known groups
d numeric vector | logical vector | character array | string array | cell array of character vectors | categorical vector

Known groups for categorizing observations, specified as a numeric vector, logical vector, character array, string array, cell array of character vectors, or categorical vector.

group is a grouping variable of the same type as grouphat. The group argument must have the same number of observations as grouphat, as described in “Grouping Variables” (Statistics and Machine Learning Toolbox). The confusionmat function treats character arrays and string arrays as cell arrays of character vectors. Additionally, confusionmat treats NaN, empty, and 'undefined' values in group as missing values and does not count them as distinct groups or categories.

Example: {'Male','Female','Female','Male','Female'}

Data Types: single | double | logical | char | string | cell | categorical
**grouphat — Predicted groups**

numeric vector | logical vector | character array | string array | cell array of character vectors | categorical vector

Predicted groups for categorizing observations, specified as a numeric vector, logical vector, character array, string array, cell array of character vectors, or categorical vector.

grouphat is a grouping variable of the same type as group. The grouphat argument must have the same number of observations as group, as described in “Grouping Variables” (Statistics and Machine Learning Toolbox). The confusionmat function treats character arrays and string arrays as cell arrays of character vectors. Additionally, confusionmat treats NaN, empty, and 'undefined' values in grouphat as missing values and does not count them as distinct groups or categories.

Example: [1 0 0 1 0]

Data Types: single | double | logical | char | string | cell | categorical

**grouporder — Group order**

numeric vector | logical vector | character array | string array | cell array of character vectors | categorical vector

Group order, specified as a numeric vector, logical vector, character array, string array, cell array of character vectors, or categorical vector.

grouporder is a grouping variable containing all the distinct elements in group and grouphat. Specify grouporder to define the order of the rows and columns of C. If grouporder contains elements that are not in group or grouphat, the corresponding entries in C are 0.

By default, the group order depends on the data type of s = [group;grouphat]:

- For numeric and logical vectors, the order is the sorted order of s.
- For categorical vectors, the order is the order returned by categories(s).
- For other data types, the order is the order of first appearance in s.

Example: 'order',{'setosa','versicolor','virginica'}

Data Types: single | double | logical | char | string | cell | categorical
Output Arguments

C — Confusion matrix
matrix

Confusion matrix, returned as a square matrix with size equal to the total number of
distinct elements in the group and grouphat arguments. $C(i,j)$ is the count of
observations known to be in group $i$ but predicted to be in group $j$.

The rows and columns of $C$ have identical ordering of the same group indices. By default,
the group order depends on the data type of $s = [\text{group}; \text{grouphat}]$:

- For numeric and logical vectors, the order is the sorted order of $s$.
- For categorical vectors, the order is the order returned by $\text{categories}(s)$.
- For other data types, the order is the order of first appearance in $s$.

To change the order, specify $\text{grouporder}$,

The $\text{confusionmat}$ function treats NaN, empty, and 'undefined' values in the grouping
variables as missing values and does not include them in the rows and columns of $C$.

order — Order of rows and columns
numeric vector | logical vector | categorical vector | cell array of character vectors

Order of rows and columns in $C$, returned as a numeric vector, logical vector, categorical
vector, or cell array of character vectors. If $\text{group}$ and $\text{grouphat}$ are character arrays,
string arrays, or cell arrays of character vectors, then the variable $\text{order}$ is a cell array of
character vectors. Otherwise, $\text{order}$ is of the same type as $\text{group}$ and $\text{grouphat}$.

Alternative Functionality

- Use $\text{confusionchart}$ to calculate and plot a confusion matrix. Additionally,$\text{confusionchart}$ displays summary statistics about your data and sorts the classes of
the confusion matrix according to the class-wise precision (positive predictive value),
class-wise recall (true positive rate), or total number of correctly classified
observations.
See Also
categories | classify | confusionchart

Topics
“Deep Learning in MATLAB”
**dlarray**

Deep learning array for custom training loops

**Description**

A deep learning array stores data with optional data format labels for custom training loops, and enables functions to compute and use derivatives through automatic differentiation.

**Tip** For most deep learning tasks, you can use a pretrained network and adapt it to your own data. For an example showing how to use transfer learning to retrain a convolutional neural network to classify a new set of images, see “Train Deep Learning Network to Classify New Images”. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Custom Training Loops”.

**Creation**

**Syntax**

```
dlX = dlarray(X)
dlX = dlarray(X,fmt)
dlX = dlarray(v,dim)
```

**Description**

`dlX = dlarray(X)` returns a `dlarray` object representing `X`. If `X` is a `dlarray`, `dlX` is a copy of `X`. 
dlX = dlarray(X,fmt) labels the data in dlX according to the data format in fmt. Labels help in passing deep learning data between functions. See “Usage” on page 1-1521. If X is a labeled dlarray, then fmt replaces the existing labels.

dlX = dlarray(v,dim) accepts a vector v and a single character format dim, and returns a column vector dlarray. The first dimension of dlX has the label dim, and the second (singleton) dimension has the label 'U'.

**Input Arguments**

- **X** — Data array
  numeric array of data type double or single | logical array | gpuArray object | dlarray object

  Data array, specified as a numeric array of data type double or single, logical array, gpuArray object, or dlarray object. X must be full, not sparse, and must be real, not complex.
  
  Example: rand(31*23,23)

- **fmt** — Data format
  character vector | string scalar

  Data format, specified as a character vector or string scalar. Each character in fmt must be one of these labels:

  - S — Spatial
  - C — Channel
  - B — Batch observations
  - T — Time or sequence
  - U — Unspecified

  You can specify any number of S and U labels. You can specify at most one of each of the C, B, and T labels.

  Each element of fmt labels the matching dimension of dlX. If fmt is not in the listed order ('S' followed by 'C' and so on), then dlarray implicitly permutes both fmt and the data to match the order, but without changing the storage of the data.
\texttt{fmt} must have at least the same number of labels as the number of dimensions of \texttt{dlX}. If you specify more than that number of labels, \texttt{dlarray} creates empty (singleton) dimensions for the additional labels.

For information on \texttt{fmt}, see “Usage” on page 1-1521.

Example: "SSB"

Example: 'CBUSS', which \texttt{dlarray} reorders to 'SSCBU'

\textbf{\texttt{v} — Data vector}

numeric vector of data type double or single | logical vector | \texttt{dlarray} vector object

Data vector, specified as a numeric vector of data type double or single, logical vector, \texttt{gpuArray} vector object, or \texttt{dlarray} vector object. Here, "vector" means any array with exactly one nonsingleton dimension.

Example: \texttt{rand(100,1)}

\textbf{\texttt{dim} — Dimension label}

single character

Dimension label, specified as a single character of the type allowed for \texttt{fmt}.

Example: "S"

Example: 'S'

\section*{Output Arguments}

\textbf{\texttt{dlX} — Deep learning array}

\texttt{dlarray} object

Deep learning array, returned as a \texttt{dlarray} object. \texttt{dlX} enables automatic differentiation using \texttt{dlgradient} and \texttt{dlfeval}. If you supply the \texttt{fmt} argument, \texttt{dlX} has labels.

- If \texttt{X} is a numeric or logical array, \texttt{dlX} contains its data, possibly reordered because of labels in \texttt{fmt}.
- If \texttt{X} is a \texttt{gpuArray}, the data in \texttt{dlX} is also on the GPU. Subsequent calculations using \texttt{dlX} are performed on the GPU.
Usage

dlarray labels enable you to use the functions in this table to execute with assurance that the data has the appropriate format.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
<th>Validates Input Dimension</th>
<th>Affects Size of Input Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>avgpool</td>
<td>Compute the average of the input data over moving rectangular (or cuboidal) spatial ('S') regions defined by a pool size parameter.</td>
<td>'S'</td>
<td>'S'</td>
</tr>
<tr>
<td>batchnorm</td>
<td>Normalize the values contained in each channel ('C') of the input data.</td>
<td>'C'</td>
<td></td>
</tr>
<tr>
<td>crossentropy</td>
<td>Compute the cross-entropy between estimates and target values, averaged by the size of the batch ('B') dimension.</td>
<td>'S', 'C', 'B', 'T', 'U'</td>
<td>'S', 'C', 'B', 'T', 'U' (The output is an unlabeled scalar.)</td>
</tr>
<tr>
<td>dlconv</td>
<td>Compute the deep learning convolution of the input data using an array of filters, matching the number of spatial ('S') and (a function of the) channel ('C') dimensions of the input, and adding a constant bias.</td>
<td>'S', 'C'</td>
<td>'S', 'C'</td>
</tr>
<tr>
<td>Function</td>
<td>Operation</td>
<td>Validates Input Dimension</td>
<td>Affects Size of Input Dimension</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>dltranspconv</td>
<td>Compute the deep learning transposed convolution of the input data using an array of filters, matching the number of spatial ('S') and (a function of the) channel ('C') dimensions of the input, and adding a constant bias.</td>
<td>'S', 'C'</td>
<td>'S', 'C'</td>
</tr>
<tr>
<td>fullyconnect</td>
<td>Compute a weighted sum of the input data and apply a bias for each batch ('B') and time ('T') dimension.</td>
<td>'S', 'C', 'U'</td>
<td>'S', 'C', 'B', 'T', 'U' (The output always has labels 'CB', 'CT', or 'CTB'.)</td>
</tr>
<tr>
<td>lstm</td>
<td>Apply a long short-term memory calculation to the input data.</td>
<td>'S', 'C', 'T'</td>
<td>'C'</td>
</tr>
<tr>
<td>maxpool</td>
<td>Compute the maximum of the input data over moving rectangular spatial ('S') regions defined by a pool size parameter.</td>
<td>'S'</td>
<td>'S'</td>
</tr>
<tr>
<td>maxunpool</td>
<td>Compute the unpooling operation over the spatial ('S') dimensions.</td>
<td>'S'</td>
<td>'S'</td>
</tr>
<tr>
<td>mse</td>
<td>Compute the half mean squared error between estimates and target values, averaged by the size of the batch ('B') dimension.</td>
<td>'S', 'C', 'B', 'T', 'U' (Estimates and target arrays must have the same sizes.)</td>
<td>'S', 'C', 'B', 'T', 'U' (The output is an unlabeled scalar.)</td>
</tr>
<tr>
<td>softmax</td>
<td>Apply the softmax activation to each channel ('C') of the input data.</td>
<td>'C'</td>
<td></td>
</tr>
</tbody>
</table>

1-1522
These functions require each dimension to have a label, specified either as the labels of their first dlarray input, or as the 'DataFormat' name-value pair argument containing dimension labels.

dlarray enforces the order of labels 'SCBTU'. This enforcement eliminates ambiguous semantics in operations, which implicitly match labels between inputs. dlarray also enforces that the labels 'C', 'B', and 'T' can each appear at most once. The functions that use these labels accept at most one dimension for each label.

dlarray provides functions for removing labels (stripdims), obtaining the dimensions associated with labels (finddim), and listing the labels associated with a dlarray (dims).

For more information on how a dlarray behaves with labels, see “Notable dlarray Behaviors”.

**Object Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avgpool</td>
<td>Pool data to average values over spatial dimensions</td>
</tr>
<tr>
<td>batchnorm</td>
<td>Normalize each channel of input data</td>
</tr>
<tr>
<td>crossentropy</td>
<td>Categorical cross-entropy loss</td>
</tr>
<tr>
<td>dms</td>
<td>Dimension labels of dlarray</td>
</tr>
<tr>
<td>dlconv</td>
<td>Deep learning convolution</td>
</tr>
<tr>
<td>dlgradient</td>
<td>Compute gradients for custom training loops using automatic differentiation</td>
</tr>
<tr>
<td>dltranspconv</td>
<td>Deep learning transposed convolution</td>
</tr>
<tr>
<td>extractdata</td>
<td>Extract data from dlarray</td>
</tr>
<tr>
<td>finddim</td>
<td>Find dimensions with specified label</td>
</tr>
<tr>
<td>fullyconnect</td>
<td>Sum all weighted input data and apply a bias</td>
</tr>
<tr>
<td>leakyrelu</td>
<td>Apply leaky rectified linear unit activation</td>
</tr>
<tr>
<td>lstm</td>
<td>Long short-term memory</td>
</tr>
<tr>
<td>maxpool</td>
<td>Pool data to maximum value</td>
</tr>
<tr>
<td>maxunpool</td>
<td>Unpool the output of a maximum pooling operation</td>
</tr>
<tr>
<td>mse</td>
<td>Half mean squared error</td>
</tr>
<tr>
<td>relu</td>
<td>Apply rectified linear unit activation</td>
</tr>
<tr>
<td>sigmoid</td>
<td>Apply sigmoid activation</td>
</tr>
<tr>
<td>softmax</td>
<td>Apply softmax activation to channel dimension</td>
</tr>
<tr>
<td>stripdims</td>
<td>Remove dlarray labels</td>
</tr>
</tbody>
</table>

A dlarray also allows functions for numeric, matrix, and other operations. See the full list in “List of Functions with dlarray Support”.
Examples

Create Unlabeled dlarray

Create an unlabeled dlarray from a matrix.

```matlab
rng default % For reproducibility
X = randn(3,5);
dlX = dlarray(X)
dlX = dlarray(0.5377, 0.8622, -0.4336, 2.7694, 0.7254)
    1.8339, 0.3188, 0.3426, -1.3499, -0.0631)
    -2.2588, -1.3077, 3.5784, 3.0349, 0.7147)
```

Create Labeled dlarray

Create a dlarray that has a data format with the labels 'S' and 'C'.

```matlab
rng default % For reproducibility
X = randn(3,5);
dlX = dlarray(X, 'SC')
dlX = dlarray(0.5377, 0.8622, -0.4336, 2.7694, 0.7254)
    1.8339, 0.3188, 0.3426, -1.3499, -0.0631)
    -2.2588, -1.3077, 3.5784, 3.0349, 0.7147)
```

If you specify the labels in the opposite order, dlarray implicitly reorders the underlying data.

```matlab
dlX = dlarray(X, 'CS')
dlX = dlarray(0.5377, 0.8622, -0.4336, 2.7694, 0.7254)
    1.8339, 0.3188, 0.3426, -1.3499, -0.0631)
    -2.2588, -1.3077, 3.5784, 3.0349, 0.7147)
```
Create Labeled `dlarray` Vector

Create a `dlarray` vector with the first label 'T'. The second label, which `dlarray` creates automatically, is 'U'.

```matlab
rng default % For reproducibility
X = randn(6,1);
dlX = dlarray(X,'T')
```

```
dlX = 6(T) x 1(U) dlarray
   0.5377
   1.8339
  -2.2588
   0.8622
   0.3188
  -1.3077
```

If you specify a row vector for `X`, `dlarray` implicitly reorders the result to be a column vector.

```matlab
X = X';
dlX = dlarray(X,'T')
```

```
dlX = 6(T) x 1(U) dlarray
   0.5377
   1.8339
  -2.2588
   0.8622
   0.3188
   -1.3077
```
Tips

• A dlgradient call must be inside a function. To obtain a numeric value of a gradient, you must evaluate the function using dlfeval, and the argument to the function must be a dlarray. See “Use Automatic Differentiation In Deep Learning Toolbox”.
• To enable the correct evaluation of gradients, dlfeval must call functions that use only supported functions for dlarray. See “List of Functions with dlarray Support”.

See Also
dims | dlfeval | dlgradient | dlnetwork | finddim | stripdims

Topics
“Train Generative Adversarial Network (GAN)”
“Grad-CAM Reveals the Why Behind Deep Learning Decisions”
“Define Custom Training Loops, Loss Functions, and Networks”
“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”

Introduced in R2019b
dlgradient

Compute gradients for custom training loops using automatic differentiation

Syntax

[dydx1,...,dydxk] = dlgradient(y,x1,...,xk)
[dydx1,...,dydxk] = dlgradient(y,x1,...,xk,'RetainData',true)

Description

Use dlgradient to compute derivatives using automatic differentiation for custom training loops.

**Tip** For most deep learning tasks, you can use a pretrained network and adapt it to your own data. For an example showing how to use transfer learning to retrain a convolutional neural network to classify a new set of images, see “Train Deep Learning Network to Classify New Images”. Alternatively, you can create and train networks from scratch using layerGraph objects with the trainNetwork and trainingOptions functions.

If the trainingOptions function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Custom Training Loops”.

[dydx1,...,dydxk] = dlgradient(y,x1,...,xk) returns the gradients of y with respect to the variables x1 through xk.

Call dlgradient from inside a function passed to dlfeval. See “Compute Gradient Using Automatic Differentiation” on page 1-1528 and “Use Automatic Differentiation In Deep Learning Toolbox”.

[dydx1,...,dydxk] = dlgradient(y,x1,...,xk,'RetainData',true) causes the gradient to retain intermediate values for reuse in subsequent dlgradient calls. This syntax can save time, but uses more memory. See “Tips” on page 1-1532.
Examples

Compute Gradient Using Automatic Differentiation

Rosenbrock's function is a standard test function for optimization. The `rosenbrock.m` helper function computes the function value and uses automatic differentiation to compute its gradient.

```matlab
type rosenbrock.m
function [y,dydx] = rosenbrock(x)
y = 100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2;
dydx = dlgradient(y,x);
end
```

To evaluate Rosenbrock's function and its gradient at the point \([-1,2]\), create a `dlarray` of the point and then call `dlfeval` on the function handle `@rosenbrock`.

```matlab
x0 = dlarray([-1,2]);
[fval,gradval] = dlfeval(@rosenbrock,x0)
```

```
fval =
  1x1 dlarray

    104

gradval =
  1x2 dlarray

    396   200
```

Alternatively, define Rosenbrock's function as a function of two inputs, \(x_1\) and \(x_2\).

```matlab
type rosenbrock2.m
function [y,dydx1,dydx2] = rosenbrock2(x1,x2)
y = 100*(x2 - x1.^2).^2 + (1 - x1).^2;
[dydx1,dydx2] = dlgradient(y,x1,x2);
```
Call dlfeval to evaluate rosenbrock2 on two dlarray arguments representing the inputs –1 and 2.

```matlab
x1 = dlarray(-1);
x2 = dlarray(2);
[fval,dydx1,dydx2] = dlfeval(@rosenbrock2,x1,x2)
```

```matlab
fval =
    1x1 dlarray

    104

dydx1 =
    1x1 dlarray

    396

dydx2 =
    1x1 dlarray

    200
```

Plot the gradient of Rosenbrock's function for several points in the unit square. First, initialize the arrays representing the evaluation points and the output of the function.

```matlab
[X1 X2] = meshgrid(linspace(0,1,10));
X1 = dlarray(X1(:));
X2 = dlarray(X2(:));
Y = dlarray(zeros(size(X1)));
DYDX1 = Y;
DYDX2 = Y;
```

Evaluate the function in a loop. Plot the result using quiver.

```matlab
for i = 1:length(X1)
    [Y(i),DYDX1(i),DYDX2(i)] = dlfeval(@rosenbrock2,X1(i),X2(i));
end
quiver(extractdata(X1),extractdata(X2),extractdata(DYDX1),extractdata(DYDX2))
xlabel('x1')
ylabel('x2')
```
Input Arguments

**y — Variable to differentiate**

scalar dlarray object

Variable to differentiate, specified as a scalar dlarray object. For differentiation, y must be a traced function of dlarray inputs (see “Traced dlarray” on page 1-1532) and must consist of supported functions for dlarray (see “List of Functions with dlarray Support”).

Example: `100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2`

Example: `relu(X)`
x — Variable in function
dlarray object | cell array containing dlarray objects | structure containing dlarray objects | table containing dlarray objects

Variable in the function, specified as a dlarray object, a cell array, structure, or table containing dlarray objects, or any combination of such arguments recursively. For example, an argument can be a cell array containing a cell array that contains a structure containing dlarray objects.

Example: dlarray([1 2;3 4])

Data Types: single | double | logical | struct | cell

'RretainData' — Indicator for retaining trace data during function call
false (default) | true

Indicator for retaining trace data during the function call, specified as false or true. When this argument is false, a dlarray discards the derivative trace immediately after computing a derivative. When this argument is true, a dlarray retains the derivative trace until the end of the dlfeval function call that evaluates the dlgradient. The true setting is useful only when the dlfeval call contains more than one dlgradient call. The true setting causes the software to use more memory, but can save time when multiple dlgradient calls use at least part of the same trace.

Example: dydx = dlgradient(y,x,'RetainData',true)

Data Types: logical

Output Arguments

dydx — Gradient
dlarray object | cell array containing dlarray objects | structure containing dlarray objects | table containing dlarray objects

Gradient, returned as a dlarray object, or a cell array, structure, or table containing dlarray objects, or any combination of such arguments recursively. The size and data type of dydx are the same as those of the associated input variable x.
More About

Traced dlarray

During the computation of a function, a dlarray internally records the steps taken in a trace, enabling reverse mode automatic differentiation. The trace occurs within a dlfeval call. See “Automatic Differentiation Background”.

Tips

- dlgradient does not support higher order derivatives. In other words, you cannot pass the output of a dlgradient call into another dlgradient call.
- A dlgradient call must be inside a function. To obtain a numeric value of a gradient, you must evaluate the function using dlfeval, and the argument to the function must be a dlarray. See “Use Automatic Differentiation In Deep Learning Toolbox”.
- To enable the correct evaluation of gradients, the y argument must use only supported functions for dlarray. See “List of Functions with dlarray Support”.
- If you set the ‘RetainData’ name-value pair argument to true, the software preserves tracing for the duration of the dlfeval function call instead of erasing the trace immediately after the derivative computation. This preservation can cause a subsequent dlgradient call within the same dlfeval call to be executed faster, but uses more memory. For example, in training an adversarial network, the ‘RetainData’ setting is useful because the two networks share data and functions during training. See “Train Generative Adversarial Network (GAN)”.

See Also
dlarray | dlfeval

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”
“Train Generative Adversarial Network (GAN)”
“Grad-CAM Reveals the Why Behind Deep Learning Decisions”
Introduced in R2019b
**dlfeval**

Evaluate deep learning model for custom training loops

**Syntax**

\[ y_1, \ldots, y_k = \text{dlfeval}(fun, x_1, \ldots, x_n) \]

**Description**

Use `dlfeval` to evaluate custom deep learning models for custom training loops.

**Tip**  For most deep learning tasks, you can use a pretrained network and adapt it to your own data. For an example showing how to use transfer learning to retrain a convolutional neural network to classify a new set of images, see “Train Deep Learning Network to Classify New Images”. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Custom Training Loops”.

\[ y_1, \ldots, y_k = \text{dlfeval}(fun, x_1, \ldots, x_n) \] evaluates the deep learning array function `fun` at the input arguments `x_1, ..., x_n`. Functions passed to `dlfeval` can contain calls to `dlgradient`, which compute gradients from the inputs `x` by using automatic differentiation.

**Examples**
Compute Gradient Using Automatic Differentiation

Rosenbrock's function is a standard test function for optimization. The rosenbrock.m helper function computes the function value and uses automatic differentiation to compute its gradient.

type rosenbrock.m

function [y,dydx] = rosenbrock(x)
    y = 100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2;
    dydx = dlgradient(y,x);
end

To evaluate Rosenbrock's function and its gradient at the point \([-1, 2]\), create a dlarray of the point and then call dlfeval on the function handle @rosenbrock.

x0 = dlarray([-1, 2]);
[fval,gradval] = dlfeval(@rosenbrock,x0)

fval =
    1x1 dlarray
        104

gradval =
    1x2 dlarray
        396   200

Alternatively, define Rosenbrock's function as a function of two inputs, x1 and x2.

type rosenbrock2.m

function [y,dydx1,dydx2] = rosenbrock2(x1,x2)
    y = 100*(x2 - x1.^2).^2 + (1 - x1).^2;
    [dydx1,dydx2] = dlgradient(y,x1,x2);
end

Call dlfeval to evaluate rosenbrock2 on two dlarray arguments representing the inputs \(-1\) and \(2\).
x1 = dlarray(-1);
x2 = dlarray(2);
[fval,dydx1,dydx2] = dlfeval(@rosenbrock2,x1,x2)

fval =
  1×1 dlarray
       104

dydx1 =
  1×1 dlarray
       396

dydx2 =
  1×1 dlarray
       200

Plot the gradient of Rosenbrock's function for several points in the unit square. First, initialize the arrays representing the evaluation points and the output of the function.

[X1 X2] = meshgrid(linspace(0,1,10));
X1 = dlarray(X1(:));
X2 = dlarray(X2(:));
Y = dlarray(zeros(size(X1)));
DYDX1 = Y;
DYDX2 = Y;

Evaluate the function in a loop. Plot the result using `quiver`.

for i = 1:length(X1)
    [Y(i),DYDX1(i),DYDX2(i)] = dlfeval(@rosenbrock2,X1(i),X2(i));
end
quiver(extractdata(X1),extractdata(X2),extractdata(DYDX1),extractdata(DYDX2))
xlabel('x1')
ylabel('x2')
Input Arguments

fun — Function to evaluate
function handle

Function to evaluate, specified as a function handle. If fun includes a dlgradient call, then dlfeval evaluates the gradient by using automatic differentiation. In this gradient evaluation, each argument of the dlgradient call must be a dlarray or a cell array, structure, or table containing a dlarray. The number of input arguments to dlfeval must be the same as the number of input arguments to fun.
Example: @rosenbrock

Data Types: function_handle

\[ \mathbf{x} \rightarrow \textbf{Function argument} \]
any MATLAB data type

Function argument, specified as any MATLAB data type.

An input argument \( x_j \) that is a variable of differentiation in a \( \text{dlgradient} \) call must be a traced \( \text{dlarray} \) or a cell array, structure, or table containing a traced \( \text{dlarray} \). An extra variable such as a hyperparameter or constant data array does not have to be a \( \text{dlarray} \).

Example: \( \text{dlarray}([1 2;3 4]) \)

Data Types: \( \text{single} \mid \text{double} \mid \text{int8} \mid \text{int16} \mid \text{int32} \mid \text{int64} \mid \text{uint8} \mid \text{uint16} \mid \text{uint32} \mid \text{uint64} \mid \text{logical} \mid \text{char} \mid \text{string} \mid \text{struct} \mid \text{table} \mid \text{cell} \mid \text{function_handle} \mid \text{categorical} \mid \text{datetime} \mid \text{duration} \mid \text{calendarDuration} \mid \text{fi} \)

Output Arguments

\[ \mathbf{y} \rightarrow \textbf{Function output} \]
any data type | \( \text{dlarray} \)

Function output, returned as any data type. If the output results from a \( \text{dlgradient} \) call, the output is a \( \text{dlarray} \).

Tips

- A \( \text{dlgradient} \) call must be inside a function. To obtain a numeric value of a gradient, you must evaluate the function using \( \text{dlfeval} \), and the argument to the function must be a \( \text{dlarray} \). See “Use Automatic Differentiation In Deep Learning Toolbox”.
- \( \text{dlgradient} \) does not support higher order derivatives. In other words, you cannot pass the output of a \( \text{dlgradient} \) call into another \( \text{dlgradient} \) call.
- To enable the correct evaluation of gradients, the function \( \text{fun} \) must use only supported functions for \( \text{dlarray} \). See “List of Functions with \text{dlarray} Support”.
See Also
dlarray | dlgradient

Topics
“Automatic Differentiation Background”
“Use Automatic Differentiation In Deep Learning Toolbox”
“List of Functions with dlarray Support”
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Generative Adversarial Network (GAN)”
“Grad-CAM Reveals the Why Behind Deep Learning Decisions”

Introduced in R2019b
**dims**

Dimension labels of `dlarray`

**Syntax**

```matlab
 d = dims(dlX)
```

**Description**

`d = dims(dlX)` returns the labels of `dlX` as a character array.

**Examples**

**Obtain Dimension Labels**

Obtain the dimension labels of a `dlarray`.

```matlab
dlX = dlarray(randn(3,4), 'TS');
d = dims(dlX)
```

```
d =
'ST'
```

Obtain the labels of an unlabeled `dlarray`.

```matlab
y = stripdims(dlX);
d = dims(y)
```

```
d =
0x0 empty char array
```
Input Arguments

dlX — Input dlarray
dlarray object

Input dlarray, specified as a dlarray object.
Example: dlX = dlarray(randn(3,4),'ST')

Output Arguments

d — Dimension labels
character vector

Dimension labels, returned as a character vector. If the input dlX is unlabeled, d is empty.

See Also
dlarray | finddim | stripdims

Introduced in R2019b
extractdata

Extract data from dlarray

Syntax

\[ y = \text{extractdata}(\text{dlX}) \]

Description

\[ y = \text{extractdata}(\text{dlX}) \] returns the data in the \text{dlarray} \text{dlX}. The output \( y \) has the same data type as the data in \text{dlX} and is unlabeled

Examples

Extract Data from dlarray

Create a logical dlarray labeled 'SS'.

\begin{verbatim}
rng default \% For reproducibility
dlX = dlarray(rand(4,3) > 0.5,'SS')
\end{verbatim}

\[ dlX = \]
\[
4(\text{S}) \times 3(\text{S}) \text{ logical dlarray}
\]

\[
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 0 & 1 \\
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

Extract the data from \text{dlX}.

\[ y = \text{extractdata}(\text{dlX}) \]

\[ y = 4 \times 3 \text{ logical array} \]
Input Arguments

dlX — Input dlarray
dlarray object

Input dlarray, specified as a dlarray object.
Example: dlX = dlarray(randn(50,3),'SC')

Output Arguments

y — Data array
single array | double array | logical array | gpuArray

Data array, returned as a single, double, or logical array, or as a gpuArray of one of these array types. The output y has the same data type as the underlying data type in dlX. The output y is unlabeled.

Tips

• If dlX contains an implicit permutation because of labeling, y has that permutation explicitly.
• The output y has no tracing for the computation of derivatives. See “Derivative Trace”.

See Also
dlarray | gather

Introduced in R2019b
**finddim**

Find dimensions with specified label

**Syntax**

```matlab
dim = finddim(dlX,label)
```

**Description**

`dim = finddim(dlX,label)` returns the dimensions in `dlX` that have the label `label`. If no dimension matches `label`, `dim` is empty.

**Examples**

**Obtain Dimension with Specified Labels**

Create a `dlarray` with some repeated labels. Specify the labels as 'TSSU'. The `dlarray` call reorders the labels, because it enforces the order 'SCBTU'. See “Usage” on page 1-1521.

```matlab
dlX = dlarray(randn(5,4,3,2),'TSSU');
```

Obtain the dimensions with the label 'T'.

```matlab
dimU = finddim(dlX,'T')
dimU = 3
```

Obtain the dimensions with the label 'S'.

```matlab
dimS = finddim(dlX,'S')
dimS = 1x2
```

```matlab
    1     2
```
Obtain the dimensions with the label 'B'.

dimB = finddim(dlX,'B')

dimB =
    1x0 empty double row vector

Obtain the size of the dlX dimensions labeled 'S'.

SSize = size(dlX,finddim(dlX,'S'))

SSize = 1x2
    4    3

Input Arguments

dlX — Input dlarray
dlarray object

Input dlarray, specified as a dlarray object.
Example: dlX = dlarray(randn(3,4),'ST')

label — Single dlarray label
'S'  | 'C'  | 'B'  | 'T'  | 'U'

Single dlarray label, specified as one of these characters:

- S — Spatial
- C — Channel
- B — Batch observations
- T — Time or sequence
- U — Unspecified

Example: "C"
Data Types: char | string
Output Arguments

**dim — Dimension**  
real vector

Dimension, returned as a real vector. If no label in the input array dlX matches label, \texttt{dim} is empty. So if dlX is unlabeled, \texttt{dim} is empty.

**See Also**

dims | dlarray | stripdims

*Introduced in R2019b*
**functionToLayerGraph**

Convert deep learning model function to a layer graph

**Syntax**

\[ \text{lgraph} = \text{functionToLayerGraph}(\text{fun}, \text{x}) \]

**Description**

\[ \text{lgraph} = \text{functionToLayerGraph}(\text{fun}, \text{x}) \]
returns a layer graph based on the deep learning array function \( \text{fun} \). \text{functionToLayerGraph} converts only those operations in \( \text{fun} \) that operate on \text{dlarray} objects among the inputs in \( \text{x} \). To include extra parameters or data in \( \text{fun} \), see the topic “Parameterizing Functions” (MATLAB) or the example “Create Layer Graph from Function” on page 1-1547.

\text{functionToLayerGraph} evaluates \( \text{fun}(\text{x}) \) and traces the execution to derive an equivalent layer graph, to the extent possible. The steps in \( \text{fun}(\text{x}) \) that \text{functionToLayerGraph} can trace are both based on \text{dlarray} arguments and are supported calls for \text{dlarray}. See “List of Functions with \text{dlarray} Support”. For unsupported functions, \text{functionToLayerGraph} creates a \text{PlaceholderLayer}.

**Examples**

**Create Layer Graph from Function**

The \text{simplemodel} function at the end of this example creates fully connected outputs followed by a \text{softmax} operation. To create a layer graph from this function based on \text{dlarray} data, create input arrays as \text{dlarray} objects, and create a function handle to the \text{simplemodel} function including the data.

\begin{verbatim}
    rng default % For reproducibility
    dlX1 = dlarray(rand(10),'CB');
    dlX2 = dlarray(zeros(10,1),'CB');
    fun = @(x) simplemodel(x,dlX1,dlX2);
\end{verbatim}
Call `functionToLayerGraph` using a `dlarray` for the input data `dlX`.

```matlab
dlX = dlarray(ones(10,1),'CB');
lgraph = functionToLayerGraph(fun,dlX)
```

```
lgraph = 
    LayerGraph with properties:

    Layers: [2x1 nnet.cnn.layer.Layer]
    Connections: [1x2 table]
    InputNames: {1x0 cell}
    OutputNames: {1x0 cell}
```

Examine the resulting layers in `lgraph`.

```matlab
disp(lgraph.Layers)
```

```
2x1 Layer array with layers:

1    'fc_1'    Fully Connected    10 fully connected layer
2    'sm_1'    Softmax          softmax
```

```matlab
function y = simplemodel(x,w,b)
y = fullyconnect(x,w,b);
y = softmax(y);
end
```

### Input Arguments

- **fun** — Function to convert
  
  Function handle

  Function to convert, specified as a function handle.

  Example: `@relu`

  Data Types: `function_handle`

- **x** — Data for function
  
  any data type
Data for the function, specified as any data type. Only dlarray data is traced and converted to a layer graph.

Example: dlarray(zeros(12*50,23))

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |
uint32 | uint64 | logical | char | string | struct | table | cell |
function_handle | categorical | datetime | duration | calendarDuration | fi

Output Arguments

lgraph — Layer graph
LayerGraph object

Layer graph, returned as a LayerGraph object.

When a portion of fun is unsupported, lgraph contains PlaceholderLayer layers representing the unsupported functionality. To create a working network in this case, see “Define Custom Deep Learning Layers” or “Define Custom Networks”.

See Also

PlaceholderLayer | dlarray | findPlaceholderLayers | layerGraph

Topics

“List of Functions with dlarray Support”

Introduced in R2019b
stripdims

Remove dlarray labels

Syntax

\[ y = \text{stripdims}(\text{dlX}) \]

Description

\[ y = \text{stripdims}(\text{dlX}) \] returns the dlarray \( \text{dlX} \) without any labels.

Examples

Remove Labels from dlarray

Create a labeled dlarray.

\[ \text{dlX} = \text{dlarray}([\text{randn}(3,2,1,2)],'SSTU') \]

\[ \text{dlX} = \\
3(S) \times 2(S) \times 1(T) \times 2(U) \text{ dlarray} \]

\[ (\text{:,},1,1) = \\
0.5377 \hspace{1em} 0.8622 \\
1.8339 \hspace{1em} 0.3188 \\
-2.2588 \hspace{1em} -1.3077 \]

\[ (\text{:,},1,2) = \\
-0.4336 \hspace{1em} 2.7694 \\
0.3426 \hspace{1em} -1.3499 \\
3.5784 \hspace{1em} 3.0349 \]
Create an array that is the same as \( \text{dlX} \) but has no labels.

\[
y = \text{stripdims}(\text{dlX})
\]

\[
y = \\
3 \times 2 \times 1 \times 2 \text{ dlarray}
\]

\[
(:,:,1,1) = \\
0.5377 0.8622 \\
1.8339 0.3188 \\
-2.2588 -1.3077
\]

\[
(:,:,1,2) = \\
-0.4336 2.7694 \\
0.3426 -1.3499 \\
3.5784 3.0349
\]

**Input Arguments**

\( \text{dlX} \) — Input dlarray

dlarray object

Input dlarray, specified as a dlarray object.

Example: \( \text{dlX} = \text{dlarray}([\text{randn}(3,4)],'ST') \)

**Output Arguments**

\( y \) — Unlabeled dlarray

unlabeled dlarray object

Unlabeled dlarray, returned as an unlabeled dlarray object that is the same as the input array \( \text{dlX} \), but without any labels. If \( \text{dlX} \) is unlabeled, then \( y = \text{dlX} \).
Tips

- Use `stripdims` to ensure that a `dlarray` behaves like a numeric array of the same size, without any special behavior due to dimension labels.
- `ndims(dlX)` can decrease after a `stripdims` call because the function removes trailing singleton labels.

```matlab
dlX = dlarray(ones(3,2), 'SCB');
ndims(dlX)
ans =
    3
dlX = stripdims(dlX);
ndims(dlX)
ans =
    2
```

See Also

dims | dlarray | finddim

Introduced in R2019b
**dlnetwork**

Deep learning network for custom training loops

**Description**

A `dlnetwork` object enables support for custom training loops using automatic differentiation.

**Tip** For most deep learning tasks, you can use a pretrained network and adapt it to your own data. For an example showing how to use transfer learning to retrain a convolutional neural network to classify a new set of images, see “Train Deep Learning Network to Classify New Images”. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Custom Training Loops”.

**Creation**

**Syntax**

```matlab
dlnet = dlnetwork(lgraph)
```

**Description**

`dlnet = dlnetwork(lgraph)` converts a layer graph to a `dlnetwork` object representing a deep neural network for custom training loops.

**Input Arguments**

- `lgraph` — Network architecture
  - `layerGraph` object
Network architecture, specified as a layer graph.

The layer graph must not contain output layers. When training the network, calculate the loss separately.

For a list of layers supported by dlnetwork, see “Supported Layers” on page 1-1561.

**Properties**

**Layers — Network layers**

Layer array

Network layers, specified as a Layer array.

**Connections — Layer connections**

Table

Layer connections, specified as a table with two columns.

Each table row represents a connection in the layer graph. The first column, Source, specifies the source of each connection. The second column, Destination, specifies the destination of each connection. The connection sources and destinations are either layer names or have the form 'layerName/IOName', where 'IOName' is the name of the layer input or output.

Data Types: table

**Learnables — Network learnable parameters**

Table

Network learnable parameters, specified as a table with three columns:

- **Layer** - Layer name, specified as a string scalar.
- **Parameter** - Parameter name, specified as a string scalar.
- **Value** - Value of parameter, specified as a dlarray.

The network learnable parameters contain the features learned by the network. For example, the weights of convolution and fully connected layers.

Data Types: table
**State — Network state**

Network state, specified as a table.

The network state is a table with three columns:

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a numeric array object.

The network state contains information remembered by the network between iterations. For example, the state of LSTM and batch normalization layers.

During training or inference, you can update the network state using the output of the `forward` and `predict` functions.

Data Types: `table`

**Object Functions**

- `forward` — Compute deep learning network output for training
- `predict` — Compute deep learning network output for inference
- `layerGraph` — Graph of network layers for deep learning

**Examples**

**Convert Pretrained Network to dlnetwork Object**

To implement a custom training loop for your network, first convert it to a `dlnetwork` object. Do not include output layers in a `dlnetwork` object. Instead, you must specify the loss function in the custom training loop.

Load a pretrained GoogLeNet model using the `googlenet` function. This function requires the Deep Learning Toolbox™ Model for GoogLeNet Network support package. If this support package is not installed, then the function provides a download link.

```matlab
net = googlenet;
```
Convert the network to a layer graph and remove the layers used for classification using `removeLayers`.

```matlab
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,["prob" "output"]);
```

Convert the network to a `dlnetwork` object.

```matlab
dlnet = dlnetwork(lgraph)
dlnet =
    dlnetwork with properties:
        Layers: [142x1 nnet.cnn.layer.Layer]
        Connections: [168x2 table]
        Learnables: [116x3 table]
        State: [0x3 table]
```

### Train Network Using Custom Training Loop

This example shows how to train a network that classifies handwritten digits with a custom learning rate schedule.

If `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation.

This example trains a network to classify handwritten digits with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by

\[ \rho_t = \frac{\rho_0}{1 + kt} \]

where \( t \) is the iteration number, \( \rho_0 \) is the initial learning rate, and \( k \) is the decay.

#### Load Training Data

Load the digits data.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```
Define Network

Define the network and specify the average image using the 'Mean' option in the image input layer.

layers = [
    imageInputLayer([28 28 1], 'Name', 'input', 'Mean', mean(XTrain,4))
    convolution2dLayer(5, 20, 'Name', 'conv1')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3, 20, 'Padding', 1, 'Name', 'conv2')
    reluLayer('Name', 'relu2')
    convolution2dLayer(3, 20, 'Padding', 1, 'Name', 'conv3')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(numClasses, 'Name', 'fc')];
lgraph = layerGraph(layers);

Create a dlnetwork object from the layer graph.

dlnet = dlnetwork(lgraph)

dlnet =
dlnetwork with properties:
    Layers: [8×1 nnet.cnn.layer.Layer]
    Connections: [7×2 table]
    Learnables: [8×3 table]
    State: [0×3 table]

Define Model Gradients Function

Create the function modelGradients, listed at the end of the example, that takes a dlnetwork object dlnet, a mini-batch of input data dlX with corresponding labels Y and returns the gradients of the loss with respect to the learnable parameters in dlnet and the corresponding loss.

Specify Training Options

Specify the training options.

velocity = []; numEpochs = 20;
miniBatchSize = 128;
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
initialLearnRate = 0.01;
momentum = 0.9;
decay = 0.01;

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

executionEnvironment = "auto";

Train Model

Train the model using a custom training loop.

For each epoch, shuffle the data and loop over mini-batches of data. At the end of each epoch, display the training progress.

For each mini-batch:

• Convert the labels to dummy variables.
• Convert the data to dlarray objects with underlying type single and specify the dimension labels 'SSCB' (spatial, spatial, channel, batch).
• For GPU training, convert to gpuArray objects.
• Evaluate the model gradients and loss using dlfeval and the modelGradients function.
• Determine the learning rate for the time-based decay learning rate schedule.
• Update the network parameters using the sgdupdate function.

Initialize the training progress plot.

plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")
end

Train the network.

iteration = 0;
start = tic;
% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,:,idx);
    YTrain = YTrain(idx);

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data and convert the labels to dummy variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,:,idx);

        Y = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y(c,YTrain(idx)==classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(X),'SSCB');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients and loss using dlfeval and the modelGradients function.
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [dlnet.Learnables, velocity] = sgdmupdate(dlnet.Learnables, gradients, velocity, learnRate, momentum);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
        end
    end
end
Test Model

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

[XTest, YTest] = digitTest4DArrayData;

Convert the data to a dlarray object with dimension format 'SSCB'. For GPU prediction, also convert the data to gpuArray.
dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end

To classify images using a dlnetwork object, use the predict function and find the classes with the highest scores.

dlYPred = predict(dlnet,dlXTest);
 [~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);

Evaluate the classification accuracy.

accuracy = mean(YPred==YTest)
accuracy = 0.9780

**Model Gradients Function**

The modelGradients function takes a dlnetwork object dlnet, a mini-batch of input data dlX with corresponding labels Y and returns the gradients of the loss with respect to the learnable parameters in dlnet and the corresponding loss. To compute the gradients automatically, use the dlgradient function.

```matlab
function [gradients,loss] = modelGradients(dlnet,dlX,Y)

dlYPred = forward(dlnet,dlX);
dlYPred = softmax(dlYPred);
loss = crossentropy(dlYPred,Y);
gradients = dlgradient(loss,dlnet.Learnables);
end
```

**More About**

**Supported Layers**

The dlnetwork function supports the layers listed below and custom layers without a custom backward function.
### Input Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imageInputLayer</td>
<td>An image input layer inputs 2-D images to a network and applies data normalization.</td>
</tr>
<tr>
<td>sequenceInputLayer</td>
<td>A sequence input layer inputs sequence data to a network.</td>
</tr>
</tbody>
</table>

### Convolution and Fully Connected Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convolution2dLayer</td>
<td>A 2-D convolutional layer applies sliding convolutional filters to the input.</td>
</tr>
<tr>
<td>groupedConvolution2dLayer</td>
<td>A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. If grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution.</td>
</tr>
<tr>
<td>transposedConv2dLayer</td>
<td>A transposed 2-D convolution layer upsamples feature maps.</td>
</tr>
<tr>
<td>fullyConnectedLayer</td>
<td>A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.</td>
</tr>
</tbody>
</table>

### Sequence Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequenceInputLayer</td>
<td>A sequence input layer inputs sequence data to a network.</td>
</tr>
<tr>
<td>lstmLayer</td>
<td>An LSTM layer learns long-term dependencies between time steps in time series and sequence data.</td>
</tr>
</tbody>
</table>
### Activation Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reluLayer</td>
<td>A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.</td>
</tr>
<tr>
<td>leakyReluLayer</td>
<td>A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.</td>
</tr>
<tr>
<td>clippedReluLayer</td>
<td>A clipped ReLU layer performs a threshold operation, where any input value less than zero is set to zero and any value above the clipping ceiling is set to that clipping ceiling.</td>
</tr>
<tr>
<td>eluLayer</td>
<td>An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.</td>
</tr>
<tr>
<td>tanhLayer</td>
<td>A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.</td>
</tr>
<tr>
<td>softmaxLayer</td>
<td>A softmax layer applies a softmax function to the input.</td>
</tr>
</tbody>
</table>

### Normalization, Dropout, and Cropping Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>batchNormalizationLayer</td>
<td>A batch normalization layer normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.</td>
</tr>
<tr>
<td>Layer</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>crossChannelNormalizationLayer</td>
<td>A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.</td>
</tr>
<tr>
<td>dropoutLayer</td>
<td>A dropout layer randomly sets input elements to zero with a given probability.</td>
</tr>
<tr>
<td>crop2dLayer</td>
<td>A 2-D crop layer applies 2-D cropping to the input.</td>
</tr>
</tbody>
</table>

**Pooling and Unpooling Layers**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>averagePooling2dLayer</td>
<td>An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.</td>
</tr>
<tr>
<td>globalAveragePooling2dLayer</td>
<td>A global average pooling layer performs down-sampling by computing the mean of the height and width dimensions of the input.</td>
</tr>
<tr>
<td>maxPooling2dLayer</td>
<td>A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.</td>
</tr>
<tr>
<td>maxUnpooling2dLayer</td>
<td>A max unpooling layer un pools the output of a max pooling layer.</td>
</tr>
</tbody>
</table>

**Combination Layers**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>additionLayer</td>
<td>An addition layer adds inputs from multiple neural network layers element-wise.</td>
</tr>
<tr>
<td>Layer</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>depthConcatenationLayer</td>
<td>A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).</td>
</tr>
<tr>
<td>concatenationLayer</td>
<td>A concatenation layer takes inputs and concatenates them along a specified dimension. The inputs must have the same size in all dimensions except the concatenation dimension.</td>
</tr>
</tbody>
</table>

**See Also**
dlarray | dlfeval | dlgradient | forward | layerGraph | predict

**Topics**
“Train Generative Adversarial Network (GAN)“
“Automatic Differentiation Background”
“Define Custom Training Loops, Loss Functions, and Networks”

**Introduced in R2019b**
forward

Compute deep learning network output for training

Syntax

dlY = forward(dlnet,dlX)
[dlY1,...,dlYN] = forward(dlnet,dlX,'Outputs',layerNames)
[dlY1,...,dlYN,state] = forward(____)

Description

Some deep learning layers behave differently during training and inference (prediction). For example, during training, dropout layers randomly set input elements to zero to help prevent overfitting, but during inference, dropout layers do not change the input.

To compute network outputs for training, use the forward function. To compute network outputs for inference, use the predict function.

dlY = forward(dlnet,dlX) computes the network output dlY during training given the input data dlX.

[dlY1,...,dlYN] = forward(dlnet,dlX,'Outputs',layerNames) returns the outputs dlY1, ..., dlYN for the specified layers.

[dlY1,...,dlYN,state] = forward(____) also returns the updated network state using any of the previous syntaxes.

Examples

Train Network Using Custom Training Loop

This example shows how to train a network that classifies handwritten digits with a custom learning rate schedule.
If `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation.

This example trains a network to classify handwritten digits with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by

\[
\rho_t = \frac{\rho_0}{1 + k t}
\]

where \( t \) is the iteration number, \( \rho_0 \) is the initial learning rate, and \( k \) is the decay.

**Load Training Data**

Load the digits data.

[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);

**Define Network**

Define the network and specify the average image using the 'Mean' option in the image input layer.

```matlab
layers = [
    imageInputLayer([28 28 1], 'Name', 'input', 'Mean', mean(XTrain,4))
    convolution2dLayer(5, 20, 'Name', 'conv1')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3, 20, 'Padding', 1, 'Name', 'conv2')
    reluLayer('Name', 'relu2')
    convolution2dLayer(3, 20, 'Padding', 1, 'Name', 'conv3')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(numClasses, 'Name', 'fc')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```matlab
dlnet = dlnetwork(lgraph)
```

dlnet =
    dlnetwork with properties:
        Layers: [8×1 nnet.cnn.layer.Layer]
    Connections: [7×2 table]
    Learnables: [8×3 table]
Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding labels `Y` and returns the gradients of the loss with respect to the learnable parameters in `dlnet` and the corresponding loss.

Specify Training Options

Specify the training options.

```matlab
velocity = []; 
numEpochs = 20; 
miniBatchSize = 128; 
numObservations = numel(YTrain); 
numIterationsPerEpoch = floor(numObservations./miniBatchSize); 
initialLearnRate = 0.01; 
momentum = 0.9; 
decay = 0.01; 
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```matlab
executionEnvironment = "auto"; 
```

Train Model

Train the model using a custom training loop.

For each epoch, shuffle the data and loop over mini-batches of data. At the end of each epoch, display the training progress.

For each mini-batch:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels 'SSCB' (spatial, spatial, channel, batch).
- For GPU training, convert to `gpuArray` objects.
- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
• Determine the learning rate for the time-based decay learning rate schedule.
• Update the network parameters using the `sgdupdate` function.

Initialize the training progress plot.

```matlab
plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")
end
```

Train the network.

```matlab
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,idx);
    YTrain = YTrain(idx);

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data and convert the labels to dummy variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,idx);
        Y = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y(c,YTrain(idx)==classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(X),"SSCB");

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" || canUseGPU) || executionEnvironment == "gpu"
            % ... (GPU code here) ...
        end
```

dlX = gpuArray(dlX);
end

% Evaluate the model gradients and loss using dlfeval and the
% modelGradients function.
[gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

% Determine learning rate for time-based decay learning rate schedule.
learnRate = initialLearnRate/(1 + decay*iteration);

% Update the network parameters using the SGDM optimizer.
[dlnet.Learnables, velocity] = sgdmupdate(dlnet.Learnables, gradients, velocity, learnRate, momentum);

% Display the training progress.
if plots == "training-progress"
    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
    title("Epoch: " + epoch + ", Elapsed: " + string(D))
    drawnow
end
end
**Test Model**

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

\[
[X\text{Test}, Y\text{Test}] = \text{digitTest4DArrayData};
\]

Convert the data to a \texttt{dlarray} object with dimension format 'SSCB'. For GPU prediction, also convert the data to \texttt{gpuArray}.

\[
dlX\text{Test} = \text{dlarray}(X\text{Test},'SSCB');
\]

\[
\text{if} \ (\text{executionEnvironment} == \"auto\" \&\& \text{canUseGPU}) \|\| \text{executionEnvironment} == \"gpu\"
\]

\[
dlX\text{Test} = \text{gpuArray}(dlX\text{Test});
\]

\[
\text{end}
\]
To classify images using a `dlnetwork` object, use the `predict` function and find the classes with the highest scores.

```matlab
dlYPred = predict(dlnet,dlXTest);
 [~,idx] = max(extractdata(dlYPred),[],1);
 YPred = classes(idx);
```

Evaluate the classification accuracy.

```matlab
accuracy = mean(YPred==YTest)
accuracy = 0.9780
```

**Model Gradients Function**

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding labels `Y` and returns the gradients of the loss with respect to the learnable parameters in `dlnet` and the corresponding loss. To compute the gradients automatically, use the `dlgradient` function.

```matlab
function [gradients,loss] = modelGradients(dlnet,dlX,Y)

dlYPred = forward(dlnet,dlX);
 dlYPred = softmax(dlYPred);

loss = crossentropy(dlYPred,Y);
 gradients = dlgradient(loss,dlnet.Learnables);
end
```

**Input Arguments**

- **dlnet** — Network for custom training loops
  *dlnetwork object*

  Network for custom training loops, specified as a `dlnetwork` object.

- **dlX** — Input data
  *formatted dlarray*

  Input data, specified as a formatted `dlarray`. For more information about `dlarray` formats, see the `fmt` input argument of `dlarray`.  
layerNames — Layers to extract outputs from

string array | cell array of character vectors

Layers to extract outputs from, specified as a string array or a cell array of character vectors containing the layer names.

- If layerNames(i) corresponds to a layer with a single output, then layerNames(i) is the name of the layer.
- If layerNames(i) corresponds to a layer with multiple outputs, then layerNames(i) is the layer name followed by the character "/" and the name of the layer output: 'layerName/outputName'.

Output Arguments

dlY — Output data

formatted dlarray

Output data, returned as a formatted dlarray. For more information about dlarray formats, see the fmt input argument of dlarray.

state — Updated network state

table

Updated network state, returned as a table.

The network state is a table with three columns:

- **Layer** - Layer name, specified as a string scalar.
- **Parameter** - Parameter name, specified as a string scalar.
- **Value** - Value of parameter, specified as a numeric array object.

The network state contains information remembered by the network between iterations. For example, the state of LSTM and batch normalization layers.

Update the state of a dlnetwork using the State property.

See Also
dlayarray | dlfeval | dlgradient | dlnetwork | predict
Topics
“Train Generative Adversarial Network (GAN)”
“Automatic Differentiation Background”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2019b
**predict**

Compute deep learning network output for inference

**Syntax**

```matlab
dlY = predict(dlnet,dlX)
[dlY1,...,dlYN] = predict(dlnet,dlX,'Outputs',layerNames)
[dlY1,...,dlYN,state] = predict(____)
```

**Description**

Some deep learning layers behave differently during training and inference (prediction). For example, during training, dropout layers randomly set input elements to zero to help prevent overfitting, but during inference, dropout layers do not change the input.

To compute network outputs for inference, use the `predict` function. To compute network outputs for training, use the `forward` function. For prediction with `SeriesNetwork` and `DAGNetwork` objects, see `predict`.

```matlab
dlY = predict(dlnet,dlX) computes the network output dlY during inference given the input data dlX.
```

```matlab
[dlY1,...,dlYN] = predict(dlnet,dlX,'Outputs',layerNames) returns the outputs dlY1, ..., dlYN for the specified layers.
```

```matlab
[dlY1,...,dlYN,state] = predict(____) also returns the updated network state using any of the previous syntaxes.
```

**Tip** For prediction with `SeriesNetwork` and `DAGNetwork` objects, see `predict`.

**Examples**
Make Predictions Using dlnetwork Object

This example shows how to make predictions using a dlnetwork object by splitting data into mini-batches.

For large data sets, or when predicting on hardware with limited memory, make predictions by splitting the data into mini-batches. When making predictions with SeriesNetwork or DAGNetwork objects, the predict function automatically splits the input data into mini-batches. For dlnetwork objects, you must split the data into mini-batches manually.

Load dlnetwork Object

Load a trained dlnetwork object and the corresponding classes.

```matlab
s = load("digitsCustom.mat");
dlnet = s.dlnet;
classes = s.classes;
```

Load Data for Prediction

Load the digits data for prediction.

```matlab
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ... 
    'IncludeSubfolders',true);
```

Make Predictions

Loop over the mini-batches of the test data and make predictions using a custom prediction loop. To read a mini-batch of data from the datastore, set the ReadSize property to the mini-batch size.

For each mini-batch:

- Convert the data to dlarray objects with underlying type single and specify the dimension labels 'SSCB' (spatial, spatial, channel, batch).
- For GPU prediction, convert to gpuArray objects.
- Make predictions using the predict function.
- Determine the class labels by finding the maximum scores.
Specify the prediction options. Specify a mini-batch size of 128 and make predictions on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```matlab
miniBatchSize = 128;
executionEnvironment = "auto";
```

Set the read size property of the image datastore to the mini-batch size.

```matlab
imds.ReadSize = miniBatchSize;
```

Make predictions by looping over the mini-batches of data.

```matlab
numObservations = numel(imds.Files);
YPred = strings(1,numObservations);
i = 1;

% Loop over mini-batches.
while hasdata(imds)
    % Read mini-batch of data.
    data = read(imds);
    X = cat(4,data{:});

    % Normalize the images.
    X = single(X)/255;

    % Convert mini-batch of data to dlarray.
    dlX = dlarray(X,'SSCB');

    % If training on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        dlX = gpuArray(dlX);
    end

    % Make predictions using the predict function.
    dlYPred = predict(dlnet,dlX);

    % Determine corresponding classes.
    [~,idxTop] = max(extractdata(dlYPred),[],1);
    idxMiniBatch = i:min((i+miniBatchSize-1),numObservations);
    YPred(idxMiniBatch) = classes(idxTop);
    i = i + miniBatchSize;
end
```
Visualize some of the predictions.

```matlab
idx = randperm(numObservations,9);
figure
for i = 1:9
    subplot(3,3,i)
    I = imread(imds.Files{idx(i)});
    label = YPred(idx(i));
    imshow(I)
    title("Label: " + label)
end
```
**Input Arguments**

**dlnet** — Network for custom training loops

dlnetwork object

Network for custom training loops, specified as a dlnetwork object.

**dlX** — Input data

formatted dlarray

Input data, specified as a formatted dlarray. For more information about dlarray formats, see the fmt input argument of dlarray.
layerNames — Layers to extract outputs from
string array | cell array of character vectors

Layers to extract outputs from, specified as a string array or a cell array of character vectors containing the layer names.

• If \texttt{layerNames(i)} corresponds to a layer with a single output, then \texttt{layerNames(i)} is the name of the layer.

• If \texttt{layerNames(i)} corresponds to a layer with multiple outputs, then \texttt{layerNames(i)} is the layer name followed by the character "/" and the name of the layer output: 'layerName/outputName'.

Output Arguments

dlY — Output data
formatted dlarray

Output data, returned as a formatted dlarray. For more information about dlarray formats, see the \texttt{fmt} input argument of dlarray.

state — Updated network state
table

Updated network state, returned as a table.

The network state is a table with three columns:

• Layer - Layer name, specified as a string scalar.

• Parameter - Parameter name, specified as a string scalar.

• Value - Value of parameter, specified as a numeric array object.

The network state contains information remembered by the network between iterations. For example, the state of LSTM and batch normalization layers.

Update the state of a dlnetwork using the \texttt{State} property.

See Also
dlarray | dlfeval | dlgradient | dlnetwork | forward
Topics
“Train Generative Adversarial Network (GAN)“
“Automatic Differentiation Background”
“Define Custom Training Loops, Loss Functions, and Networks”

Introduced in R2019b
avgpool
Pool data to average values over spatial dimensions

Syntax

dlY = avgpool(dlX,poolsize)
dlY = avgpool(dlX,poolsize,'DataFormat',FMT)
dlY = avgpool(___,Name,Value)

Description

The average pooling operation performs downsampling by dividing the input into pooling regions and computing the average value of each region.

**Note** This function applies the average pooling operation to dlarray data. If you want to apply average pooling within a layerGraph object or Layer array, use one of the following layers:

- averagePooling2dLayer
- averagePooling3dLayer
- globalAveragePooling2dLayer
- globalAveragePooling3dLayer

dlY = avgpool(dlX,poolsize) performs downsampling by dividing the input dlX into rectangular or cuboidal regions defined by poolsize and computing the average value of the data in each region. The input dlX is a formatted dlarray with dimension labels. Pooling acts on spatial dimensions labeled 'S'. The output dlY is a formatted dlarray with the same dimension labels as dlX.

dlY = avgpool(dlX,poolsize,'DataFormat',FMT) specifies the dimension labels FMT when dlX is not a formatted dlarray. The output dlY is an unformatted dlarray with the same dimension order as dlX.
dlY = avgpool(___ ,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, 'Stride',3 sets the stride of the pooling operation.

Examples

**Pool Data to Average Values**

Pool data to average values over two spatial dimensions.

Create the input data as a dlarray. The data contains a single observation of random values with a height and width of six and a single channel.

```matlab
height = 6;
width = 6;
channels = 1;
observations = 1;
X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB')
```

```
dlX = 6(S) × 6(S) × 1(C) × 1(B) dlarray
0.1781    0.8819    0.1564    0.4820    0.2518    0.7302
0.1280    0.6692    0.8555    0.1206    0.2904    0.3439
0.9991    0.1904    0.6448    0.5895    0.6171    0.5841
0.1711    0.3689    0.3763    0.2262    0.2653    0.1078
0.0326    0.4607    0.1909    0.3846    0.8244    0.9063
0.5612    0.9816    0.4283    0.5830    0.9827    0.8797
```

Pool the data to average values over pooling regions of size 2 using a stride of 2.

```matlab
dlY = avgpool(dlX,2,'Stride',2)
```

```
dlY = 3(S) × 3(S) × 1(C) × 1(B) dlarray
0.4643    0.4036    0.4041
```
Input Arguments

dlX — Input data
dlarray

Input data, specified as a dlarray with or without dimension labels. When dlX is not a formatted dlarray, you must specify the dimension label format using 'DataFormat', FMT.

Pooling acts on dimensions that you specify as spatial dimensions using the 'S' dimension label. dlX must have at least one 'S' dimension. You can specify up to three dimensions in dlX as 'S' dimensions. The avgpool operation divides the data along each 'S' dimension into regions defined by poolsize. Values within each pooling region are averaged.

Data Types: single | double

poolsize — Size of pooling regions
numeric scalar | numeric vector

Size of the pooling regions, specified as a numeric scalar or numeric vector. If you specify poolsize as a scalar, the pooling regions have the same size along all spatial dimensions. To use rectangular or cuboidal pooling regions that have different sizes along each spatial dimension, specify poolsize as a vector with the same length as the number of spatial dimensions in dlX.

Example: 3
Data Types: single | double

FMT — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
• 'C' — Channel
• 'B' — Batch (for example, samples and observations)
• 'T' — Time (for example, sequences)
• 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.
Example: 'DataFormat','SSCB'
Data Types: char | string

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Stride',2 specifies the stride of the pooling regions as 2.

**Stride — Step size for traversing input data**

1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of 'Stride' is 1. If 'Stride' is less than poolsize in any dimension, then the pooling regions overlap.
Example: 'Stride',3
Data Types: single | double

**Padding — Padding applied to edges of data**

0 (default) | 'same' | numeric scalar | numeric vector | numeric matrix
Padding applied to edges of data, specified as the comma-separated pair consisting of 'Padding' and one of the following:

- 'same' — Padding is set so that the output size is the same as the input size when the stride is 1. More generally, the output size of each spatial dimension is \( \text{ceil}(\text{inputSize}/\text{stride}) \), where \( \text{inputSize} \) is the size of the input along a spatial dimension.

- Numeric scalar — The same padding value is applied to both ends of all spatial dimensions.

- Numeric vector — A different padding value is applied along each spatial dimension. Use a vector of size \( d \), where \( d \) is the number of spatial dimensions of the input data. The \( i \)th element of the vector specifies the padding applied to the start and the end along the \( i \)th spatial dimension.

- Numeric matrix — A different padding value is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-\( d \), where \( d \) is the number of spatial dimensions of the input data. The element \( (1,d) \) specifies the padding applied to the start of spatial dimension \( d \). The element \( (2,d) \) specifies the padding applied to the end of spatial dimension \( d \). For example, in 2-D, the format is \([\text{top, left}; \text{bottom, right}]\).

The default value of 'Padding' is 0.

Example: 'Padding','same'

Data Types: single | double

**Output Arguments**

dlY — Pooled data
dlarray

Pooled data, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.
More About

Average Pooling
The avgpool function pools the input data to average values over the spatial dimensions. For more information, see the definition of “Average Pooling Layer” on page 1-846 on the averagePooling2dLayer reference page.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

• When the input argument dlX is a dlarray with underlying data of type gpuArray, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
dlarray | dlconv | dlfeval | dlgradient | maxpool

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2019b
**batchnorm**

Normalize each channel of input data

**Syntax**

```
[dlY,mu,sigma] = batchnorm(dlX,offset,scaleFactor)
dlY = batchnorm(dlX,offset,scaleFactor,mu,sigma)
[dlY,datasetMu,datasetSigma] = batchnorm(dlX,offset,scaleFactor,datasetMu,datasetSigma)
[___] = batchnorm(___,'DataFormat',FMT)
[___] = batchnorm(___,Name,Value)
```

**Description**

The batch normalization operation normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization between convolution and nonlinear operations such as relu.

**Note** This function applies the batch normalization operation to dlarray data. If you want to apply batch normalization within a layerGraph object or Layer array, use the following layer:

- batchNormalizationLayer

```
[dlY,mu,sigma] = batchnorm(dlX,offset,scaleFactor) normalizes each channel of the input mini-batch dlX using the mean and variance statistics computed from each channel and applies a scale factor and offset.

The normalized activation is calculated using the following formula:

\[
\hat{x}_i = \frac{x_i - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}
\]
where \( x_i \) is the input activation, \( \mu_c \) (\( \text{mu} \)) and \( \sigma^2_c \) (\( \text{sigma} \)) are the per-channel mean and variance, respectively, and \( \epsilon \) is a small constant. \( \mu \) and \( \sigma \) are calculated over all 'S' (spatial), 'B' (batch), 'T' (time), and 'U' (unspecified) dimensions in \( \text{dlX} \) for each channel.

The normalized activation is offset and scaled according to the following formula:

\[
y_i = \gamma \hat{x}_i + \beta.
\]

The offset \( \beta \) and scale factor \( \gamma \) are specified with the offset and scaleFactor arguments.

The input \( \text{dlX} \) is a formatted \( \text{dlarray} \) with dimension labels. The output \( \text{dlY} \) is a formatted \( \text{dlarray} \) with the same dimension labels as \( \text{dlX} \).

\[
\text{dlY} = \text{batchnorm}(\text{dlX}, \text{offset}, \text{scaleFactor}, \mu, \sigma)
\]

normalizes each channel of the input \( \text{dlX} \) using the specified \( \mu \) and \( \sigma \) statistics and applies a scale factor and offset.

\[
[\text{dlY}, \text{datasetMu}, \text{datasetSigma}] = \text{batchnorm}(\text{dlX}, \text{offset}, \text{scaleFactor}, \text{datasetMu}, \text{datasetSigma})
\]

normalizes each channel of the input mini-batch \( \text{dlX} \) using the mean and variance statistics computed from each channel and applies a scale factor and offset. The function also updates the data set statistics \( \text{datasetMu} \) and \( \text{datasetSigma} \) using the following formula:

\[
s_n = \phi s_x + (1 - \phi)s_{n-1}
\]

where \( s_n \) is the statistic computed over several mini-batches, \( s_x \) is the per-channel statistic of the current mini-batch, and \( \phi \) is the decay value for the statistic.

Use this syntax to iteratively update the mean and variance statistics over several mini-batches of data during training. Use the final value of the mean and variance computed over all training mini-batches to normalize data for prediction and classification.

\[
[\_\_] = \text{batchnorm}(\_\_, \text{'DataFormat'}, \text{FMT})
\]

also specifies the dimension labels \( \text{FMT} \) when \( \text{dlX} \) is not a formatted \( \text{dlarray} \) in addition to the input arguments in previous syntaxes. The output \( \text{dlY} \) is an unformatted \( \text{dlarray} \) with the same dimension order as \( \text{dlX} \).

\[
[\_\_] = \text{batchnorm}(\_\_, \text{Name}, \text{Value})
\]

specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, \( \text{'MeanDecay'} \), 3 sets the decay rate of the moving average computation.
Examples

Normalize Data and Obtain the Statistics

Use `batchnorm` to normalize each channel of a mini-batch and obtain the per-channel normalization statistics.

Create the input data as a single observation of random values with a height and width of four and three channels.

```matlab
height = 4;
width = 4;
channels = 3;
observations = 1;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Create the learnable parameters.

```matlab
offset = zeros(channels,1);
scaleFactor = ones(channels,1);
```

Compute the batch normalization and obtain the statistics of each channel of the batch.

```matlab
[dY,mu,sigma] = batchnorm(dlX,offset,scaleFactor);
mu
sigma
```

```matlab
mu = 3x1
    0.6095
    0.6063
    0.4619
sigma = 3x1
    0.1128
    0.0880
    0.0805
```

Update Mean and Variance over Multiple Batches of Data

Use `batchnorm` to normalize several batches of data and update the statistics of the whole dataset after each normalization.
Create three batches of data. The data consists of 10-by-10 arrays with 5 channels. Each batch contains 20 observations.

```
height = 10;
width = 10;
channels = 5;
observations = 20;

X_1 = rand(height,width,channels,observations);
dlX_1 = dlarray(X_1,'SSCB');

X_2 = 1.5*rand(height,width,channels,observations);
dlX_2 = dlarray(X_2,'SSCB');

X_3 = 2.5*rand(height,width,channels,observations);
dlX_3 = dlarray(X_3,'SSCB');
```

Create the learnable parameters.

```
offset = rand(channels,1);
scale = ones(channels,1);
```

Compute the batchnormalization of the first batch of data, \(dlX_1\). Obtain the values of the mean and variance of this batch as outputs.

```
[dY_1,mu,sigma] = batchnorm(dlX_1,offset,scale);
```

Compute the batch normalisation of the second batch of data, \(dlX_2\). Use \(mu\) and \(sigma\) is inputs to obtain the values of the combined mean and variance of the data in batches \(dlX_1\) and \(dlX_2\).

```
[dY_2,datasetMu,datasetSigma] = batchnorm(dlX_2,offset,scale,mu,sigma);
```

Compute the batch normalisation of the final batch of data, \(dlX_3\). Update the dataset statistics \(datasetMu\) and \(datasetSigma\) to obtain the values of the combined mean and variance of all data in batches \(dlX_1\), \(dlX_2\), and \(dlX_3\).

```
[dY_3,datasetMuFull,datasetSigmaFull] = batchnorm(dlX_3,offset,scale,datasetMu,datasetSigma);
```

Observe the change in the mean across each dataset

```
plot([mu';datasetMu';datasetMuFull'])
legend({'Channel 1','Channel 2','Channel 3','Channel 4','Channel 5'},'Location','southeast')
xticks([1 2 3])
xlabel('Number of Batches')
```
\texttt{xlim([0.9 3.1])}
\texttt{ylabel('Per-Channel Mean')}
\texttt{title('Dataset Mean')}

\section*{Input Arguments}

\textbf{dlX — Input data}

\texttt{dlarray} | numeric array

Input data, specified as a \texttt{dlarray} with or without dimension labels or a numeric array. When \texttt{dlX} is not a formatted \texttt{dlarray}, you must specify the dimension label format using
'DataFormat', FMT. If dlX is a numeric array, at least one of offset or scaleFactor must be a dlarray.

dlX must have a 'C' channel dimension.

Data Types: single | double

offset — Channel offset
dlarray vector | numeric vector

Channel offset β, specified as a dlarray vector with or without dimension labels or a numeric vector.

If offset is a formatted dlarray, it must contain a 'C' dimension of the same size as the 'C' dimension of the input data.

Data Types: single | double

scaleFactor — Channel scale factor
dlarray vector | numeric vector

Channel scale factor γ, specified as a dlarray vector with or without dimension labels or a numeric vector.

If scaleFactor is a formatted dlarray, it must contain a 'C' dimension of the same size as the 'C' dimension of the input data.

Data Types: single | double

mu — Mean statistic for normalization
numeric vector

Mean statistic for normalization, specified as a numeric vector of the same length as the 'C' dimension of the input data.

mu is calculated over all 'S' (spatial), 'B' (batch), 'T' (time), and 'U' (unspecified) dimensions in dlX for each channel.

Data Types: single | double

sigma — Variance statistic for normalization
numeric vector

Variance statistic for normalization, specified as a numeric vector of the same length as the 'C' dimension of the input data.
sigma is calculated over all 'S' (spatial), 'B' (batch), 'T' (time), and 'U' (unspecified) dimensions in dlX for each channel.

Data Types: single | double

**datasetMu — Mean statistic of several batches of data**

numeric vector

Mean statistic of several batches of data, specified as a numeric vector of the same length as the 'C' dimension of the input data. To iteratively update the dataset mean over several batches of input data, use the datasetMu output of a previous call to batchnorm as the datasetMu input argument.

Data Types: single | double

**datasetSigma — Variance statistic of several batches of data**

numeric vector

Variance statistic of several batches of data, specified as a numeric vector of the same length as the 'C' dimension of the input data. To iteratively update the dataset variance over several batches of input data, use the datasetSigma output of a previous call to batchnorm as the datasetSigma input argument.

Data Types: single | double

**FMT — Dimension order of unformatted data**

char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.
Example: 'DataFormat','SSCB'
Data Types: char | string

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: 'MeanDecay',0.3,'MeanVariance',0.5 sets the decay rate for the moving average computations of the mean and variance of several batches of data to 0.3 and 0.5, respectively.

**Epsilon — Variance offset**
numeric scalar

Variance offset for preventing divide-by-zero errors, specified as the comma-separated pair consisting of 'Epsilon' and a numeric scalar. The specified value must be greater than 1e-5. The default value is 1e-5.

Data Types: single | double

**MeanDecay — Mean decay value**
numeric scalar between 0 and 1

Decay value for the moving average computation of the datasetMu output, specified as the comma-separated pair consisting of 'MeanDecay' and a numeric scalar between 0 and 1. The default value is 0.1.

Data Types: single | double

**VarianceDecay — Variance decay value**
numeric scalar between 0 and 1

Decay value for the moving average computation of the datasetSigma output, specified as the comma-separated pair consisting of 'VarianceDecay' and a numeric scalar between 0 and 1. The default value is 0.1.

Data Types: single | double
Output Arguments

dlY — Normalized data
dlarray

Normalized data, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.

mu — Per-channel mean
numeric column vector

Per-channel mean of the input data, returned as a numeric column vector with length equal to the size of the 'C' dimension of the input data.

sigma — Per-channel variance
numeric column vector

Per-channel variance of the input data, returned as a numeric column vector with length equal to the size of the 'C' dimension of the input data.

datasetMu — Updated mean statistic of several batches of data
numeric vector

Updated mean statistic of several batches of data, returned as a numeric vector with length equal to the size of the 'C' dimension of the input data. datasetMu is returned with the same shape as the input datasetMu.

The datasetMu output is the moving average computation of the mean statistic for each channel over several batches of input data. datasetMu is computed from the channel mean of the input data and the input datasetMu using the following formula:

\[ \text{datasetMu} = \text{meanDecay} \times \text{currentMu} + (1 - \text{meanDecay}) \times \text{datasetMu}, \]

where currentMu is the channel mean computed from the input data and the value of meanDecay is specified using the 'MeanDecay' name-value pair argument.

datasetSigma — Updated variance statistic of several batches of data
numeric vector
Updated variance statistic of several batches of data, returned as a numeric vector with length equal to the size of the 'C' dimension of the input data. \texttt{datasetSigma} is returned with the same shape as the input \texttt{datasetSigma}.

The \texttt{datasetSigma} output is the moving average computation of the variance statistic for each channel over several batches of input data. \texttt{datasetSigma} is computed from the channel variance of the input data and the input \texttt{datasetSigma} using the following formula:

\[
\text{datasetSigma} = \text{varianceDecay} \times \text{currentSigma} + (1 - \text{varianceDecay}) \times \text{datasetSigma},
\]

where \texttt{currentSigma} is the channel variance computed from the input data and the value of \texttt{varianceDecay} is specified using the 'VarianceDecay' name-value pair.

**More About**

**Batch Normalization**

The \texttt{batchnorm} function normalizes each input channel of a mini-batch of data. For more information, see the definition of “Batch Normalization Layer” on page 1-1053 on the \texttt{batchNormalizationLayer} reference page.

**Extended Capabilities**

**GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a \texttt{gpuArray} or a \texttt{dlarray} with underlying data of type \texttt{gpuArray}, this function runs on the GPU:
  - \texttt{dlX}
• offset
• scaleFactor

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
dlarray | dlconv | dlfeval | dlgradient | fullyconnect | relu

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Train Network with Multiple Outputs”

Introduced in R2019b
crossentropy

Categorical cross-entropy loss

Syntax

dlY = crossentropy(dlX,targets)
dlY = crossentropy(dlX,targets,'DataFormat',FMT)

Description

The cross-entropy operation computes the categorical cross-entropy loss between network predictions and target values for multiclass classification problems.

The loss is calculated using the following formula

\[
\text{loss} = -\frac{1}{N} \sum_{i=1}^{M} T_i \log(X_i)
\]

where \(X_i\) is the network response, \(T_i\) is the target value, \(M\) is the total number of responses in \(X\) (across all observations), and \(N\) is the total number of observations in \(X\).

Note This function computes the cross-entropy loss between predictions and targets stored as dlarray data. If you want to calculate the cross-entropy loss within a layerGraph object or Layer array for use with trainNetwork, use the following layer:

- classificationLayer

\[
dlY = \text{crossentropy}(dlX,targets)\]

computes the categorical cross-entropy loss between the predictions \(dlX\) and the target values \(targets\) for multiclass classification problems. The input \(dlX\) is a formatted dlarray with dimension labels. The output \(dlY\) is an unformatted scalar dlarray with no dimension labels.

\[
dlY = \text{crossentropy}(dlX,targets,'DataFormat',FMT)\]

also specifies the dimension labels \(FMT\) when \(dlX\) is not a formatted dlarray.
Examples

Find Cross-Entropy Loss Between Predicted and Target Labels

The cross-entropy loss evaluates how well the network predictions correspond to the target classification.

Create the input classification data as a matrix of random variables. The data can be any of 10 categories, and there are 12 observations.

```matlab
numCategories = 10;
observations = 12;
X = rand(numCategories,observations);
dlX = dlarray(X,'CB');
```

Convert the category values in the data to probability scores for each category.

```matlab
dlX = softmax(dlX); 
```

Create the target data that holds the correct category for each observation in `dlX`.

```matlab
targetsIdx = randi(10,1,12);
targets = zeros(10,12);
for i = 1:numel(targetsIdx)
    targets(targetsIdx(i),i) = 1;
end
```

Compute the cross-entropy loss between the predictions and the targets

```matlab
dlY = crossentropy(dlX,targets)
```

```matlab
dlY =
    1x1 dlarray

2.3343
```
Input Arguments

dlX — Predictions
dlarray | numeric array

Predictions, specified as a dlarray with or without dimension labels or a numeric array. When dlX is not a formatted dlarray, you must specify the dimension label format using 'DataFormat', FMT. If dlX is a numeric array, targets must be a dlarray.

Data Types: single | double

targets — Target classification labels
dlarray | numeric array

Target classification labels, specified as a dlarray with or without dimension labels or a numeric array. If targets is an unformatted dlarray or a numeric array, it must have the same dimension order as the input dlX. The size of each dimension of targets must match the size of each corresponding dimension in dlX.

Data Types: single | double

FMT — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.

Example: 'DataFormat','SSCB'

Data Types: char | string
Output Arguments

dlY — Cross-entropy loss
dlarray scalar

Cross-entropy loss, returned as a dlarray scalar without dimension labels. The output dlY has the same underlying data type as the input dlX.

The cross-entropy loss dlY is the average logarithmic loss across the 'B' batch dimension of dlX.

More About

Cross-Entropy Loss

The crossentropy function computes the cross-entropy loss for classification problems. For more information, see the definition of “Classification Output Layer” on page 1-864 on the ClassificationOutputLayer reference page.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

• When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU:
  • dlX
  • targets

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).
See Also
dllarray | dlfeval | dlgradient | mse | softmax

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Custom Training Loop”
“Train Network Using Model Function”
“Train Network with Multiple Outputs”

Introduced in R2019b
**dlconv**

Deep learning convolution

**Syntax**

\[
dlY = \text{dlconv}(dlX, \text{weights}, \text{bias}) \\
dlY = \text{dlconv}(dlX, \text{weights}, \text{bias}, '\text{DataFormat}', \text{FMT}) \\
dlY = \text{dlconv}(\text{___Name}, \text{Value})
\]

**Description**

The convolution operation applies sliding filters to the input data. Use 1-D and 2-D filters with ungrouped or grouped convolutions and 3-D filters with ungrouped convolutions.

Use grouped convolution for channel-wise separable (also known as depth-wise separable) convolution. For each group, the operation convolves the input by moving filters along spatial dimensions of the input data, computing the dot product of the weights and the data and adding a bias. If the number of groups is equal to the number of channels, then this function performs channel-wise convolution. If the number of groups is equal to 1, this function performs ungrouped convolution.

**Note** This function applies the deep learning convolution operation to `dlarray` data. If you want to apply convolution within a `layerGraph` object or `Layer` array, use one of the following layers:

- `convolution2dLayer`
- `groupedConvolution2dLayer`
- `convolution3dLayer`

\[
dlY = \text{dlconv}(dlX, \text{weights}, \text{bias})
\]

computes the deep learning convolution of the input `dlX` using sliding convolutional filters defined by `weights`, and adds a constant bias. The input `dlX` is a formatted `dlarray` with dimension labels. Convolution acts on
dimensions that you specify as 'S' dimensions. The output dlY is a formatted dlarray with the same dimension labels as dlX.

dlY = dlconv(dlX,weights,bias,'DataFormat',FMT) also specifies dimension label format FMT when dlX is not a formatted dlarray. The output dlY is an unformatted dlarray with the same dimension order as dlX.

dlY = dlconv(___ Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, 'Stride',3 sets the stride of the convolution operation.

Examples

Perform Ungrouped Convolution

Convolve all channels of an image input using a single filter.

Import the image data and convert it to a dlarray.

\[
X = \text{imread('sherlock.jpg')};
\]
\[
dlX = \text{dlarray} \left( \text{single}(X),'SSC' \right);
\]

Display the image.

\[
\text{imshow}(X,'\text{DisplayRange}',[])\]

Initialize the convolutional filters. Specify an ungrouped convolution that applies a single filter to all three channels of the input data.

```matlab
filterHeight = 10;
filterWidth = 10;
umChannelsPerGroup = 3;
numFiltersPerGroup = 1;
umGroups = 1;

weights = rand(filterHeight,filterWidth,numChannelsPerGroup,numFiltersPerGroup,numGroups);

Initialize the bias term.

bias = rand(numFiltersPerGroup*numGroups,1);
```
Perform the convolution. Use a 'Stride' value of 2 and a 'DilationFactor' value of 2.

dlY = dlconv(dlX,weights,bias,'Stride',2,'DilationFactor',2);

Display the convolved image.

Y = extractdata(dlY);
imshow(Y,'DisplayRange',[])

Perform Grouped Convolution

Convolve the input data in three groups of two channels each. Apply four filters per group.
Create the input data as ten observations of size 100-by-100 with six channels.

\[
\begin{align*}
\text{height} &= 100; \\
\text{width} &= 100; \\
\text{channels} &= 6; \\
\text{numObservations} &= 10;
\end{align*}
\]

\[
X = \text{rand}(\text{height}, \text{width}, \text{channels}, \text{numObservations});
\]

\[
dlX = \text{dlarray}(X,'SSCB');
\]

Initialize the convolutional filters. Specify three groups of convolutions that each apply four convolution filters to two channels of the input data.

\[
\begin{align*}
\text{filterHeight} &= 8; \\
\text{filterWidth} &= 8; \\
\text{numChannelsPerGroup} &= 2; \\
\text{numFiltersPerGroup} &= 4; \\
\text{numGroups} &= 3;
\end{align*}
\]

\[
\text{weights} = \text{rand}(\text{filterHeight}, \text{filterWidth}, \text{numChannelsPerGroup}, \text{numFiltersPerGroup}, \text{numGroups});
\]

Initialize the bias term.

\[
\text{bias} = \text{rand}(\text{numFiltersPerGroup}\times\text{numGroups},1);
\]

Perform the convolution.

\[
dlY = \text{dlconv}(
dlX, \text{weights}, \text{bias});
\]

\[
\text{size}(
dlY) \\
\text{dims}(
dlY)
\]

\[
\begin{align*}
\text{ans} &= 1\times4 \\
&= 93 \quad 93 \quad 12 \quad 10 \\
\text{ans} &= 'SSCB'
\end{align*}
\]

The 12 channels of the convolution output represent the three groups of convolutions with four filters per group.

**Perform Channel-Wise Separable Convolution**

Separate the input data into channels and perform convolution on each channel separately.
Create the input data as a single observation with a size of 64-by-64 and ten channels. Create the data as an unformatted `dlarray`.

```matlab
height = 64;
width = 64;
channels = 10;

X = rand(height,width,channels);
dlX = dlarray(X);
```

Initialize the convolutional filters. Specify an ungrouped convolution that applies a single convolution to all three channels of the input data.

```matlab
filterHeight = 8;
filterWidth = 8;
numChannelsPerGroup = 1;
numFiltersPerGroup = 1;
numGroups = channels;

weights = rand(filterHeight,filterWidth,numChannelsPerGroup,numFiltersPerGroup,numGroups);
```

Initialize the bias term.

```matlab
bias = rand(numFiltersPerGroup*numGroups,1);
```

Perform the convolution. Specify the dimension labels of the input data using the `'DataFormat'` option.

```matlab
dlY = dlconv(dlX,weights,bias,'DataFormat','SSC');
size(dlY)
```

```matlab
ans = 1x3
     57    57    10
```

Each channel is convolved separately, so there are ten channels in the output.

### Input Arguments

- **dlX — Input data**
  - `dlarray` | numeric array

Input data, specified as a `dlarray` with or without dimension labels or a numeric array. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using
'DataFormat', FMT. If dlX is a numeric array, at least one of weights or bias must be a dlarray.

Convolution acts on dimensions that you specify as spatial dimensions using the 'S' dimension label. You can specify up to three dimensions in dlX as 'S' dimensions.

Data Types: single | double

weights — Convolutional filters
dlarray | numeric array

Convolutional filters, specified as a dlarray with or without labels or a numeric array. The weights argument specifies the size and values of the filters, as well as the number of filters and the number of groups for grouped convolutions.


- filterSize — Size of the convolutional filters. filterSize can have up to three dimensions, depending on the number of spatial dimensions in the input data.

<table>
<thead>
<tr>
<th>Input Data 'S' Dimensions</th>
<th>filterSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-D</td>
<td>h, where h corresponds to the height of the filter</td>
</tr>
<tr>
<td>2-D</td>
<td>h-by-w, where h and w correspond to the height and width of the filter, respectively</td>
</tr>
<tr>
<td>3-D</td>
<td>h-by-w-by-d, where h, w, and d correspond to the height, width, and depth of the filter, respectively</td>
</tr>
</tbody>
</table>

- numChannelsPerGroup — Number of channels to convolve within each group. numChannelsPerGroup must equal the number of channels in the input data divided by numGroups, the number of groups. For ungrouped convolutions, where numGroups = 1, numChannelsPerGroup must equal the number of channels in the input data.

- numFiltersPerGroup — Number of filters to apply within each group.

- numGroups — Number of groups (optional). When numGroups > 1, the function performs grouped convolutions. Grouped convolutions are not supported for input data with more than two 'S' dimensions. When numGroups = 1, the function performs ungrouped convolutions; in this case, this dimension is singleton and can be omitted.
If `weights` is a formatted `dlarray`, it can have multiple spatial dimensions labeled 'S', one channel dimension labeled 'C', and up to two other dimensions labeled 'U'. The number of 'S' dimensions must match the number of 'S' dimensions of the input data. The labeled dimensions correspond to the filter specifications as follows.

<table>
<thead>
<tr>
<th>Filter Specification</th>
<th>Dimension Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filterSize</code></td>
<td>Up to three 'S' dimensions</td>
</tr>
<tr>
<td><code>numChannelsPerGroup</code></td>
<td>'C' dimension</td>
</tr>
<tr>
<td><code>numFiltersPerGroup</code></td>
<td>First 'U' dimension</td>
</tr>
<tr>
<td><code>numGroups</code> (optional)</td>
<td>Second 'U' dimension</td>
</tr>
</tbody>
</table>

Data Types: `single` | `double`

**bias — Bias constant**

`dlarray vector` | `dlarray scalar` | `numeric vector` | `numeric scalar` | `0`

Bias constant, specified as a `dlarray` vector or `dlarray` scalar with or without labels, a numeric vector, or a numeric scalar.

- If `bias` is a scalar or has only singleton dimensions, the same bias is applied to each output.
- If `bias` has a nonsingleton dimension, each element of `bias` is the bias applied to the corresponding convolutional filter specified by `weights`. The number of elements of `bias` must match the number of filters specified by `weights`.
- If `bias` is a scalar numeric array with value `0`, the bias term is disabled and no bias is added during the convolution operation.

If `bias` is a formatted `dlarray`, the nonsingleton dimension must be a channel dimension labeled 'C'.

Data Types: `single` | `double`

**FMT — Dimension order of unformatted data**

`char array` | `string`

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:
• 'S' — Spatial
• 'C' — Channel
• 'B' — Batch (for example, samples and observations)
• 'T' — Time (for example, sequences)
• 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.
Example: 'DataFormat','SSCB'
Data Types: char | string

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.
Example: 'DilationFactor',2 sets the dilation factor for each convolutional filter to 2.

**Stride — Step size for traversing input data**
1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of 'Stride' is 1.
Example: 'Stride',3
Data Types: single | double

**DilationFactor — Filter dilation factor**
1 (default) | numeric scalar | numeric vector
Filter dilation factor, specified as the comma-separated pair consisting of 'DilationFactor' and one of the following.

- Numeric scalar — The same dilation factor value is applied for all spatial dimensions.
- Numeric vector — A different dilation factor value is applied along each spatial dimension. Use a vector of size d, where d is the number of spatial dimensions of the input data. The ith element of the vector specifies the dilation factor applied to the ith spatial dimension.

Use the dilation factor to increase the receptive field of the filter (the area of the input that the filter can see) on the input data. Using a dilation factor corresponds to an effective filter size of \(\text{filterSize} + (\text{filterSize}-1) \times (\text{dilationFactor}-1)\).

Example: 'DilationFactor', 2

Data Types: single | double

Padding — Padding applied to edges of data
0 (default) | 'same' | numeric scalar | numeric vector | numeric matrix

Padding applied to edges of data, specified as the comma-separated pair consisting of 'Padding' and one of the following:

- 'same' — Padding is set so that the output size is the same as the input size when the stride is 1. More generally, the output size of each spatial dimension is \(\text{ceil}(\text{inputSize}/\text{stride})\), where \(\text{inputSize}\) is the size of the input along a spatial dimension.
- Numeric scalar — The same padding value is applied to both ends of all spatial dimensions.
- Numeric vector — A different padding value is applied along each spatial dimension. Use a vector of size d, where d is the number of spatial dimensions of the input data. The ith element of the vector specifies the padding applied to the start and the end along the ith spatial dimension.
- Numeric matrix — A different padding value is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-d, where d is the number of spatial dimensions of the input data. The element (1,d) specifies the padding applied to the start of spatial dimension d. The element (2,d) specifies the padding applied to the end of spatial dimension d. For example, in 2-D, the format is [top, left; bottom, right].

The default value of 'Padding' is 0.
Example: 'Padding','same'
Data Types: single | double

**Output Arguments**

**dlY — Convolved feature map**

dlarray

Convolved feature map, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.

The size of the 'C' channel dimension of dlY depends on the size of the weights input. The size of the 'C' dimension of output Y is the product of the size of the dimensions numFiltersPerGroup and numGroups in the weights argument. If weights is a formatted dlarray, this product is the same as the product of the size of the 'U' dimensions.

**More About**

**Deep Learning Convolution**

The dlconv function applies sliding convolution filters to the spatial dimensions of the input data. The dlconv function supports convolution in one, two, or three spatial dimensions. For more information, see the definition of convolutional layer on page 1-879 on the convolution2dLayer reference page.
Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

• When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.
  • dlX
  • weights
  • bias

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
batchnorm | dlarray | dlfeval | dlgradient | fullyconnect | maxpool | relu

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2019b
dltranspconv
Deep learning transposed convolution

Syntax

dlY = dltranspconv(dlX,weights,bias)
dlY = dltranspconv(dlX,weights,bias,'DataFormat',FMT)
dlY = dltranspconv(___ Name,Value)

Description
The transposed convolution operation upsamples feature maps.

**Note** This function applies the deep learning transposed convolution operation to dlarray data. If you want to apply transposed convolution within a layerGraph object or Layer array, use one of the following layers:

- transposedConv2dLayer
- transposedConv3dLayer

\[ dlY = dltranspconv(dlX,weights,bias) \]
computes the deep learning transposed convolution of the input \( dlX \) using the filters defined by \( weights \), and adds a constant \( bias \). The input \( dlX \) is a formatted dlarray with dimension labels. Transposed convolution acts on dimensions that you specify as 'S' and 'C' dimensions. The output \( dlY \) is a formatted dlarray with the same dimension labels as \( dlX \).

\[ dlY = dltranspconv(dlX,weights,bias,'DataFormat',FMT) \]
also specifies the dimension format \( FMT \) when \( dlX \) is not a formatted dlarray. The output \( dlY \) is an unformatted dlarray with the same dimension order as \( dlX \).

\[ dlY = dltranspconv(____ Name,Value) \]
specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, 'Stride',3 sets the stride of the convolution operation.
Examples

Upsample Image Using Transposed Convolution

Convolve an image and then use transposed convolution to resize the convolved image to the same size as the original image.

Import the image data and convert it to a dlarray.

```matlab
X = imread('sherlock.jpg');
dlX = dlarray(single(X),'SSC');
```

Display the image.

```matlab
imshow(X)
```
Initialize the convolutional filters and bias term. Specify an ungrouped convolution that applies a single filter to all three channels of the input data.

```matlab
cfilterHeight = 10;
cfilterWidth = 10;
cnumChannelsPerGroup = 3;
cnumFiltersPerGroup = 1;
cnumGroups = 1;

weights = rand(filterHeight,filterWidth,numChannelsPerGroup,numFiltersPerGroup,numGroups);
bias = rand(numFiltersPerGroup*numGroups,1);

Perform the convolution. Use a 'Stride' value of 2 and a 'DilationFactor' value of 3.
dlY = dlconv(dlX,weights,bias,'Stride',2,'DilationFactor',3);

Display the convolved image.
Y = extractdata(dlY);
imshow(rescale(Y))
```
Initialize the transposed convolutional filters and bias. Specify an ungrouped transposed convolution that applies three filters to the input. Use the same filter height and filter width as for the convolution operation.

numChannelsPerGroupTC = 1;
numFiltersPerGroupTC = 3;

weightsTC = rand(filterHeight,filterWidth,numFiltersPerGroupTC,numChannelsPerGroupTC,numGroups);
biasTC = rand(numFiltersPerGroupTC*numGroups,1);

Perform the transposed convolution. Use the same stride and dilation factor as for the convolution operation.

dlZ = dltranspconv(dlY,weightsTC,biasTC,'Stride',2,'DilationFactor',3);

Display the image after the transposed convolution.
\[
Z = \text{extractdata(dlZ)}; \\
\text{imshow(rescale(Z))}
\]

Compare the size of the original image, the convolved image, and the image after the transposed convolution.

\[
\begin{align*}
\text{sizeX} &= \text{size}(X) \\
\text{sizeY} &= \text{size}(Y) \\
\text{sizeZ} &= \text{size}(Z)
\end{align*}
\]

\[
\begin{align*}
\text{sizeX} &= 1\times3 \\
&= 640 \quad 960 \quad 3 \\
\text{sizeY} &= 1\times2 \\
&= 311 \quad 471
\end{align*}
\]
sizeZ = 1x3
   640   960    3

The transposed convolution upsamples the convolved data to the size of the original input data.

**Perform Grouped Transposed Convolution**

Apply transposed convolution to the input data in three groups of two channels each. Apply four filters per group.

Create the input data as ten observations of size 100-by-100 with six channels.

```plaintext
height = 100;
width = 100;
channels = 6;
numObservations = 10;

X = rand(height,width,channels,numObservations);
dlX = dlarray(X,'SSCB');

```

Initialize the filters for the transposed convolution operation. Specify three groups of transposed convolutions that each apply four filters to two channels of the input data.

```plaintext
filterHeight = 8;
filterWidth = 8;
umChannelsPerGroup = 2;
numFiltersPerGroup = 4;
numGroups = 3;

weights = rand(filterHeight,filterWidth,numFiltersPerGroup,numChannelsPerGroup,numGroups);

```

Initialize the bias term.

```plaintext
bias = rand(numFiltersPerGroup*numGroups,1);
```

Perform the transposed convolution.

```plaintext
dlY = dltranspconv(dlX,weights,bias);
size(dlY)
dims(dlY)
ans = 1x4
   107   107    12    10
```
ans = 'SSCB'

The 12 channels of the convolution output represent the three groups of transposed convolutions with four filters per group.

**Input Arguments**

**dlX — Input data**

dlarray | numeric array

Input data, specified as a dlarray with or without dimension labels or a numeric array. When dlX is not a formatted dlarray, you must specify the dimension label format using 'DataFormat', FMT. If dlX is a numeric array, at least one of weights or bias must be a dlarray.

Convolution acts on dimensions that you specify as spatial dimensions using the 'S' dimension label. You can specify up to three dimensions in dlX as 'S' dimensions.

Data Types: single | double

**weights — Filters**

dlarray | numeric array

Filters, specified as a dlarray with or without labels or a numeric array. The weights argument specifies the size and values of the filters, as well as the number of filters and the number of groups for grouped transposed convolutions.


- **filterSize** — Size of the convolutional filters. filterSize can have up to three dimensions, depending on the number of spatial dimensions in the input data.

<table>
<thead>
<tr>
<th>Input Data 'S' Dimensions</th>
<th>filterSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-D</td>
<td>$h$, where $h$ corresponds to the height of the filter</td>
</tr>
</tbody>
</table>
Input Data 'S' Dimensions | filterSize
---|---
2-D | \( h \)-by-\( w \), where \( h \) and \( w \) correspond to the height and width of the filter, respectively
3-D | \( h \)-by-\( w \)-by-\( d \), where \( h \), \( w \), and \( d \) correspond to the height, width, and depth of the filter, respectively

- **numFiltersPerGroup** — Number of filters to apply within each group.
- **numChannelsPerGroup** — Number of channels within each group for grouped transposed convolutions. `numChannelsPerGroup` must equal the number of channels in the input data divided by `numGroups`, the number of groups. For ungrouped convolutions, where `numGroups = 1`, `numChannelsPerGroup` must equal the number of channels in the input data.
- **numGroups** — Number of groups (optional). When `numGroups > 1`, the function performs grouped transposed convolutions. When `numGroups = 1`, the function performs ungrouped transposed convolutions; in this case, this dimension is singleton and can be omitted.

If `weights` is a formatted `dlarray`, it can have multiple spatial dimensions labeled 'S', one channel dimension labeled 'C', and up to two other dimensions labeled 'U'. The number of 'S' dimensions must match the number of 'S' dimensions of the input data. The labeled dimensions correspond to the filter specifications as follows.

<table>
<thead>
<tr>
<th>Filter Specification</th>
<th>Dimension Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filterSize</code></td>
<td>Up to three 'S' dimensions</td>
</tr>
<tr>
<td><code>numFiltersPerGroup</code></td>
<td>'C' dimension</td>
</tr>
<tr>
<td><code>numChannelsPerGroup</code></td>
<td>First 'U' dimension</td>
</tr>
<tr>
<td><code>numGroups</code> (optional)</td>
<td>Second 'U' dimension</td>
</tr>
</tbody>
</table>

Data Types: `single` | `double`

**bias** — Bias constant

dlarray vector | dlarray scalar | numeric vector | numeric scalar

Bias constant, specified as a `dlarray` vector or `dlarray` scalar with or without labels, a numeric vector, or a numeric scalar.
• If \texttt{bias} is a scalar or has only singleton dimensions, the same bias is applied to each entry of the output.

• If \texttt{bias} has a nonsingleton dimension, each element of \texttt{bias} is the bias applied to the corresponding convolutional filter specified by \texttt{weights}. The number of elements of \texttt{bias} must match the number of filters specified by \texttt{weights}.

If \texttt{bias} is a formatted \texttt{dlarray}, the nonsingleton dimension must be a channel dimension labeled 'C'.

Data Types: single | double

\textbf{FMT — Dimension order of unformatted data}

\texttt{char array | string}

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in \texttt{FMT} must be one of the following:

• 'S' — Spatial
• 'C' — Channel
• 'B' — Batch (for example, samples and observations)
• 'T' — Time (for example, sequences)
• 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data \texttt{dlX} is an unformatted \texttt{dlarray}.

Example: 'DataFormat','SSCB'

Data Types: char | string

\textbf{Name-Value Pair Arguments}

Specify optional comma-separated pairs of \texttt{Name}, \texttt{Value} arguments. \texttt{Name} is the argument name and \texttt{Value} is the corresponding value. \texttt{Name} must appear inside quotes. You can specify several name and value pair arguments in any order as \texttt{Name1},\texttt{Value1},...,\texttt{NameN},\texttt{ValueN}.

Example: 'Stride',\texttt{2} sets the stride of each filter to 2.
**Stride — Step size for traversing input data**
1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of 'Stride' is 1.

Example: 'Stride',3

Data Types: single | double

**DilationFactor — Filter dilation factor**
1 (default) | numeric scalar | numeric vector

Filter dilation factor, specified as the comma-separated pair consisting of 'DilationFactor' and one of the following.

- Numeric scalar — The same dilation factor value is applied for all spatial dimensions.
- Numeric vector — A different dilation factor value is applied along each spatial dimension. Use a vector of size d, where d is the number of spatial dimensions of the input data. The ith element of the vector specifies the dilation factor applied to the ith spatial dimension.

Use the dilation factor to increase the receptive field of the filter (the area of the input that the filter can see) on the input data. Using a dilation factor corresponds to an effective filter size of filterSize + (filterSize-1)*(dilationFactor-1).

Example: 'DilationFactor',2

Data Types: single | double

**Cropping — Cropping applied to edges of data**
0 (default) | 'same' | numeric scalar | numeric vector | numeric matrix

Cropping applied to edges of data, specified as the comma-separated pair consisting of 'Cropping' and one of the following.

- 'same' — Cropping is set so that the output size is the same as the input size when the stride is 1. More generally, the output size of each spatial dimension is
inputSize*stride, where inputSize is the size of the input along a spatial dimension.

- Numeric scalar — The same cropping value is applied to both ends of all spatial dimensions.
- Numeric vector — A different cropping value is applied along each spatial dimension. Use a vector of size d, where d is the number of spatial dimensions of the input data. The ith element of the vector specifies the cropping applied to the start and the end along the ith spatial dimension.
- Numeric matrix — A different cropping value is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-d, where d is the number of spatial dimensions of the input data. The element (1,d) specifies the cropping applied to the start of spatial dimension d. The element (2,d) specifies the cropping applied to the end of spatial dimension d. For example, in 2-D the format is [top, left; bottom, right].

Example: 'Cropping','same'

Data Types: single | double

**Output Arguments**

**dlY — Feature map**

dlarray

Feature map, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray or numeric array with the same dimension order as the input data.

The size of the 'C' channel dimension of dlY depends on the size of the weights input. The size of the 'C' dimension of output Y is the product of the size of the dimensions numFiltersPerGroup and numGroups in the weights argument. If weights is a formatted dlarray, this product is the same as the product of the size of the 'C' dimension and the second 'U' dimension.
Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

• When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.
  • dlX
  • weights
  • bias

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
avgpool | dlarray | dlconv | dlfeval | dlgradient | maxpool | maxunpool

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2019b
fullyconnect

Sum all weighted input data and apply a bias

Syntax

dlY = fullyconnect(dlX,weights,bias)
dlY = fullyconnect(dlX,weights,bias,'DataFormat',FMT)

Description

The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.

**Note** This function applies the fully connect operation to dlarray data. If you want to apply average pooling within a layerGraph object or Layer array, use the following layer:

- fullyConnectedLayer

$$dlY = fullyconnect(dlX,weights,bias)$$ computes the weighted sum of the spatial, channel, and unspecified data in dlX using the weights specified by weights, and adds a bias. The input dlX is a formatted dlarray with dimension labels. The output dlY is a formatted dlarray.

$$dlY = fullyconnect(dlX,weights,bias,'DataFormat',FMT)$$ also specifies the dimension labels FMT when dlX is not a formatted dlarray. The output dlY is an unformatted dlarray.

Examples
Fully Connect All Input Data to Output Features

The `fullyconnect` function uses the weighted sum to connect all inputs of an observation to each output feature.

Create the input data as a single observation of random values with a height and width of 12 and 32 channels.

```matlab
height = 12;
width = 12;
channels = 32;
observations = 1;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Create the learnable parameters. For this operation there are ten output features.

```matlab
outputFeatures = 10;
weights = ones(outputFeatures,height,width,channels);
bias = ones(outputFeatures,1);
```

Apply the `fullyconnect` operation.

```matlab
dlY = fullyconnect(dlX,weights,bias);
```

```
dlY =
10(C) × 1(B) dlarray

1.0e+03 *
  2.3266
  2.3266
  2.3266
  2.3266
  2.3266
  2.3266
  2.3266
  2.3266
  2.3266
  2.3266
```
The output dlY is a 2-D dlarray with one channel dimension of size ten and one singleton batch dimension.

**Input Arguments**

**dlX — Input data**
dlarray | numeric array

Input data, specified as a dlarray with or without dimension labels or a numeric array. When dlX is not a formatted dlarray, you must specify the dimension label format using 'DataFormat', FMT. If dlX is a numeric array, at least one of weights or bias must be a dlarray.

The fullyconnect operation sums over the 'S', 'C', and 'U' dimensions of dlX for each output feature specified by weights. The size of each 'B' or 'T' dimension of dlX is preserved.

Data Types: single | double

**weights — Weights**
dlarray | numeric array

Weights, specified as a dlarray with or without labels or a numeric array.

If weights is an unformatted dlarray or a numeric array, the first dimension of weights must match the number of output features. If weights is a formatted dlarray, the size of the 'C' dimension must match the number of output features. weights must contain the same number of elements as the combined size of the 'S', 'C', and 'U' dimensions of input dlX multiplied by the number of output features.

Data Types: single | double

**bias — Bias constant**
dlarray vector | numeric vector

Bias constant, specified as a dlarray vector with or without labels or a numeric vector.

Each element of bias is the bias applied to the corresponding feature output. The number of elements of bias must match the number of output features specified by the first dimension of weights.
If bias is a formatted dlarray, the nonsingleton dimension must be a channel dimension labeled 'C'.

Data Types: single | double

FMT — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.

Example: 'DataFormat','SSCB'

Data Types: char | string

Output Arguments

dlY — Weighted output features
dlarray

Weighted output features, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input dlX is a formatted dlarray, the output dlY has one dimension labeled 'C' representing the output features, and the same number of 'B' or 'T' dimensions as the input dlX, if either or both are present. If dlX has no 'B' or 'T' dimensions, dlY has the format 'CB', where the 'B' dimension is singleton.

If the input dlX is not a formatted dlarray, output dlY is unformatted. The first dimension of dlY contains the output features. Other dimensions of dlY correspond to
the 'B' and 'T' dimensions of dlX, if either or both are present, and are provided in the same order as in FMT. If dlX has no 'B' or 'T' dimensions, the first dimension of dlY contains the output features and the second dimension is singleton.

More About

Fully Connect Operation

The fullyconnect function connects all outputs of the previous operation to the outputs of the fullyconnect function. For more information, see the definition of “Fully Connected Layer” on page 1-978 on the fullyConnectedLayer reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU:
  - dlX
  - weights
  - bias

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

batchnorm | dlarray | dlconv | dleval | dlgradient | relu | sigmoid | softmax
Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Make Predictions Using Model Function”
“Train a Siamese Network to Compare Images”
“Train Network with Multiple Outputs”

Introduced in R2019b
leakyrelu

Apply leaky rectified linear unit activation

Syntax

dlY = leakyrelu(dlX)
dlY = leakyrelu(dlX,scaleFactor)

Description

The leaky rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is multiplied by a fixed scale factor.

This operation is equivalent to

\[
    f(x) = \begin{cases} 
    x, & x \geq 0 \\
    scale \times x, & x < 0
    \end{cases}
\]

Note This function applies the leaky ReLU operation to dlarray data. If you want to apply leaky ReLU activation within a layerGraph object or Layer array, use the following layer:

• leakyReluLayer

dlY = leakyrelu(dlX) computes the leaky ReLU activation of the input dlX by applying a threshold operation. All values in dlX less than zero are multiplied by a default scale factor of 0.01.

dlY = leakyrelu(dlX,scaleFactor) specifies the scale factor for the leaky ReLU operation.
Examples

Apply Leaky ReLU Activation

Use the `leakyrelu` function to scale negative values in the input data.

Create the input data as a single observation of random values with a height and width of 12 and 32 channels.

```matlab
height = 12;
width = 12;
channels = 32;
observations = 1;

X = randn(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Compute the leaky ReLU activation using a scale factor of `0.05` for the negative values in the input.

```matlab
dlY = leakyrelu(dlX,0.05);
```

Input Arguments

- **dlX** — Input data
  - `dlarray`

  Input data, specified as a `dlarray` with or without dimension labels.
  - Data Types: `single` | `double`

- **scaleFactor** — Scale factor for negative inputs
  - `0.01` (default) | numeric scalar

  Scale factor for negative inputs, specified as a numeric scalar. The default value is `0.01`.
  - Data Types: `single` | `double`
Output Arguments

dlY — Leaky ReLU activations

dlarray

Leaky ReLU activations, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument dlX is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

batchnorm | dlarray | dlconv | dlfeval | dlgradient | relu

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2019b
Istm

Long short-term memory

Syntax

dlY = lstm(dlX,H0,C0,weights,recurrentWeights,bias)
[dly,hiddenState,cellState] = lstm(dlX,H0,C0,weights,
recurrentWeights,bias)
[___] = lstm(___,'DataFormat',FMT)

Description

The long short-term memory (LSTM) operation allows a network to learn long-term
dependencies between time steps in time series and sequence data.

Note This function applies the deep learning LSTM operation to dlarray data. If you
want to apply an LSTM operation within a layerGraph object or Layer array, use the
following layer:
• lstmLayer

dlY = lstm(dlX,H0,C0,weights,recurrentWeights,bias) applies a long short-
term memory (LSTM) calculation to input dlX using the initial hidden state H0, initial cell
state C0, and parameters weights, recurrentWeights, and bias. The input dlX is a
formatted dlarray with dimension labels. The output dlY is a formatted dlarray with
the same dimension labels as dlX, except for any 'S' dimensions.

The lstm function updates the cell and hidden states using the hyperbolic tangent
function (tanh) as the state activation function. The lstm function uses the sigmoid
function given by \( \sigma(x) = (1 + e^{-x})^{-1} \) as the gate activation function.

[dly,hiddenState,cellState] = lstm(dlX,H0,C0,weights,
recurrentWeights,bias) also returns the hidden state and cell state after the LSTM
operation.
[___] = lstm(___,'DataFormat',FMT) also specifies the dimension labels FMT when dlX is not a formatted dlarray. The output dlY is an unformatted dlarray with the same dimension order as dlX, except for any 'S' dimensions.

**Examples**

**Apply LSTM Operation to Sequence Data**

Perform an LSTM operation using three hidden units.

Create the input sequence data as 32 observations with ten channels and a sequence length of 64.

```matlab	numFeatures = 10;
numObservations = 32;
sequenceLength = 64;

X = randn(numFeatures,numObservations,sequenceLength);
dlX = dlarray(X,'CBT');
```

Create the initial hidden and cell states with three hidden units. Use the same initial hidden state and cell state for all observations.

```matlab
numHiddenUnits = 3;
H0 = zeros(numHiddenUnits,1);
C0 = zeros(numHiddenUnits,1);
```

Create the learnable parameters for the LSTM operation.

```matlab
weights = dlarray(randn(4*numHiddenUnits,numFeatures),'CU');
recurrentWeights = dlarray(randn(4*numHiddenUnits,numHiddenUnits),'CU');
bias = dlarray(randn(4*numHiddenUnits,1),'C');
```

Perform the LSTM calculation.

```matlab
[dlY,hiddenState,cellState] = lstm(dlX,H0,C0,weights,recurrentWeights,bias);
```

View the size and dimensions of dlY.

```matlab
size(dlY)
dlY.dims
```
ans = 1×3  
   3    32    64

ans = 'CBT'

View the size of hiddenState and cellState.

size(hiddenState)  
size(cellState)  

ans = 1×2  
   3    32

ans = 1×2  
   3    32

Check that the output hiddenState is equal to the last time step of output dlY.

if extractdata(dlY(:,:,end)) == hiddenState  
    disp("The hidden state and the last time step are equal.")  
else  
    disp("The hidden state and the last time step are not equal.")  
end

The hidden state and the last time step are equal.

You can use the hidden state and cell state to keep track of the state of the LSTM operation and input further sequential data.

### Input Arguments

**dlX — Input data**  
dlarray | numeric array

Input data, specified as a dlarray with or without dimension labels or a numeric array. When dlX is not a formatted dlarray, you must specify the dimension label format using 'DataFormat', FMT. If dlX is a numeric array, at least one of H0, C0, weights, recurrentWeights, or bias must be a dlarray.

dlX must contain a sequence dimension labeled 'T'. If dlX has any spatial dimensions labeled 'S', they are flattened into the 'C' channel dimensions. If dlX has any unspecified dimensions labeled 'U', they must be singleton.
Data Types: single | double

**H₀ — Initial hidden state vector**
dlarray | numeric array

Initial hidden state vector, specified as a `dlarray` with or without dimension labels or a numeric array.

If H₀ is a formatted `dlarray`, it must contain a channel dimension labeled 'C' and optionally a batch dimension labeled 'B' with the same size as the 'B' dimension of dlX. If H₀ does not have a 'B' dimension, the function uses the same hidden state vector for each observation in dlX.

The size of the 'C' dimension determines the number of hidden units. The size of the 'C' dimension of H₀ must be equal to the size of the 'C' dimensions of C₀.

If H₀ is a not a formatted `dlarray`, the size of the first dimension determines the number of hidden units and must be the same size as the first dimension or the 'C' dimension of C₀.

Data Types: single | double

**C₀ — Initial cell state vector**
dlarray | numeric array

Initial cell state vector, specified as a `dlarray` with or without dimension labels or a numeric array.

If C₀ is a formatted `dlarray`, it must contain a channel dimension labeled 'C' and optionally a batch dimension labeled 'B' with the same size as the 'B' dimension of dlX. If C₀ does not have a 'B' dimension, the function uses the same cell state vector for each observation in dlX.

The size of the 'C' dimension determines the number of hidden units. The size of the 'C' dimension of C₀ must be equal to the size of the 'C' dimensions of H₀.

If C₀ is a not a formatted `dlarray`, the size of the first dimension determines the number of hidden units and must be the same size as the first dimension or the 'C' dimension of H₀.

Data Types: single | double

**weights — Weights**
dlarray | numeric array
Weights, specified as a `dlarray` with or without dimension labels or a numeric array.

Specify `weights` as a matrix of size 4*`NumHiddenUnits`-by-`InputSize`, where `NumHiddenUnits` is the size of the 'C' dimension of both `C0` and `H0`, and `InputSize` is the size of the 'C' dimension of `dlX` multiplied by the size of each 'S' dimension of `dlX`, where present.

If `weights` is a formatted `dlarray`, it must contain a 'C' dimension of size 4*`NumHiddenUnits` and a 'U' dimension of size `InputSize`.

Data Types: `single` | `double`

`recurrentWeights` — Recurrent weights
`dlarray` | numeric array

Recurrent weights, specified as a `dlarray` with or without dimension labels or a numeric array.

Specify `recurrentWeights` as a matrix of size 4*`NumHiddenUnits`-by-`NumHiddenUnits`, where `NumHiddenUnits` is the size of the 'C' dimension of both `C0` and `H0`.

If `recurrentWeights` is a formatted `dlarray`, it must contain a 'C' dimension of size 4*`NumHiddenUnits` and a 'U' dimension of size `NumHiddenUnits`.

Data Types: `single` | `double`

`bias` — Bias
`dlarray` vector | numeric vector

Bias, specified as a `dlarray` vector with or without dimension labels or a numeric vector.

Specify `bias` as a vector of length 4*`NumHiddenUnits`, where `NumHiddenUnits` is the size of the 'C' dimension of both `C0` and `H0`.

If `bias` is a formatted `dlarray`, the nonsingleton dimension must be labeled with 'C'.

Data Types: `single` | `double`

`FMT` — Dimension order of unformatted data
`char array` | `string`
Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.

Example: 'DataFormat','SSCB'

Data Types: char | string

**Output Arguments**

*dlY — LSTM output*

dlarray

LSTM output, returned as a dlarray. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX, except for any 'S' dimensions. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.

The size of the 'C' dimension of dlY is the same as the number of hidden units, specified by the size of the 'C' dimension of H0 or C0.

*hiddenState — Hidden state vector*

dlarray | numeric array

Hidden state vector for each observation, returned as a dlarray or a numeric array. hiddenState is returned with the same data type as H0.
If the input $H_0$ is a formatted `dlarray`, the output `hiddenState` is returned as a formatted `dlarray` with the format 'CB'.

**cellState — Cell state vector**

dlarray | numeric array

Cell state vector for each observation, returned as a `dlarray` or a numeric array. `cellState` is returned with the same data type as $C_0$.

If the input $C_0$ is a formatted `dlarray`, the output `cellState` is returned as a formatted `dlarray` with the format 'CB'.

**More About**

**Long Short-Term Memory**

The LSTM operation allows a network to learn long-term dependencies between time steps in time series and sequence data. For more information, see the definition of Long Short-Tem Memory Layer on page 1-1096 on the `lstmLayer` reference page.

**Extended Capabilities**

**GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a `gpuArray` or a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU:
  - `dlX`
  - `H0`
  - `C0`
• weights
• recurrentWeights
• bias

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
dlarray | dlfeval | dlgradient | fullyconnect | softmax

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Sequence-to-Sequence Translation Using Attention”

Introduced in R2019b
maxpool

Pool data to maximum value

Syntax

dlY = maxpool(dlX,poolsize)
[dlY,indx,inputSize] = maxpool(dlX,poolsize)
___ = maxpool(___,'DataFormat',FMT)
___ = maxpool(___,Name,Value)

Description

The maximum pooling operation performs downsampling by dividing the input into pooling regions and computing the maximum value of each region.

Note This function applies the maximum pooling operation to dlarray data. If you want to apply maximum pooling within a layerGraph object or Layer array, use one of the following layers:

- maxPooling2dLayer
- maxPooling3dLayer


dlY = maxpool(dlX,poolsize) performs downsampling by dividing the input dlX into rectangular or cuboidal regions defined by poolsize and computing the maximum value of the data in each region. The input dlX is a formatted dlarray with dimension labels. Pooling acts on spatial dimensions labeled 'S'. The output dlY is a formatted dlarray with the same dimension labels as dlX.

[dlY,indx,inputSize] = maxpool(dlX,poolsize) also returns the linear indices of the maximum value within each pooled region and the size of the input feature map dlX for use with the maxunpool operation.

___ = maxpool(___,'DataFormat',FMT) also specifies the dimension labels FMT when dlX is not a formatted dlarray, in addition to the input arguments in previous
syntaxes. The output dlY is an unformatted dlarray with the same dimension order as dlX.

___ = maxpool(___ ,Name,Value) specifies options using one or more name-value pair arguments. For example, 'Stride',3 sets the stride of the pooling operation.

Examples

Pool Data to Maximum Values

Create the input data as a single observation of random values with a height and width of six and a single channel.

height = 6;
width = 6;
channels = 1;
observations = 1;

X = rand(height,width,channels,observations);
dlX = dlarray(X,'SSCB')

dlX =

  6(S) × 6(S) × 1(C) × 1(B) dlarray

  0.1781  0.8819  0.1564  0.4820  0.2518  0.7302
  0.1280  0.6692  0.8555  0.1206  0.2904  0.3439
  0.9991  0.1904  0.6448  0.5895  0.6171  0.5841
  0.1711  0.3689  0.3763  0.2262  0.2653  0.1078
  0.0326  0.4607  0.1909  0.3846  0.8244  0.9063
  0.5612  0.9816  0.4283  0.5830  0.9827  0.8797

Pool the data to maximum values over pooling regions of size 2 using a stride of 2.

dlY = maxpool(dlX,2,'Stride',2)

dlY =

  3(S) × 3(S) × 1(C) × 1(B) dlarray

  0.8819  0.8555  0.7302
Input Arguments

dlX — Input data
dlarray

Input data, specified as a dlarray with or without dimension labels. When dlX is not a formatted dlarray, you must specify the dimension label format using 'DataFormat',FMT.

Pooling acts on dimensions that you specify as spatial dimensions using the 'S' dimension label. dlX must have at least one 'S' dimension. You can specify up to three dimensions in dlX as 'S' dimensions. The maxpool operation divides the data along each 'S' dimension into regions defined by poolsize. The function computes the maximum of all values within each pooling region.

Data Types: single | double

poolsize — Size of pooling regions
numeric scalar | numeric vector

Size of the pooling regions, specified as a numeric scalar or numeric vector. If you specify poolsize as a scalar, the pooling regions have the same size along all spatial dimensions. To use rectangular or cuboidal pooling regions that have different sizes along each spatial dimension, specify poolsize as a vector with the same length as the number of spatial dimensions.

Example: 3

Data Types: single | double

FMT — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in FMT must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data dlX is an unformatted dlarray.

Example: 'DataFormat','SSCB'

Data Types: char | string

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes.

You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Stride',2 specifies the stride of the pooling regions as 2.

**Stride — Step size for traversing input data**

1 (default) | numeric scalar | numeric vector

Step size for traversing the input data, specified as the comma-separated pair consisting of 'Stride' and a numeric scalar or numeric vector. If you specify 'Stride' as a scalar, the same value is used for all spatial dimensions. If you specify 'Stride' as a vector of the same size as the number of spatial dimensions of the input data, the vector values are used for the corresponding spatial dimensions.

The default value of 'Stride' is 1. If 'Stride' is less than poolsize in any dimension, then the pooling regions overlap.

Example: 'Stride',3

Data Types: single | double

**Padding — Padding applied to edges of data**

0 (default) | 'same' | numeric scalar | numeric vector | numeric matrix
Padding applied to edges of data, specified as the comma-separated pair consisting of 'Padding' and one of the following:

- 'same' — Padding is set so that the output size is the same as the input size when the stride is 1. More generally, the output size of each spatial dimension is \( \text{ceil}(\text{inputSize}/\text{stride}) \), where \( \text{inputSize} \) is the size of the input along a spatial dimension.
- Numeric scalar — The same padding value is applied to both ends of all spatial dimensions.
- Numeric vector — A different padding value is applied along each spatial dimension. Use a vector of size \( d \), where \( d \) is the number of spatial dimensions of the input data. The \( i \)th element of the vector specifies the padding applied to the start and the end along the \( i \)th spatial dimension.
- Numeric matrix — A different padding value is applied to the start and end of each spatial dimension. Use a matrix of size 2-by-\( d \), where \( d \) is the number of spatial dimensions of the input data. The element \((1, d)\) specifies the padding applied to the start of spatial dimension \( d \). The element \((2, d)\) specifies the padding applied to the end of spatial dimension \( d \). For example, in 2-D, the format is \([\text{top, left; bottom, right}]\).

The default value of 'Padding' is 0.

Example: 'Padding','same'

Data Types: single | double

Output Arguments

dlY — Pooled data
dlarray

Pooled data, returned as a dlarray. The output \( \text{dlY} \) has the same underlying data type as the input \( \text{dlX} \).

If the input data \( \text{dlX} \) is a formatted dlarray, \( \text{dlY} \) has the same dimension labels as \( \text{dlX} \). If the input data is not a formatted dlarray, \( \text{dlY} \) is an unformatted dlarray with the same dimension order as the input data.

indx — Indices of maximum values
numeric vector
Indices of maximum values in each pooled region, returned as a numeric vector.

Use the `indx` output with the `maxunpool` function to unpool the output of `maxpool`.

**inputSize — Size of input feature map**
numeric vector

Size of the input feature map, returned as a numeric vector.

Use the `inputSize` output with the `maxunpool` function to unpool the output of `maxpool`.

**More About**

**Maximum Pooling**

The `maxpool` function pools the input data to maximum values over the spatial dimensions. For more information, see the definition of “Max Pooling Layer” on page 1-1004 on the `maxPooling2dLayer` reference page.

**Extended Capabilities**

**GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument `dlX` is a `dlarray` with underlying data of type `gpuArray`, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).
See Also
avgpool | dlarray | dlconv | dlfeval | dlgradient | maxunpool

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2019b
maxunpool

Unpool the output of a maximum pooling operation

Syntax

```matlab
dlY = maxunpool(dlX,indx,outputSize)
dlY = maxunpool(dlX,indx,outputSize,'DataFormat',FMT)
```

Description

The maximum unpooling operation unpools the output of a maximum pooling operation by upsampling and padding with zeros.

**Note** This function applies the maximum unpooling operation to **dlarray** data. If you want to apply maximum unpooling within a **layerGraph** object or **Layer** array, use the following layer:

- maxUnpooling2dLayer

```matlab
dlY = maxunpool(dlX,indx,outputSize) upsamples the spatial dimensions of input data dlX to match the size outputSize. The data is padded with zeros between the locations of maximum values specified by indx. The input dlX is a formatted dlarray with dimension labels. The output dlY is a formatted dlarray with the same dimension labels as dlX.

dlY = maxunpool(dlX,indx,outputSize,'DataFormat',FMT) also specifies the dimension labels FMT when dlX is not a formatted dlarray. The output dlY is an unformatted dlarray with the same dimension order as dlX.
```

Examples
Unpool Pooled Data

Create the input data as a single observation of random values with a height and width of six and a single channel.

```matlab
definition = height = 6;
definition = width = 6;
definition = channels = 1;
definition = observations = 1;

X = rand(height, width, channels, observations);
dlX = dlarray(X, 'SSCB')
```

Pool the data to maximum values over pooling regions of size 2 using a stride of 2.

```matlab
[dlY, indx, dataSize] = maxpool(dlX, 2, 'Stride', 2);
```

```matlab
dlX =
6(S) × 6(S) × 1(C) × 1(B) dlarray

0.8147  0.2785  0.9572  0.7922  0.6787  0.7060
0.9058  0.5469  0.4854  0.9595  0.7577  0.0318
0.1270  0.9575  0.8003  0.6557  0.7431  0.2769
0.9134  0.9649  0.1419  0.0357  0.3922  0.0462
0.6324  0.1576  0.4218  0.8491  0.6555  0.0971
0.0975  0.9706  0.9157  0.9340  0.1712  0.8235

dlY =
3(S) × 3(S) × 1(C) × 1(B) dlarray

0.9058  0.9595  0.7577
0.9649  0.8003  0.7431
0.9706  0.9340  0.8235

indx = 9×1

2
10
12
20
15
24
26
27
36
Unpool the data using the indices and output size from the `maxpool` operation.

\[ \text{dlY} = \text{maxunpool}(\text{dlY}, \text{indx}, \text{dataSize}) \]

```
dlY =
  6(S) \times 6(S) \times 1(C) \times 1(B) \text{ dlarray}
0         0         0         0         0         0
0.9058    0         0          0.9595    0.7577    0
0         0          0.8003    0          0.7431    0
0         0.9649    0         0          0         0
0         0         0         0          0         0
0         0         0          0.9340    0       0.8235
```

**Input Arguments**

- **dlX** — Input data
  
  dlarray
  
  Input data, specified as a `dlarray` with or without dimension labels. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using `'DataFormat',FMT.'

  Unpooling acts on dimensions that you specify as spatial dimensions using the 'S' dimension label. `dlX` must have at least one 'S' dimension. You can specify up to three dimensions in `dlX` as 'S' dimensions. Use the `dlY` output of the `maxpool` function as the `dlX` input to `maxunpool`.

  Data Types: `single` | `double`

- **indx** — Indices of maximum values
  
  numeric array
  
  Indices of maximum values in each pooled region, specified as a numeric array.

  Use the `indx` output of the `maxpool` function as the `indx` input to `maxpool`.

  Data Types: `single` | `double`
**outputSize — Size of output feature map**

numeric array

Size of the output feature map, specified as a numeric array.

Use the `inputSize` output of the `maxpool` function as the `outputSize` input to `maxunpool`.

Data Types: `single` | `double`

**FMT — Dimension order of unformatted data**

char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of `'DataFormat'` and a character array or string that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- `'S'` — Spatial
- `'C'` — Channel
- `'B'` — Batch (for example, samples and observations)
- `'T'` — Time (for example, sequences)
- `'U'` — Unspecified

You can specify multiple dimensions labeled `'S'` or `'U'`. You can use the labels `'C'`, `'B'`, or `'T' at most once.

You must specify `'DataFormat'` when the input data `dlX` is an unformatted `dlarray`.

Example: `'DataFormat','SSCB'`

Data Types: `char` | `string`

**Output Arguments**

**dlY — Unpooled data**

dlarray

Unpooled data, returned as a `dlarray`. The output `dlY` has the same underlying data type as the input `dlX`.  

1-1655
If the input data \( dlX \) is a formatted \( dlarray \), \( dlY \) has the same dimension labels as \( dlX \). If the input data is not a formatted \( dlarray \), \( dlY \) is an unformatted \( dlarray \) with the same dimension order as the input data.

### Extended Capabilities

#### GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument \( dlX \) is a \( dlarray \) with underlying data of type \( gpuArray \), this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

#### See Also

- \( dlarray \)
- \( dlfeval \)
- \( dlgradient \)
- \( maxpool \)

#### Topics

- “Define Custom Training Loops, Loss Functions, and Networks”
- “Train Network Using Model Function”

### Introduced in R2019b
**mse**

Half mean squared error

**Syntax**

\[
\text{dlY} = \text{mse}(\text{dlX}, \text{targets})
\]

\[
\text{dlY} = \text{mse}(\text{dlX}, \text{targets}, 'DataFormat', \text{FMT})
\]

**Description**

The half mean squared error operation computes the half mean squared error loss between network predictions and target values for regression problems.

The loss is calculated using the following formula

\[
\text{loss} = \frac{1}{2N} \sum_{i=1}^{M} (X_i - T_i)^2
\]

where \(X_i\) is the network response, \(T_i\) is the target value, \(M\) is the total number of responses in \(X\) (across all observations), and \(N\) is the total number of observations in \(X\).

**Note** This function computes the half mean squared error loss between predictions and targets stored as `dlarray` data. If you want to calculate the half mean squared error loss within a `layerGraph` object or `Layer` array for use with `trainNetwork`, use the following layer:

- `regressionLayer`

\[
\text{dlY} = \text{mse}(\text{dlX}, \text{targets})
\]

computes the half mean squared error loss between the predictions `dlX` and the target values `targets` for regression problems. The input `dlX` is a formatted `dlarray` with dimension labels. The output `dlY` is an unformatted scalar `dlarray` with no dimension labels.
dlY = mse(dlX,targets,'DataFormat',FMT) also specifies the dimension labels FMT when dlX is not a formatted dlarray.

Examples

Find Half Mean Squared Error Between Predicted and Target Values

The half mean squared error evaluates how well the network predictions correspond to the target values.

Create the input predictions as a single observation of random values with a height and width of six and a single channel.

\[
\begin{align*}
\text{height} &= 6; \\
\text{width} &= 6; \\
\text{channels} &= 1; \\
\text{observations} &= 1;
\end{align*}
\]

\[
X = \text{rand}([\text{height} \times \text{width} \times \text{channels} \times \text{observations}]);
\]
\[
dlX = \text{dlarray}(X,'SSCB')
\]

Create the target values as a numeric array with the same dimension order as the input data dlX.

\[
\text{targets} = \text{ones}([\text{height} \times \text{width} \times \text{channels} \times \text{observations}]);
\]

Compute the half mean squared error between the predictions and the targets.

\[
dlY = \text{mse}(dlX,\text{targets})
\]

\[
dlY = \\
1x1 \text{ dlarray} \\
5.2061
\]

Input Arguments

dlX — Predictions

dlarray | numeric array
Predictions, specified as a `dlarray` with or without dimension labels or a numeric array. When `dlX` is not a formatted `dlarray`, you must specify the dimension label format using 'DataFormat', FMT. If `dlX` is a numeric array, `targets` must be a `dlarray`.

Data Types: `single` | `double`

`targets` — Target values
`dlarray` | numeric array

Target values, specified as a `dlarray` with or without dimension labels or a numeric array. If `targets` is an unformatted `dlarray` or a numeric array, it must have the same dimension order as the input `dlX`. The size of each dimension of `targets` must match the size of each corresponding dimension in `dlX`.

Data Types: `single` | `double`

`FMT` — Dimension order of unformatted data
char array | string

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in `FMT` must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data `dlX` is an unformatted `dlarray`.

Example: 'DataFormat','SSCB'

Data Types: char | string
Output Arguments

dlY — Half mean squared error loss
dlarray scalar

Half mean squared error loss, returned as a dlarray scalar without dimension labels. The output dlY has the same underlying data type as the input dlX.

More About

Half Mean Squared Error Loss

The mse function computes the half-mean-squared-error loss for regression problems. For more information, see the definition of “Regression Output Layer” on page 1-1021 on the RegressionOutputLayer reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When at least one of the following input arguments is a gpuArray or a dlarray with underlying data of type gpuArray, this function runs on the GPU:
  - dlX
  - targets

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).
See Also
crossentropy | dlarray | dlfeval | dlgradient | sigmoid

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network with Multiple Outputs”

Introduced in R2019b
relu

Apply rectified linear unit activation

Syntax

dlY = relu(dlX)

Description

The rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is set to zero.

This operation is equivalent to

\[ f(x) = \begin{cases} 
    x, & x > 0 \\
    0, & x \leq 0.
\end{cases} \]

Note This function applies the ReLU operation to dlarray data. If you want to apply ReLU activation within a layerGraph object or Layer array, use the following layer:

- reluLayer

\[ \text{dlY} = \text{relu} (\text{dlX}) \] computes the ReLU activation of the input dlX by applying a threshold operation. All values in dlX that are less than zero are set to zero.

Examples

Apply ReLU Activation

Use the relu function to set negative values in the input data to zero.
Create the input data as a single observation of random values with a height and width of 12 and 32 channels.

```matlab
height = 12;
width = 12;
channels = 32;
observations = 1;

X = randn(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```

Compute the leaky ReLU activation.

```matlab
dlY = relu(dlX);
```

All negative values in `dlX` are now set to 0.

**Input Arguments**

- **dlX** — Input data
dlarray

Input data, specified as a dlarray with or without dimension labels.

Data Types: single | double

**Output Arguments**

- **dlY** — ReLU activations
dlarray

ReLU activations, returned as a dlarray. The output `dlY` has the same underlying data type as the input `dlX`.

If the input data `dlX` is a formatted dlarray, `dlY` has the same dimension labels as `dlX`. If the input data is not a formatted dlarray, `dlY` is an unformatted dlarray with the same dimension order as the input data.
Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument dlX is a dlarray with underlying data of type gpuArray, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
batchnorm | dlarray | dlconv | dlfeval | dlgradient | leakyrelu

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

Introduced in R2019b
sigmoid

Apply sigmoid activation

Syntax

dlY = sigmoid(dlX)

Description

The sigmoid activation operation applies the sigmoid function to the input data.

This operation is equivalent to

\[
    f(x) = \frac{1}{1 + e^{-x}}
\]

dlY = sigmoid(dlX) computes the sigmoid activation of the input dlX by applying the sigmoid transfer function. All values in dlY are between 0 and 1.

Examples

Apply Sigmoid Activation

Use the sigmoid function to set all values in the input data to a value between 0 and 1.

Create the input data as a single observation of random values with a height and width of seven and 32 channels.

```matlab
height = 7;
width = 7;
channels = 32;
observations = 1;

X = randn(height,width,channels,observations);
dlX = dlarray(X,'SSCB');
```
Compute the sigmoid activation.

\[ \text{dlY} = \text{sigmoid} (\text{dlX}) ; \]

All values in \( \text{dlY} \) now range between 0 and 1.

**Input Arguments**

\( \text{dlX} \) — Input data  
\text{dlarray}

Input data, specified as a \text{dlarray} with or without dimension labels.

Data Types: \text{single} | \text{double}

**Output Arguments**

\( \text{dlY} \) — Sigmoid activations  
\text{dlarray}

Sigmoid activations, returned as a \text{dlarray}. All values in \( \text{dlY} \) are between 0 and 1. The output \( \text{dlY} \) has the same underlying data type as the input \( \text{dlX} \).

If the input data \( \text{dlX} \) is a formatted \text{dlarray}, \( \text{dlY} \) has the same dimension labels as \( \text{dlX} \). If the input data is not a formatted \text{dlarray}, \( \text{dlY} \) is an unformatted \text{dlarray} with the same dimension order as the input data.

**Extended Capabilities**

**GPU Arrays**

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:
• When the input argument dlX is a dlarray with underlying data of type gpuArray, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

**See Also**
batchnorm | dlarray | dlfeval | dlgradient | leakyrelu | relu

**Topics**
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”

**Introduced in R2019b**
softmax

Apply softmax activation to channel dimension

Syntax

dlY = softmax(dlX)
dlY = softmax(dlX,'DataFormat',FMT)

Description

The softmax activation operation applies the softmax function to the channel dimension of the input data.

The softmax function normalizes the value of the input data across the channel dimension such that it sums to one. You can regard the output of the softmax function as a probability distribution.

Note This function applies the softmax operation to dlarray data. If you want to apply softmax within a layerGraph object or Layer array, use the following layer:

• softmaxLayer

dlY = softmax(dlX) computes the softmax activation of the input dlX by applying the softmax transfer function to the channel dimension of the input data. All values in dlY are between 0 and 1, and sum to 1. The input dlX is a formatted dlarray with dimension labels. The output dlY is a formatted dlarray with the same dimension labels as dlX.

dlY = softmax(dlX,'DataFormat',FMT) also specifies dimension labels FMT when dlX is not a formatted dlarray. The output dlY is an unformatted dlarray with the same dimension order as dlX.
Examples

Apply Softmax Activation

Use the softmax function to set all values in the input data to values between 0 and 1 that sum to 1 over all channels.

Create the input classification data as two observations of random variables. The data can be in any of 10 categories.

```matlab
numCategories = 10;
observations = 2;

X = rand(numCategories,observations);
dlX = dlarray(X,'CB');

Compute the softmax activation.

dlY = softmax(dlX);
totalProb = sum(dlY,1)
```

```
dlY =

10(C) x 2(B) dlarray
0.1151    0.0578
0.1261    0.1303
0.0579    0.1285
0.1270    0.0802
0.0959    0.1099
0.0562    0.0569
0.0673    0.0753
0.0880    0.1233
0.1328    0.1090
0.1337    0.1288

totalProb =

1(C) x 2(B) dlarray
1.0000    1.0000
```
All values in \( \text{dlY} \) range between 0 and 1. The values over all channels sum to 1 for each observation.

**Input Arguments**

**\( \text{dlX} \) — Input data**

\[ \text{dlarray} \]

Input data, specified as a \text{dlarray} with or without dimension labels. When \( \text{dlX} \) is not a formatted \text{dlarray}, you must specify the dimension label format using 'DataFormat', \( \text{FMT} \).

\( \text{dlX} \) must contain a 'C' channel dimension.

Data Types: single | double

**\( \text{FMT} \) — Dimension order of unformatted data**

\[ \text{char array | string} \]

Dimension order of unformatted input data, specified as the comma-separated pair consisting of 'DataFormat' and a character array or string that provides a label for each dimension of the data. Each character in \( \text{FMT} \) must be one of the following:

- 'S' — Spatial
- 'C' — Channel
- 'B' — Batch (for example, samples and observations)
- 'T' — Time (for example, sequences)
- 'U' — Unspecified

You can specify multiple dimensions labeled 'S' or 'U'. You can use the labels 'C', 'B', or 'T' at most once.

You must specify 'DataFormat' when the input data \( \text{dlX} \) is an unformatted \text{dlarray}.

Example: 'DataFormat', 'SSCB'

Data Types: char | string
Output Arguments

dlY — Softmax activations
dlarray

Softmax activations, returned as a dlarray. All values in dlY are between 0 and 1. The output dlY has the same underlying data type as the input dlX.

If the input data dlX is a formatted dlarray, dlY has the same dimension labels as dlX. If the input data is not a formatted dlarray, dlY is an unformatted dlarray with the same dimension order as the input data.

More About

Softmax Activation

The softmax function normalizes the input across the channel dimension, such that it sums to one. For more information, see the definition of “Softmax Layer” on page 1-1057 on the softmaxLayer reference page.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When the input argument dlX is a dlarray with underlying data of type gpuArray, this function runs on the GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).
See Also
batchnorm | crossentropy | dlassay | dlfeval | dlgradient | fullyconnect | relu

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Train Network Using Model Function”
“Make Predictions Using Model Function”
“Train Network with Multiple Outputs”

Introduced in R2019b
adamupdate

Update parameters using adaptive moment estimation (Adam)

Syntax

[dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,
averageGrad,averageSqGrad,iteration)
[params,averageGrad,averageSqGrad] = adamupdate(params,grad,
averageGrad,averageSqGrad,iteration)
[___] = adamupdate( ___ learnRate,gradDecay,sqGradDecay,epsilon)

Description

Update the network learnable parameters in a custom training loop using the adaptive
moment estimation (Adam) algorithm.

Note This function applies the Adam optimization algorithm to update network
parameters in custom training loops that use networks defined as dlnetwork objects or
model functions. If you want to train a network defined as a Layer array or as a
LayerGraph, use the following functions:

• Create a TrainingOptionsADAM object using the trainingOptions function.
• Use the TrainingOptionsADAM object with the trainNetwork function.

[dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,
averageGrad,averageSqGrad,iteration) updates the learnable parameters of the
network dlnet using the Adam algorithm. Use this syntax in a training loop to iteratively
update a network defined as a dlnetwork object.

[params,averageGrad,averageSqGrad] = adamupdate(params,grad,
averageGrad,averageSqGrad,iteration) updates the learnable parameters in
params using the Adam algorithm. Use this syntax in a training loop to iteratively update
the learnable parameters of a network defined using functions.
[___] = adamupdate(___learnRate,gradDecay,sqGradDecay,epsilon) also specifies values to use for the global learning rate, gradient decay, square gradient decay, and small constant epsilon, in addition to the input arguments in previous syntaxes.

Examples

Update Learnable Parameters Using adamupdate

Perform a single adaptive moment estimation update step with a global learning rate of 0.05, gradient decay factor of 0.75, and squared gradient decay factor of 0.95.

Create the parameters and parameter gradients as numeric arrays.

params = rand(3,3,4);
grad = ones(3,3,4);

Initialize the iteration counter, average gradient, and average squared gradient for the first iteration.

iteration = 1;
averageGrad = [];
averageSqGrad = [];

Specify custom values for the global learning rate, gradient decay factor, and squared gradient decay factor.

learnRate = 0.05;
gradDecay = 0.75;
sqGradDecay = 0.95;

Update the learnable parameters using adamupdate.

[params,averageGrad,averageSqGrad] = adamupdate(params,grad,averageGrad,averageSqGrad,iteration,learnRate,gradDecay,sqGradDecay);

Update the iteration counter.

iteration = iteration + 1;
Train a Network Using `adamupdate`

Use `adamupdate` to train a network using the adaptive moment estimation (Adam) algorithm.

Load Training Data

Load the digits training data.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
umClasses = numel(classes);
```

Define the Network

Define the network and specify the average image using the 'Mean' option in the image input layer.

```matlab
layers = [imageInputLayer([28 28 1], 'Name','input','Mean',mean(XTrain,4))
  convolution2dLayer(5,20,'Name','conv1')
  reluLayer('Name', 'relu1')
  convolution2dLayer(3,20,'Padding',1,'Name','conv2')
  reluLayer('Name','relu2')
  convolution2dLayer(3,20,'Padding',1,'Name','conv3')
  reluLayer('Name','relu3')
  fullyConnectedLayer(numClasses,'Name','fc')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```matlab
dlnet = dlnetwork(lgraph);
```

Define the Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding labels `Y` and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Specify Training Options

Specify the options to use during training.

```matlab
miniBatchSize = 128;
numEpochs = 20;
```
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

executionEnvironment = "auto";

Initialize average gradients and squared average gradients.

averageGrad = [];
averageSqGrad = [];

Initialize iteration counter.

iteration = 1;

Initialize the training progress plot.

plots = "training-progress";
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Total Iterations")
    ylabel("Loss")
end

**Train the Network**

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters using the adamupdate function. At the end of each epoch, display the training progress.

```matlab
for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:, :, :, idx);
    YTrain = YTrain(idx);

    for i = 1:numIterationsPerEpoch
        % Read mini-batch of data and convert the labels to dummy variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:, :, :, idx);
```
Y = zeros(numClasses, miniBatchSize, 'single');
for c = 1:numClasses
  Y(c,YTrain(idx) == classes(c)) = 1;
end

% Convert mini-batch of data to dlarray.
dlX = dlarray(single(X),'SSCB');

% If training on a GPU, then convert data to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
dlX = gpuArray(dlX);
end

% Evaluate the model gradients and loss using dlfeval and the
% modelGradients function.
[grad,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

% Update the network parameters using the Adam optimizer.
[dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,grad,averageGrad,averageSqGrad,iteration);

% Display the training progress.
if plots == "training-progress"
  addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
  title("Loss During Training: Epoch - " + epoch + "; Iteration - " + i)
drawnow
end

% Increment iteration counter
iteration = iteration + 1;
end
Test the Network

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

```matlab
[XTest, YTest] = digitTest4DArrayData;
Convert the data to a `dlarray` object with dimension format 'SSCB'. For GPU prediction, also convert the data to `gpuArray`.

dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" || canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end```
To classify images using a dlnetwork object, use the predict function and find the classes with the highest scores.

```matlab
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```matlab
accuracy = mean(YPred==YTest)
```

accuracy = 0.9908

**Model Gradients Function**

The modelGradients function takes a dlnetwork object dlnet, a mini-batch of input data dlX with corresponding labels Y and returns the loss and the gradients of the loss with respect to the learnable parameters in dlnet. To compute the gradients automatically, use the dlgradient function.

```matlab
function [gradients,loss] = modelGradients(dlnet,dlX,Y)
    dlYPred = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);
    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);
end
```

**Input Arguments**

**dlnet — Network**

dlnetwork object

Network, specified as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object. dlnet.Learnables is a table with three variables:

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
Value — Value of parameter, specified as a cell array containing a dlarray.

The input argument grad must be a table of the same form as dlnet.Learnables.

**params — Network learnable parameters**

dlarray | numeric array | cell array | structure | table

Network learnable parameters, specified as a dlarray, a numeric array, a cell array, a structure, or a table.

If you specify params as a table, it must contain the following three variables:

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a dlarray.

You can specify params as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be dlarray or numeric values of data type double or single.

The input argument grad must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as params.

Data Types: single | double | struct | table | cell

**grad — Gradients of the loss**

dlarray | numeric array | cell array | structure | table

Gradients of the loss, specified as a dlarray, a numeric array, a cell array, a structure, or a table.

The exact form of grad depends on the input network or learnable parameters. The following table shows the required format for grad for possible inputs to adamupdate.
<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>Gradients</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlnet</td>
<td>Table dlnet.Learnables containing Layer, Parameter, and Value variables. The Value variable consists of cell arrays that contain each learnable parameter as a dlarray.</td>
<td>Table with the same data type, variables, and ordering as dlnet.Learnables. grad must have a Value variable consisting of cell arrays that contain the gradient of each learnable parameter.</td>
</tr>
<tr>
<td>params</td>
<td>dlarray</td>
<td>dlarray with the same data type and ordering as params</td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as params</td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as params</td>
</tr>
<tr>
<td></td>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as params</td>
</tr>
<tr>
<td></td>
<td>Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a dlarray.</td>
<td>Table with the same data types, variables, and ordering as params. grad must have a Value variable consisting of cell arrays that contain the gradient of each learnable parameter.</td>
</tr>
</tbody>
</table>

You can obtain grad from a call to dlfeval that evaluates a function that contains a call to dlgradient. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox”.

**averageGrad — Moving average of parameter gradients**

[ | dlarray | numeric array | cell array | structure | table

Moving average of parameter gradients, specified as an empty array, a dlarray, a numeric array, a cell array, a structure, or a table.
The exact form of `averageGrad` depends on the input network or learnable parameters. The following table shows the required format for `averageGrad` for possible inputs to `adamupdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>Average Gradients</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. <code>averageGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average gradient of each learnable parameter.</td>
</tr>
<tr>
<td><code>params</code></td>
<td><code>dlarray</code></td>
<td><code>dlarray</code> with the same data type and ordering as <code>params</code>.</td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as <code>params</code>.</td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code>.</td>
</tr>
<tr>
<td></td>
<td><code>Structure</code></td>
<td><code>Structure</code> with the same data types, fields, and ordering as <code>params</code>.</td>
</tr>
<tr>
<td></td>
<td>Table with <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data types, variables, and ordering as <code>params</code>. <code>averageGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average gradient of each learnable parameter.</td>
</tr>
</tbody>
</table>

If you specify `averageGrad` and `averageSqGrad` as empty arrays, the function assumes no previous gradients and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `averageGrad` output of a previous call to `adamupdate` as the `averageGrad` input.
averageSqGrad — Moving average of squared parameter gradients

Moving average of squared parameter gradients, specified as an empty array, a `dlarray`,
a numeric array, a cell array, a structure, or a table.

The exact form of `averageSqGrad` depends on the input network or learnable
parameters. The following table shows the required format for `averageSqGrad` for
possible inputs to `adamupdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable parameters</th>
<th>Average Squared Gradients</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. <code>averageSqGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.</td>
</tr>
<tr>
<td><code>params</code></td>
<td><code>dlarray</code></td>
<td><code>dlarray</code> with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td><code>Numeric array</code></td>
<td>Numeric array with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td><code>Cell array</code></td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td><code>Structure</code></td>
<td>Structure with the same data types, fields, and ordering as <code>params</code></td>
</tr>
<tr>
<td>Input</td>
<td>Learnable parameters</td>
<td>Average Squared Gradients</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td></td>
<td>Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a dlarray.</td>
<td>Table with the same data types, variables and ordering as params. averageSqGrad must have a Value variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.</td>
</tr>
</tbody>
</table>

If you specify `averageGrad` and `averageSqGrad` as empty arrays, the function assumes no previous gradients and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `averageSqGrad` output of a previous call to `adamupdate` as the `averageSqGrad` input.

**iteration — Iteration number**

positive integer

Iteration number, specified as a positive integer. For the first call to `adamupdate`, use a value of 1. You must increment `iteration` by 1 for each subsequent call in a series of calls to `adamupdate`. The Adam algorithm uses this value to correct for bias in the moving averages at the beginning of a set of iterations.

**learnRate — Global learning rate**

0.001 (default) | positive scalar

Global learning rate, specified as a positive scalar. The default value of `learnRate` is 0.001.

If you specify the network parameters as a `dlnetwork`, the learning rate for each parameter is the global learning rate multiplied by the corresponding learning rate factor property defined in the network layers.

**gradDecay — Gradient decay factor**

0.9 (default) | positive scalar between 0 and 1

Gradient decay factor, specified as a positive scalar between 0 and 1. The default value of `gradDecay` is 0.9.
**sqGradDecay — Squared gradient decay factor**

0.999 (default) | positive scalar between 0 and 1

Squared gradient decay factor, specified as a positive scalar between 0 and 1. The default value of `sqGradDecay` is 0.999.

**epsilon — Small constant**

1e-8 (default) | positive scalar

Small constant for preventing divide-by-zero errors, specified as a positive scalar. The default value of `epsilon` is 1e-8.

### Output Arguments

**dlnet — Updated network**

dlnetwork object

Network, returned as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object.

**params — Updated network learnable parameters**

dlarray | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table with a Value variable containing the updated learnable parameters of the network.

**averageGrad — Updated moving average of parameter gradients**

dlarray | numeric array | cell array | structure | table

Updated moving average of parameter gradients, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table.

**averageSqGrad — Updated moving average of squared parameter gradients**

dlarray | numeric array | cell array | structure | table

Updated moving average of squared parameter gradients, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table.
More About

Adam

The function uses the adaptive moment estimation (Adam) algorithm to update the learnable parameters. For more information, see the definition of the Adam algorithm under “Stochastic Gradient Descent” on page 1-691 on the trainingOptions reference page.

See Also
dlarray | dlfeval | dlgradient | dlnetwork | dlupdate | forward | rmspropupdate | sgdupdate

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”

Introduced in R2019b
dlupdate

Update parameters using custom function

Syntax

dlnet = dlupdate(fun,dlnet)
params = dlupdate(fun,params)
[___] = dlupdate(fun,___A1,...,An)
[___,X1,...,Xm] = dlupdate(fun,___)

Description

dlnet = dlupdate(fun,dlnet) updates the learnable parameters of the dlnetwork object dlnet by evaluating the function fun with each learnable parameter as an input. fun is a function handle to a function that takes one parameter array as an input argument and returns an updated parameter array.

params = dlupdate(fun,params) updates the learnable parameters in params by evaluating the function fun with each learnable parameter as an input.

[___] = dlupdate(fun,___A1,...,An) also specifies additional input arguments, in addition to the input arguments in previous syntaxes, when fun is a function handle to a function that requires n+1 input values.

[___,X1,...,Xm] = dlupdate(fun,___) returns multiple outputs X1,...,Xm when fun is a function handle to a function that returns m+1 output values.

Examples

L1 Regularization with dlupdate

Perform L1 regularization on a structure of parameter gradients.

Create the sample input data.
```plaintext
dlX = dlarray(rand(100,100,3),'SSC');

Initialize the learnable parameters for the convolution operation.

params.Weights = dlarray(rand(10,10,3,50));
params.Bias = dlarray(rand(50,1));

Calculate the gradients for the convolution operation using the helper function convGradients, defined at the end of this example.

grads = dlfeval(@convGradients,dlX,params);

Define the regularization factor.

L1Factor = 0.001;

Create an anonymous function that regularizes the gradients. By using an anonymous function to pass a scalar constant to the function, you can avoid having to expand the constant value to the same size and structure as the parameter variable.

L1Regularizer = @(grad,param) grad + L1Factor.*sign(param);

Use dlupdate to apply the regularization function to each of the gradients.

grads = dlupdate(L1Regularizer,grads,params);

The gradients in grads are now regularized according to the function L1Regularizer.

**convGradients Function**

The convGradients helper function takes the learnable parameters of the convolution operation and a mini-batch of input data dlX, and returns the gradients with respect to the learnable parameters.

```plaintext
function grads = convGradients(dlX,params)
dlY = dlconv(dlX,params.Weights,params.Bias);
dlY = sum(dlY,'all');
grads = dlgradient(dlY,params);
end
```
Use `dlupdate` to Train a Network Using a Custom Update Function

Use `dlupdate` to train a network using a custom update function that implements the stochastic gradient descent algorithm (without momentum).

Load Training Data

Load the digits training data.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
umClasses = numel(classes);
```

Define the Network

Define the network architecture and specify the average image using the 'Mean' option in the image input layer.

```matlab
layers = [    imageInputLayer([28 28 1], 'Name','input','Mean',mean(XTrain,4))    convolution2dLayer(5,20,'Name','conv1')    reluLayer('Name','relu1')    convolution2dLayer(3,20,'Padding',1,'Name','conv2')    reluLayer('Name','relu2')    convolution2dLayer(3,20,'Padding',1,'Name','conv3')    reluLayer('Name','relu3')    fullyConnectedLayer(numClasses,'Name','fc')]
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```matlab
dlnet = dlnetwork(lgraph);
```

Define the Model Gradients Function

Create the function `modelGradients`, listed at the end of this example, that takes a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Define the Stochastic Gradient Descent Function

Create the function `sgdFunction`, listed at the end of this example, that takes `param` and `paramGradient`, a learnable parameter and the gradient of the loss with respect to
that parameter, respectively. The function returns the updated parameter using the stochastic gradient descent algorithm, expressed as

\[ \theta_{l+1} = \theta - \alpha \nabla E(\theta_l) \]

where \( l \) is the iteration number, \( \alpha > 0 \) is the learning rate, \( \theta \) is the parameter vector, and \( E(\theta) \) is the loss function.

**Specify Training Options**

Specify the options to use during training.

```matlab
miniBatchSize = 128;
numEpochs = 20;
umObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```matlab
executionEnvironment = "auto";
```

Initialize the velocity parameter.

```matlab
learnRate = 0.001;
```

Initialize the training progress plot.

```matlab
plots = "training-progress";
if plots == "training-progress"
    iteration = 1;
    figure
    lineLossTrain = animatedline;
xlabel("Total Iterations")
ylabel("Loss")
end
```

**Train the Network**

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters by calling `dlupdate` with the function `sgdFunction` defined at the end of this example. At the end of each epoch, display the training progress.
for epoch = 1:numEpochs
  % Shuffle data.
  idx = randperm(numel(YTrain));
  XTrain = XTrain(:,:,:,:,idx);
  YTrain = YTrain(idx);

  for i = 1:numIterationsPerEpoch
    % Read mini-batch of data and convert the labels to dummy variables.
    idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
    X = XTrain(:,:,:,:,idx);

    Y = zeros(numClasses, miniBatchSize, 'single');
    for c = 1:numClasses
      Y(c,YTrain(idx)==classes(c)) = 1;
    end

    % Convert mini-batch of data to dlarray.
    dlX = dlarray(single(X),'SSCB');

    % If training on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
      dlX = gpuArray(dlX);
    end

    % Evaluate the model gradients and loss using dlfeval and the % modelGradients function.
    [grad,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

    % Update the network parameters using the SGD algorithm defined in the % sgdFunction.
    dlnet = dlupdate(@sgdFunction,dlnet,grad);

    % Display the training progress.
    if plots == "training-progress"
      addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
      title("Loss During Training: Epoch - " + epoch + "; Iteration - " + i)
      drawnow
      iteration = iteration + 1;
    end
  end
end
Test the Network

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

[XTest, YTest] = digitTest4DArrayData;

Convert the data to a dlarray object with dimension format 'SSCB'. For GPU prediction, also convert the data to gpuArray.

dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
To classify images using a `dlnetwork` object, use the `predict` function and find the classes with the highest scores.

```
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred==YTest)
accuracy = 0.7282
```

**Model Gradients Function**

The function `modelGradients` takes a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,loss] = modelGradients(dlnet,dlX,Y)
    dlYPred = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);
    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);
end
```

**Stochastic Gradient Descent Function**

The function `sgdFunction` takes `param` and `paramGradient`, a learnable parameter and the gradient of the loss with respect to that parameter, and returns the updated parameter using the stochastic gradient descent algorithm, expressed as

\[
\theta_{l+1} = \theta - \alpha \nabla E(\theta_l)
\]

where \( l \) is the iteration number, \( \alpha > 0 \) is the learning rate, \( \theta \) is the parameter vector, and \( E(\theta) \) is the loss function.

```
function param = sgdFunction(param,paramGradient)
    learnRate = 0.01;
end
```
param = param - learnRate.*paramGradient;
end

## Input Arguments

**fun — Function to apply**

function handle

Function to apply to the learnable parameters, specified as a function handle. dlupdate evaluates fun with each network learnable parameter as an input. fun is evaluated as many times as there are arrays of learnable parameters in dlnet or params.

**dlnet — Network**
dlnetwork object

Network, specified as a dlnetwork object. The function updates the dlnet.Learnables property of the dlnetwork object. dlnet.Learnables is a table with three variables:

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a dlarray.

**params — Network learnable parameters**
dlarray | numeric array | cell array | structure | table

Network learnable parameters, specified as a dlarray, a numeric array, a cell array, a structure, or a table.

If you specify params as a table, it must contain the following three variables.

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a dlarray.

You can specify params as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable
parameters inside the cell array, structure, or table must be `dlarray` or numeric values of data type `double` or `single`.

The input argument `grad` must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as `params`.

**Data Types:** `single`, `double`, `struct`, `table`, `cell`

### A1,...,An — Additional input arguments

dlarray | numeric array | cell array | structure | table

Additional input arguments to `fun`, specified as `dlarray` objects, numeric arrays, cell arrays, structures, or tables with a Value variable.

The exact form of `A1,...,An` depends on the input network or learnable parameters. The following table shows the required format for `A1,...,An` for possible inputs to `dlupdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>A1,...,An</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. <code>A1,...,An</code> must have a Value variable consisting of cell arrays that contain the additional input arguments for the function <code>fun</code> to apply to each learnable parameter.</td>
</tr>
<tr>
<td><code>params</code></td>
<td>dlarray</td>
<td>dlarray with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code></td>
</tr>
</tbody>
</table>
### Input

<table>
<thead>
<tr>
<th>Learnable Parameters</th>
<th>A1, ..., An</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as params</td>
</tr>
<tr>
<td>Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a dlarray.</td>
<td>Table with the same data types, variables and ordering as params. A1, ..., An must have a Value variable consisting of cell arrays that contain the additional input argument for the function fun to apply to each learnable parameter.</td>
</tr>
</tbody>
</table>

### Output Arguments

**dlnet — Updated network**

dlnetwork object

Network, returned as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object.

**params — Updated network learnable parameters**

dlarray | numeric array | cell array | structure | table

Updated network learnable parameters, returned as a dlarray, a numeric array, a cell array, a structure, or a table with a Value variable containing the updated learnable parameters of the network.

**X1, ..., Xm — Additional output arguments**

dlarray | numeric array | cell array | structure | table

Additional output arguments from the function fun, where fun is a function handle to a function that returns multiple outputs, returned as dlarray objects, numeric arrays, cell arrays, structures, or tables with a Value variable.
The exact form of $X_1, \ldots, X_m$ depends on the input network or learnable parameters. The following table shows the returned format of $X_1, \ldots, X_m$ for possible inputs to `dlupdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable parameters</th>
<th>$X_1, \ldots, X_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. $X_1,\ldots,X_m$ has a <code>Value</code> variable consisting of cell arrays that contain the additional output arguments of the function <code>fun</code> applied to each learnable parameter.</td>
</tr>
<tr>
<td><code>params</code></td>
<td><code>dlarray</code></td>
<td><code>dlarray</code> with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Table with <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data types, variables, and ordering as <code>params</code>. $X_1,\ldots,X_m$ has a <code>Value</code> variable consisting of cell arrays that contain the additional output argument of the function <code>fun</code> applied to each learnable parameter.</td>
</tr>
</tbody>
</table>
See Also
adamupdate | dlarray | dlfeval | dlgradient | dlnetwork | rmspropupdate | sgdupdate

Topics
“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”
“Sequence-to-Sequence Translation Using Attention”
“Sequence-to-Sequence Classification Using 1-D Convolutions”

Introduced in R2019b
rmspropupdate

Update parameters using root mean squared propagation (RMSProp)

Syntax

[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad)
[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad)
[___] = rmspropupdate(__learnRate,sqGradDecay,epsilon)

Description

Update the network learnable parameters in a custom training loop using the root mean squared propagation (RMSProp) algorithm.

Note This function applies the RMSProp optimization algorithm to update network parameters in custom training loops that use networks defined as dlnetwork objects or model functions. If you want to train a network defined as a Layer array or as a LayerGraph, use the following functions:

• Create a TrainingOptionsRMSProp object using the trainingOptions function.
• Use the TrainingOptionsRMSProp object with the trainNetwork function.

[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad)
updates the learnable parameters of the network dlnet using the RMSProp algorithm. Use this syntax in a training loop to iteratively update a network defined as a dlnetwork object.

[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad)
updates the learnable parameters in params using the RMSProp algorithm. Use this syntax in a training loop to iteratively update the learnable parameters of a network defined using functions.
[___] = rmspropupdate(___ learnRate, sqGradDecay, epsilon) also specifies values to use for the global learning rate, square gradient decay, and small constant epsilon, in addition to the input arguments in previous syntaxes.

Examples

Update Learnable Parameters Using rmspropupdate

Perform a single root mean squared propagation update step with a global learning rate of 0.05 and squared gradient decay factor of 0.95.

Create the parameters and parameter gradients as numeric arrays.

params = rand(3,3,4);
grad = ones(3,3,4);

Initialize the average squared gradient for the first iteration.

averageSqGrad = [];

Specify custom values for the global learning rate and squared gradient decay factor.

learnRate = 0.05;
sqGradDecay = 0.95;

Update the learnable parameters using rmspropupdate.

[params,averageSqGrad] = rmspropupdate(params, grad, averageSqGrad, learnRate, sqGradDecay);

Train a Network Using rmspropupdate

Use rmspropupdate to train a network using the root-mean-square propagation (RMSProp) algorithm.

Load Training Data

Load the digits training data.
Define the Network

Define the network architecture and specify the average image using the 'Mean' option in the image input layer.

```matlab
layers = [imageInputLayer([28 28 1], 'Name','input','Mean',mean(XTrain,4))
    convolution2dLayer(5,20,'Name','conv1')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')];
```

Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph);
```

Define the Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding labels `Y` and returns the loss and the gradients of the loss with respect to the learnable parameters in `dlnet`.

Specify Training Options

Specify the options to use during training.

```matlab
miniBatchSize = 128;
numEpochs = 20;
numObservations = numel(YTrain);
umIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```
Initialize squared average gradients.

```
averageSqGrad = []; 
```

Initialize the training progress plot.

```
plots = "training-progress";
if plots == "training-progress"
    iteration = 1;
    figure
    lineLossTrain = animatedline;
    xlabel("Total Iterations")
    ylabel("Loss")
end
```

**Train the Network**

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters using the `rmspropupdate` function. At the end of each epoch, display the training progress.

```
for epoch = 1:numEpochs 
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,:idx);
    YTrain = YTrain(idx);

    for i = 1:numIterationsPerEpoch
        % Read mini-batch of data and convert the labels to dummy variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,:idx);

        Y = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y(c,YTrain(idx)==classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(X),'SSCB');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

end
```

end

% Evaluate the model gradients and loss using dlfeval and the
% modelGradients function.
[grad,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);

% Update the network parameters using the RMSProp optimizer.
[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad);

% Display the training progress.
if plots == "training-progress"
    addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
title("Loss During Training: Epoch - " + epoch + "; Iteration - " + i)
drawnow
    iteration = iteration + 1;
end
end
Test the Network

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.

\[ [\text{XTest}, \text{YTest}] = \text{digitTest4DArrayData}; \]

Convert the data to a dlarray object with dimension format 'SSCB'. For GPU prediction, also convert the data to gpuArray.

\[ \text{dlXTest} = \text{dlarray}(\text{XTest}, 'SSCB'); \]
\[ \text{if} \ (\text{executionEnvironment} == \ "auto" \ & \ & \text{canUseGPU}) \ || \ \text{executionEnvironment} == \ "gpu" \]
\[ \text{dlXTest} = \text{gpuArray}(\text{dlXTest}); \]
\[ \text{end} \]
To classify images using a dlnetwork object, use the predict function and find the classes with the highest scores.

```matlab
dlYPred = predict(dlnet,dlXTest);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```matlab
accuracy = mean(YPred==YTest)
```

```matlab
accuracy = 0.9752
```

**Model Gradients Function**

The modelGradients function takes a dlnetwork object dlnet, a mini-batch of input data dlX with corresponding labels Y and returns the loss and the gradients of the loss with respect to the learnable parameters in dlnet. To compute the gradients automatically, use the dlgradient function.

```matlab
function [gradients,loss] = modelGradients(dlnet,dlX,Y)
    dlYPred = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);

    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);
end
```

**Input Arguments**

dlnet — Network
dlnetwork object

Network, specified as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object. dlnet.Learnables is a table with three variables:

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a dlarray.
The input argument `grad` must be a table of the same form as `dlnet.Learnables`.

**params — Network learnable parameters**

dlarray | numeric array | cell array | structure | table

Network learnable parameters, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

If you specify `params` as a table, it must contain the following three variables.

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a `dlarray`.

You can specify `params` as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be `dlarray` or numeric values of data type `double` or `single`.

The input argument `grad` must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as `params`.

Data Types: `single` | `double` | `struct` | `table` | `cell`

**grad — Gradients of loss**

dlarray | numeric array | cell array | structure | table

Gradients of the loss, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `grad` depends on the input network or learnable parameters. The following table shows the required format for `grad` for possible inputs to `rmspropupdate`.
<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th><strong>Learnable Parameters</strong></th>
<th><strong>Gradients</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. <code>grad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the gradient of each learnable parameter.</td>
</tr>
<tr>
<td><code>params</code></td>
<td><code>dlarray</code></td>
<td><code>dlarray</code> with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Table with <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data types, variables, and ordering as <code>params</code>. <code>grad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the gradient of each learnable parameter.</td>
</tr>
</tbody>
</table>

You can obtain `grad` from a call to `dlfeval` that evaluates a function that contains a call to `dlgradient`. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox”.

**averageSqGrad — Moving average of squared parameter gradients**

`[]` | `dlarray` | numeric array | cell array | structure | table

Moving average of squared parameter gradients, specified as an empty array, a `dlarray`, a numeric array, a cell array, a structure, or a table.
The exact form of `averageSqGrad` depends on the input network or learnable parameters. The following table shows the required format for `averageSqGrad` for possible inputs to `rmspropupdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>Average Squared Gradients</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. <code>averageSqGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.</td>
</tr>
<tr>
<td><code>params</code></td>
<td><code>dlarray</code></td>
<td><code>dlarray</code> with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Table with <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable must consist of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data types, variables, and ordering as <code>params</code>. <code>averageSqGrad</code> must have a <code>Value</code> variable consisting of cell arrays that contain the average squared gradient of each learnable parameter.</td>
</tr>
</tbody>
</table>
If you specify `averageSqGrad` as an empty array, the function assumes no previous gradients and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the `averageSqGrad` output of a previous call to `rmspropupdate` as the `averageSqGrad` input.

**learnRate — Global learning rate**

0.001 (default) | positive scalar

Global learning rate, specified as a positive scalar. The default value of `learnRate` is 0.001.

If you specify the network parameters as a `dlnetwork`, the learning rate for each parameter is the global learning rate multiplied by the corresponding learning rate factor property defined in the network layers.

**sqGradDecay — Squared gradient decay factor**

0.999 (default) | positive scalar between 0 and 1.

Squared gradient decay factor, specified as a positive scalar between 0 and 1. The default value of `sqGradDecay` is 0.999.

**epsilon — Small constant**

1e-8 (default) | positive scalar

Small constant for preventing divide-by-zero errors, specified as a positive scalar. The default value of `epsilon` is 1e-8.

### Output Arguments

**dlnet — Updated network**

dlnetwork object

Network, returned as a `dlnetwork` object.

The function updates the `dlnet.Learnables` property of the `dlnetwork` object.

**params — Updated network learnable parameters**

dlarray | numeric array | cell array | structure | table
Updated network learnable parameters, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table with a `Value` variable containing the updated learnable parameters of the network.

**averageSqGrad — Updated moving average of squared parameter gradients**

dlarray | numeric array | cell array | structure | table

Updated moving average of squared parameter gradients, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table.

**More About**

**RMSProp**

The function uses the root mean squared propagation algorithm to update the learnable parameters. For more information, see the definition of the RMSProp algorithm under “Stochastic Gradient Descent” on page 1-691 on the `trainingOptions` reference page.

**See Also**

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `dlupdate` | `forward` | `sgdmupdate`

**Topics**

“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”

**Introduced in R2019b**
sgdmupdate

Update parameters using stochastic gradient descent with momentum (SGDM)

Syntax

[dlnet,vel] = sgdmupdate(dlnet,grad,vel)
[params,vel] = sgdmupdate(params,grad,vel)
[___] = sgdmupdate(___ learnRate,momentum)

Description

Update the network learnable parameters in a custom training loop using the stochastic gradient descent with momentum (SGDM) algorithm.

Note

This function applies the SGDM optimization algorithm to update network parameters in custom training loops that use networks defined as dlnetwork objects or model functions. If you want to train a network defined as a Layer array or as a LayerGraph, use the following functions:

• Create a TrainingOptionsSGDM object using the trainingOptions function.
• Use the TrainingOptionsSGDM object with the trainNetwork function.

[dlnet,vel] = sgdmupdate(dlnet,grad,vel) updates the learnable parameters of the network dlnet using the SGDM algorithm. Use this syntax in a training loop to iteratively update a network defined as a dlnetwork object.

[params,vel] = sgdmupdate(params,grad,vel) updates the learnable parameters in params using the SGDM algorithm. Use this syntax in a training loop to iteratively update the learnable parameters of a network defined using functions.

[___] = sgdmupdate(___ learnRate,momentum) also specifies values to use for the global learning rate and momentum, in addition to the input arguments in previous syntaxes.
Examples

**Update Learnable Parameters Using `sgdmupdate`**

Perform a single SGDM update step with a global learning rate of $0.05$ and momentum of $0.95$.

Create the parameters and parameter gradients as numeric arrays.

```matlab
params = rand(3,3,4);
grad = ones(3,3,4);
```

Initialize the parameter velocities for the first iteration.

```matlab
vel = [];
```

Specify custom values for the global learning rate and momentum.

```matlab
learnRate = 0.05;
momentum = 0.95;
```

Update the learnable parameters using `sgdmupdate`.

```matlab
[params,vel] = sgdmupdate(params,grad,vel,learnRate,momentum);
```

---

**Train a Network Using `sgdmupdate`**

Use `sgdmupdate` to train a network using the stochastic gradient decent with momentum (SGDM) algorithm.

**Load Training Data**

Load the digits training data.

```matlab
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
umClasses = numel(classes);
```

**Define the Network**

Define the network architecture and specify the average image using the 'Mean' option in the image input layer.
layers = [
    imageInputLayer([28 28 1], 'Name','input','Mean',mean(XTrain,4)),
    convolution2dLayer(5,20,'Name','conv1')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')];
lgraph = layerGraph(layers);

Create a dlnetwork object from the layer graph.

dlnet = dlnetwork(lgraph);

**Define the Model Gradients Function**

Create the function modelGradients, listed at the end of the example, that takes a dlnetwork object dlnet, a mini-batch of input data dlX with corresponding labels Y and returns the loss and the gradients of the loss with respect to the learnable parameters in dlnet.

**Specify Training Options**

Specify the options to use during training.

miniBatchSize = 128;
numEpochs = 20;
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

executionEnvironment = "auto";

Initialize the velocity parameter.

vel = [];

Initialize the training progress plot.

plots = "training-progress";
if plots == "training-progress"
    iteration = 1;
Train the Network

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. Update the network parameters using the `sgdmupdate` function. At the end of each epoch, display the training progress.

```matlab
for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,:,idx);
    YTrain = YTrain(idx);

    for i = 1:numIterationsPerEpoch
        % Read mini-batch of data and convert the labels to dummy variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:, :, :, idx);
        Y = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y(c, YTrain(idx) == classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(X), 'SSCB');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients and loss using dlfeval and the modelGradients function.
        [grad, loss] = dlfeval(@modelGradients, dlnet, dlX, Y);

        % Update the network parameters using the SGDM optimizer.
        [dlnet, vel] = sgdmupdate(dlnet, grad, vel);
    end
end
```
% Display the training progress.
if plots == "training-progress"
    addpoints(lineLossTrain, iteration, double(gather(extractdata(loss))))
    title("Loss During Training: Epoch - " + epoch + "; Iteration - " + i)
    drawnow
end
iteration = iteration + 1;
end

Test the Network

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels.
[XTest, YTest] = digitTest4DArrayData;

Convert the data to a dlarray object with dimension format 'SSCB'. For GPU prediction, also convert the data to gpuArray.

dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end

To classify images using a dlnetwork object, use the predict function and find the classes with the highest scores.

dlYPred = predict(dlnet,dlXTest);
 [~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);

Evaluate the classification accuracy.

accuracy = mean(YPred==YTest)
accuracy = 0.9914

**Model Gradients Function**

The modelGradients function takes a dlnetwork object dlnet, a mini-batch of input data dlX with corresponding labels Y and returns the loss and the gradients of the loss with respect to the learnable parameters in dlnet. To compute the gradients automatically, use the dlgradient function.

function [gradients,loss] = modelGradients(dlnet,dlX,Y)
    dlYPred = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);

    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);
Input Arguments

dlnet — Network
dlnetwork object

Network, specified as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object. dlnet.Learnables is a table with three variables:

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a dlarray.

The input argument grad must be a table of the same form as dlnet.Learnables.

params — Network learnable parameters
dlarray | numeric array | cell array | structure | table

Network learnable parameters, specified as a dlarray, a numeric array, a cell array, a structure, or a table.

If you specify params as a table, it must contain the following three variables.

- **Layer** — Layer name, specified as a string scalar.
- **Parameter** — Parameter name, specified as a string scalar.
- **Value** — Value of parameter, specified as a cell array containing a dlarray.

You can specify params as a container of learnable parameters for your network using a cell array, structure, or table, or nested cell arrays or structures. The learnable parameters inside the cell array, structure, or table must be dlarray or numeric values of data type double or single.

The input argument grad must be provided with exactly the same data type, ordering, and fields (for structures) or variables (for tables) as params.

Data Types: single | double | struct | table | cell
**grad — Gradients of the loss**
dlarray | numeric array | cell array | structure | table

Gradients of the loss, specified as a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `grad` depends on the input network or learnable parameters. The following table shows the required format for `grad` for possible inputs to `sgdmupdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>Gradients</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlnet</code></td>
<td><code>Table dlnet.Learnables containing Layer, Parameter, and Value variables. The Value variable consists of cell arrays that contain each learnable parameter as a dlarray.</code></td>
<td><code>Table with the same data type, variables, and ordering as dlnet.Learnables. grad must have a Value variable consisting of cell arrays that contain the gradient of each learnable parameter.</code></td>
</tr>
<tr>
<td><code>params</code></td>
<td><code>dlarray</code></td>
<td><code>dlarray with the same data type and ordering as params</code></td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as params</td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as params</td>
</tr>
<tr>
<td></td>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as params</td>
</tr>
<tr>
<td></td>
<td><code>Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a dlarray.</code></td>
<td><code>Table with the same data types, variables, and ordering as params. grad must have a Value variable consisting of cell arrays that contain the gradient of each learnable parameter.</code></td>
</tr>
</tbody>
</table>
You can obtain grad from a call to dlfeval that evaluates a function that contains a call to dlgradient. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox”.

**vel — Parameter velocities**

`[] | dlarray | numeric array | cell array | structure | table`

Parameter velocities, specified as an empty array, a `dlarray`, a numeric array, a cell array, a structure, or a table.

The exact form of `vel` depends on the input network or learnable parameters. The following table shows the required format for `vel` for possible inputs to `sgdmpdate`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>Velocities</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlnet</td>
<td>Table <code>dlnet.Learnables</code> containing <code>Layer</code>, <code>Parameter</code>, and <code>Value</code> variables. The <code>Value</code> variable consists of cell arrays that contain each learnable parameter as a <code>dlarray</code>.</td>
<td>Table with the same data type, variables, and ordering as <code>dlnet.Learnables</code>. <code>vel</code> must have a <code>Value</code> variable consisting of cell arrays that contain the velocity of each learnable parameter.</td>
</tr>
<tr>
<td>params</td>
<td><code>dlarray</code></td>
<td><code>dlarray</code> with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Numeric array</td>
<td>Numeric array with the same data type and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Cell array</td>
<td>Cell array with the same data types, structure, and ordering as <code>params</code></td>
</tr>
<tr>
<td></td>
<td>Structure</td>
<td>Structure with the same data types, fields, and ordering as <code>params</code></td>
</tr>
</tbody>
</table>
Learnable Parameters

<table>
<thead>
<tr>
<th>Input</th>
<th>Learnable Parameters</th>
<th>Velocities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Table with Layer, Parameter, and Value variables. The Value variable must consist of cell arrays that contain each learnable parameter as a dlarray.</td>
<td>Table with the same data types, variables, and ordering as params.vel must have a Value variable consisting of cell arrays that contain the velocity of each learnable parameter.</td>
</tr>
</tbody>
</table>

If you specify vel as an empty array, the function assumes no previous velocities and runs in the same way as for the first update in a series of iterations. To update the learnable parameters iteratively, use the vel output of a previous call to sgdmupdate as the vel input.

**learnRate — Global learning rate**

0.01 (default) | positive scalar

Learning rate, specified as a positive scalar. The default value of learnRate is 0.01.

If you specify the network parameters as a dlnetwork object, the learning rate for each parameter is the global learning rate multiplied by the corresponding learning rate factor property defined in the network layers.

**momentum — Momentum**

0.9 (default) | positive scalar between 0 and 1

Momentum, specified as a positive scalar between 0 and 1. The default value of momentum is 0.9.

**Output Arguments**

**dlnet — Updated network**

dlnetwork object

Network, returned as a dlnetwork object.

The function updates the dlnet.Learnables property of the dlnetwork object.

**params — Updated network learnable parameters**

dlarray | numeric array | cell array | structure | table
Updated network learnable parameters, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table with a `Value` variable containing the updated learnable parameters of the network.

`vel` — Updated parameter velocities

dlarray | numeric array | cell array | structure | table

Updated parameter velocities, returned as a `dlarray`, a numeric array, a cell array, a structure, or a table.

More About

Stochastic Gradient Descent with Momentum

The function uses the stochastic gradient descent with momentum algorithm to update the learnable parameters. For more information, see the definition of the stochastic gradient descent with momentum algorithm under “Stochastic Gradient Descent” on page 1-691 on the `trainingOptions` reference page.

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `dlupdate` | `forward` | `rmspropupdate`

Topics

“Define Custom Training Loops, Loss Functions, and Networks”
“Specify Training Options in Custom Training Loop”
“Train Network Using Custom Training Loop”

Introduced in R2019b
canUseGPU

Verify supported GPU is available for computation

Syntax

tf = canUseGPU()

Description

tf = canUseGPU() returns a logical value indicating if there is a GPU available for computation. The function returns logical 1 (true) if there is a supported GPU available, a recent GPU driver is present, and Parallel Computing Toolbox is installed and licensed for use. Otherwise, the function returns logical 0 (false).

Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on supported GPUs, see “GPU Support by Release” (Parallel Computing Toolbox).

Examples

Check GPU Availability to Solve a System of Linear Equations

Solve a system of linear equations. Perform the calculation using a GPU if one is available; otherwise, use the CPU.

Create the data on the CPU.

N = 1000;
A = rand(N);
b = rand(N,1);

Transfer the matrix A to the GPU, if there is one available

tf = canUseGPU()
tf =
  1

if tf
    A = gpuArray(A);
end

Solve the system of linear equations.

x = A\b;

Since A exists on the GPU, the solution is calculated on the GPU.

See Also
gpuDevice | gpuDeviceCount

Topics
“Specify Training Options in Custom Training Loop”

Introduced in R2019b