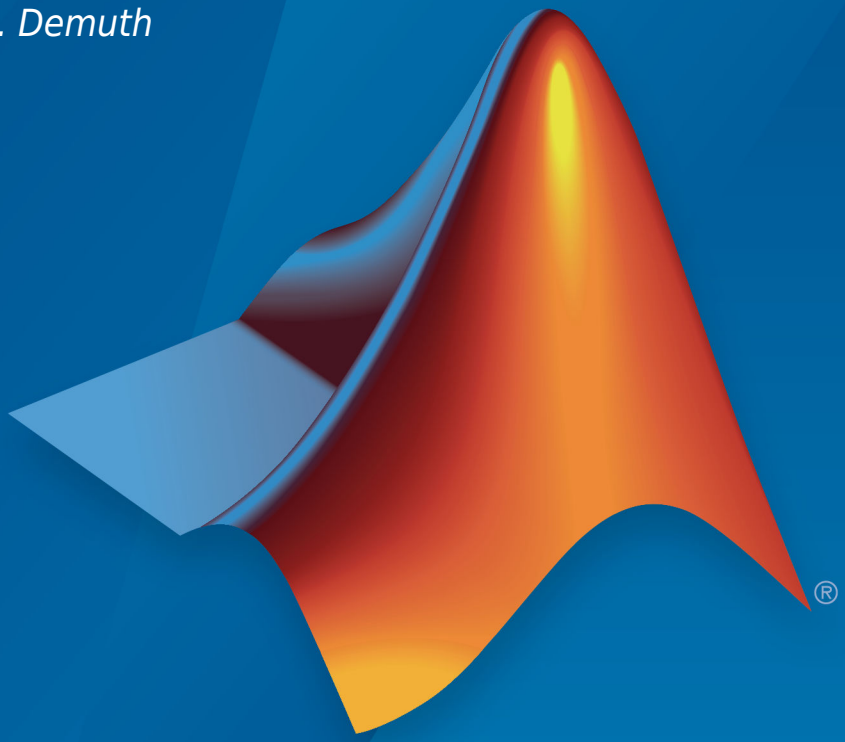
Neural Network Toolbox™

Reference

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth



MATLAB®

R2018**a**

MathWorks®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Neural Network Toolbox ™ Reference

Revision History

June 1992	First printing	
April 1993	Second printing	
January 1997	Third printing	
July 1997	Fourth printing	
January 1998	Fifth printing	Revised for Version 3 (Release 11)
September 2000	Sixth printing	Revised for Version 4 (Release 12)
June 2001	Seventh printing	Minor revisions (Release 12.1)
July 2002	Online only	Minor revisions (Release 13)
January 2003	Online only	Minor revisions (Release 13SP1)
June 2004	Online only	Revised for Version 4.0.3 (Release 14)
October 2004	Online only	Revised for Version 4.0.4 (Release 14SP1)
October 2004	Eighth printing	Revised for Version 4.0.4
March 2005	Online only	Revised for Version 4.0.5 (Release 14SP2)
March 2006	Online only	Revised for Version 5.0 (Release 2006a)
September 2006	Ninth printing	Minor revisions (Release 2006b)
March 2007	Online only	Minor revisions (Release 2007a)
September 2007	Online only	Revised for Version 5.1 (Release 2007b)
March 2008	Online only	Revised for Version 6.0 (Release 2008a)
October 2008	Online only	Revised for Version 6.0.1 (Release 2008b)
March 2009	Online only	Revised for Version 6.0.2 (Release 2009a)
September 2009	Online only	Revised for Version 6.0.3 (Release 2009b)
March 2010	Online only	Revised for Version 6.0.4 (Release 2010a)
September 2010	Online only	Revised for Version 7.0 (Release 2010b)
April 2011	Online only	Revised for Version 7.0.1 (Release 2011a)
September 2011	Online only	Revised for Version 7.0.2 (Release 2011b)
March 2012	Online only	Revised for Version 7.0.3 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.0.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.1 (Release 2013b)
March 2014	Online only	Revised for Version 8.2 (Release 2014a)
October 2014	Online only	Revised for Version 8.2.1 (Release 2014b)
March 2015	Online only	Revised for Version 8.3 (Release 2015a)
September 2015	Online only	Revised for Version 8.4 (Release 2015b)
March 2016	Online only	Revised for Version 9.0 (Release 2016a)
September 2016	Online only	Revised for Version 9.1 (Release 2016b)
March 2017	Online only	Revised for Version 10.0 (Release 2017a)
September 2017	Online only	Revised for Version 11.0 (Release 2017b)
March 2018	Online only	Revised for Version 11.1 (Release 2018a)

Contents

Functions — Alphabetical List

adapt

Adapt neural network to data as it is simulated

Syntax

```
[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)
```

To Get Help

Type `help network/adapt`.

Description

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

`[net,Y,E,Pf,Af,tr] = adapt(net,P,T,Pi,Ai)` takes

`net`	Network
`P`	Network inputs
`T`	Network targets (default = zeros)
`Pi`	Initial input delay conditions (default = zeros)
`Ai`	Initial layer delay conditions (default = zeros)

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

`net`	Updated network
`Y`	Network outputs
`E`	Network errors
`Pf`	Final input delay conditions

`Af`	Final layer delay conditions
`tr`	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,

`P`	`Ni`-by-`TS` cell array	Each element `P{i,ts}` is an `Ri`-by-`Q` matrix.
`T`	`Nt`-by-`TS` cell array	Each element `T{i,ts}` is a `Vi`-by-`Q` matrix.
`Pi`	`Ni`-by-`ID` cell array	Each element `Pi{i,k}` is an `Ri`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.
`Y`	`No`-by-`TS` cell array	Each element `Y{i,ts}` is a `Ui`-by-`Q` matrix.
`E`	`No`-by-`TS` cell array	Each element `E{i,ts}` is a `Ui`-by-`Q` matrix.
`Pf`	`Ni`-by-`ID` cell array	Each element `Pf{i,k}` is an `Ri`-by-`Q` matrix.
`Af`	`Nl`-by-`LD` cell array	Each element `Af{i,k}` is an `Si`-by-`Q` matrix.

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.

```
p1 = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t1 = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
```

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.

```
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.)

```
[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).

```
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.

```
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

```
[net,ar,Ac] = adapt(net,Pd,T,Ai)
```

Description

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

`[net,ar,Ac] = adapt(net,Pd,T,Ai)` takes these arguments,

`net`	Neural network
`Pd`	Delayed processed input states and inputs
`T`	Targets
`Ai`	Initial layer delay states

and returns

`net`	Neural network after adaption
`ar`	Adaption record
`Ac`	Combined initial layer states and layer outputs

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.

```
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)
```

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,

`net`	Neural network
`n`	Number of delays

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.

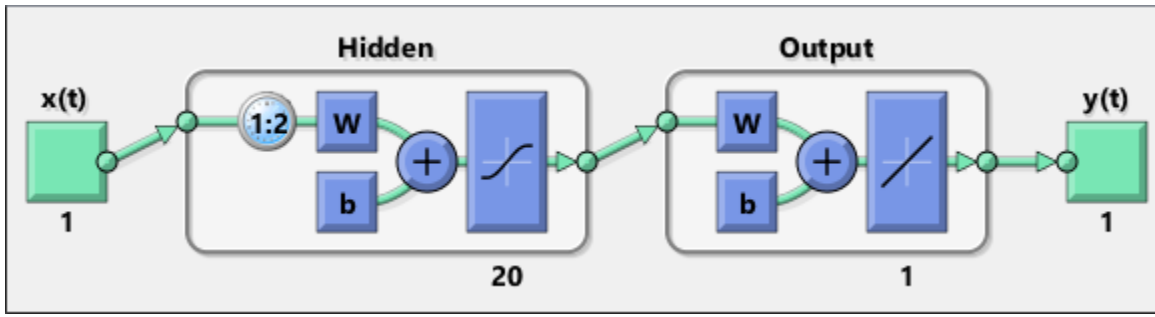
Examples

Time Delay Network

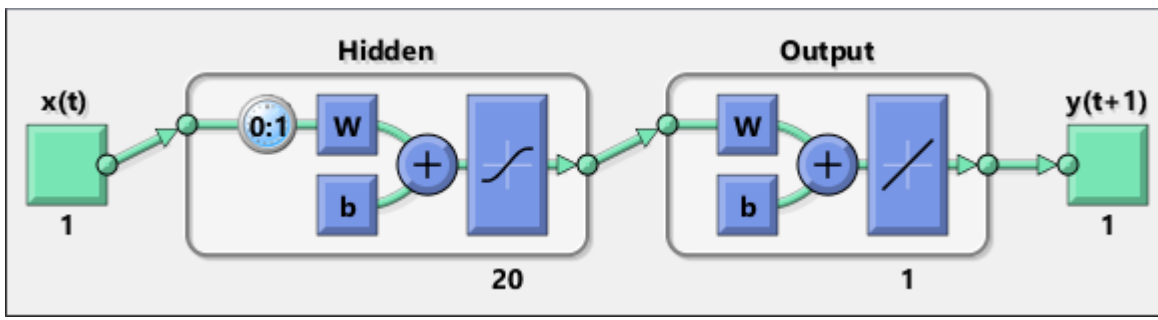
This example creates, trains, and simulates a time delay network in its original form, on an input time series `X` and target series `T`. Then the delay is removed and later added back. The first and third outputs will be identical, while the second result will include a new prediction for the following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

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

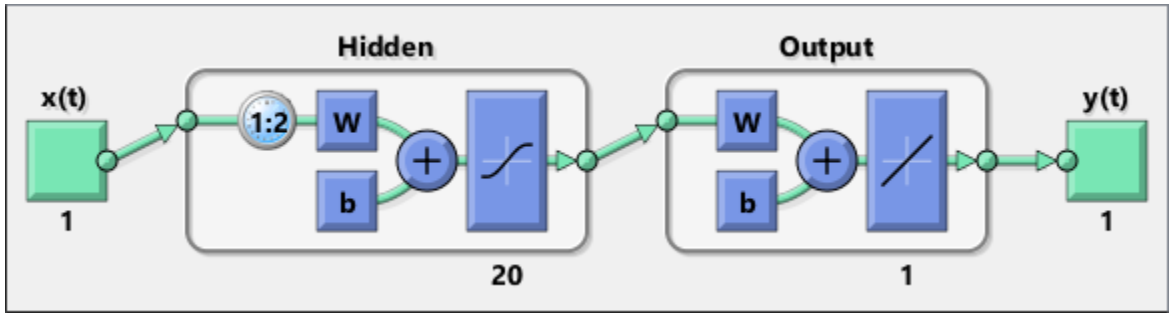
```
y1 = net1(Xs,Xi);
view(net1)
```




```
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 `Pi` and `Pj` from a set of `S` vectors is

```
Dij = max(abs(Pi-Pj))
```

See Also

`dist` | `linkdist` | `mandist` | `sim`

Introduced before R2006a

bttderiv

Backpropagation through time derivative function

Syntax

```
bttderiv('dperf_dwb',net,X,T,Xi,Ai,EW)
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.

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

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

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).

`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 = bttderiv('dperf_dwb',net,x,t)
jwb = bttderiv('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,

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

and returns a new cascade-forward neural network.

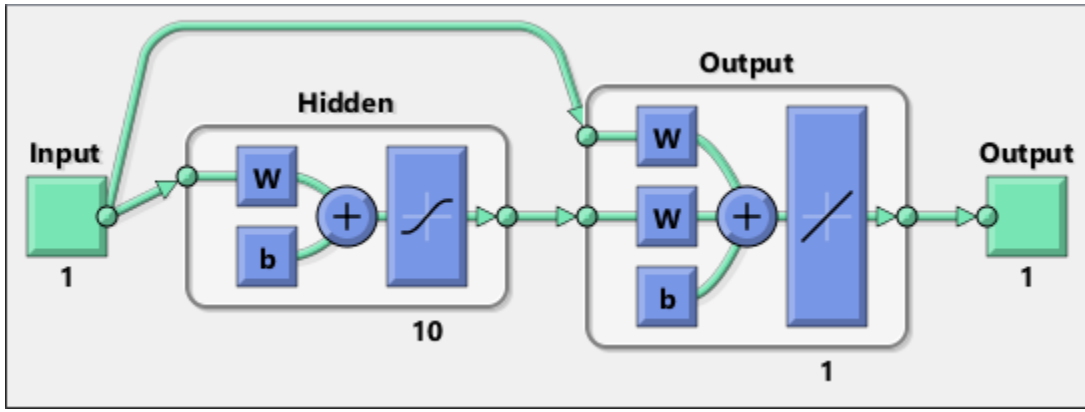
Examples

Create and Train a Cascade Network

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

```
[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 Neural Networks”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b

catelements

Concatenate neural network data elements

Syntax

```
catelements(x1,x2,...,xn)
[x1; x2; ... xn]
```

Description

`catelements(x1,x2,...,xn)` 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 `[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.

Examples

This code concatenates the elements of two matrix data values.

```
x1 = [1 2 3; 4 7 4]
x2 = [5 8 2; 4 7 6; 2 9 1]
y = catelements(x1,x2)
```

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

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

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

```
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.

```
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.

```
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.

```
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] = simplenarx_dataset;
net = narxnet(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
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`	Matrix of `N1` (column) vectors
`A2`	Matrix of `N2` (column) vectors

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)
```

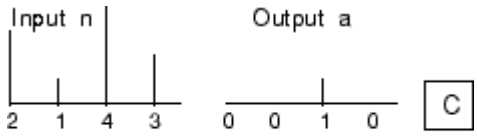
```
a3 =

     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



Compet Transfer Function

Syntax

```
A = compet(N,FP)
info = compet('code')
```

Description

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

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

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

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.

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

`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.

```
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,

`numClasses`	Number of classes to classify inputs (default = 5)
`kohonenLR`	Learning rate for Kohonen weights (default = 0.01)
`conscienceLR`	Learning rate for conscience bias (default = 0.001)

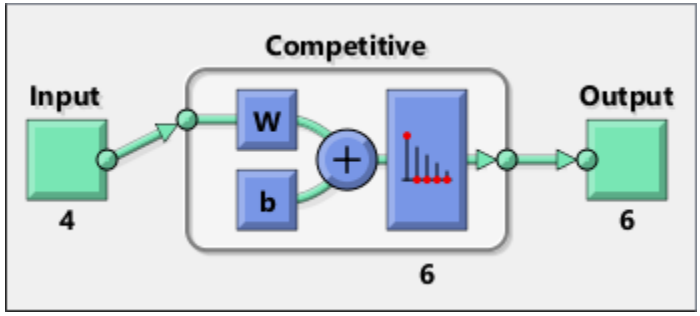
and returns a competitive layer with `numClasses` neurons.

Examples

Create and Train a Competitive Layer

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

```
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 = con2seq(b)
S = con2seq(b,TS)
```

Description

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

`con2seq` and `seq2con` allow concurrent vectors to be converted to sequential vectors, and back again.

`S = con2seq(b)` takes one input,

`b`	`R`-by-`TS` matrix

and returns one output,

`S`	`1`-by-`TS` cell array of `R`-by-`1` vectors

`S = con2seq(b,TS)` can also convert multiple batches,

`b`	`N`-by-`1` cell array of matrices with `M*TS` columns
`TS`	Time steps

and returns

`S`	`N`-by-`TS` cell array of matrices with `M` columns

Examples

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

```
p1 = [1 4 2]
p2 = 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 = con2seq(p1,2)
```

See Also

`concur` | `seq2con`

Introduced before R2006a

concur

Create concurrent bias vectors

Syntax

```
concur(B,Q)
```

Description

`concur(B,Q)`

`B`	`S`-by-`1` bias vector (or an `Nl`-by-`1` cell array of vectors)
`Q`	Concurrent size

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.

```
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:

```
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 = netsum(z1,z2,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

```
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).

```
[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:

`targets`	`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.
`outputs`	`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.

and returns these values:

`c`	Confusion value = fraction of samples misclassified
`cm`	`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`
`ind`	`S`-by-`S` cell array, where `ind{i,j}` contains the indices of samples with the `i`th target class, but `j`th output class

`per`	`S`-by-4 matrix, where each row summarizes four percentages associated with the `i`th class:
	`per(i,1) false negative rate`
	`= (false negatives)/(all output negatives)`
	`per(i,2) false positive rate`
	`= (false positives)/(all output positives)`
	`per(i,3) true positive rate`
	`= (true positives)/(all output positives)`
	`per(i,4) true negative rate`
	`= (true negatives)/(all output negatives)`

`[c,cm,ind,per] = confusion(TARGETS,OUTPUTS)` takes these values:

`targets`	`1`-by-`Q` vector of 1/0 values representing membership
`outputs`	`S`-by-`Q` matrix, of value in `[0,1]` interval, where values greater than or equal to `0.5` indicate class membership

and returns these values:

`c`	Confusion value = fraction of samples misclassified
`cm`	2-by-2 confusion matrix
`ind`	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`
`per`	2-by-4 matrix where each `i`th row represents the percentage of false negatives, false positives, true positives, and true negatives for the class and out-of-class

Examples

```
[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 = convwf(W,P)
dim = convwf('size',S,R,FP)
dw = convwf('dw',W,P,Z,FP)
info = convwf('code')
```

Description

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

`Z = convwf(W,P)` returns the convolution of a weight matrix `W` and an input `P`.

`dim = convwf('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size.

`dw = convwf('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

`info = convwf('code')` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'fullderiv'`	Reduced derivative = 2, full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'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 = rand(4,1);
P = rand(8,1);
Z = convwf(W,P)
```

Network Use

To change a network so an input weight uses `convwf`, set `net.inputWeights{i,j}.weightFcn` to `'convwf'`. For a layer weight, set `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 calculation performance on the whole dataset.

```
[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 = 0.0278
```

Set crossentropy as Performance Function

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

```
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

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 `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 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 single 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
`'none'` (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,

`net`	Neural network
`X`	Inputs, an `R`-by-`Q` matrix (or `N`-by-`TS` cell array of `Ri`-by-`Q` matrices)
`T`	Targets, an `S`-by-`Q` matrix (or `M`-by-`TS` cell array of `Si`-by-`Q` matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

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.

```
[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

`bttderiv` | `fpderiv` | `num2deriv` | `num5deriv` | `staticderiv`

Introduced in R2010b

dist

Euclidean distance weight function

Syntax

```
Z = dist(W,P,FP)
dim = dist('size',S,R,FP)
dw = dist('dw',W,P,Z,FP)
D = dist(pos)
info = dist('code')
```

Description

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

`Z = dist(W,P,FP)` takes these inputs,

`W`	S-by-R weight matrix
`P`	R-by-Q matrix of Q input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

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

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

`dw = dist('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 = dist(pos)` takes one argument,

`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:

`'deriv'`	Name of derivative function
`'fullderiv'`	Full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: 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 = 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.

```
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,

`delays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns a distributed delay neural network.

Examples

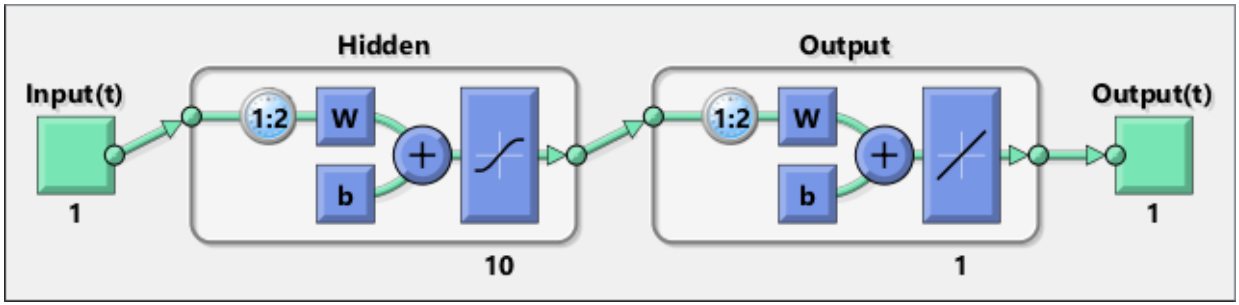
Distributed Delay Network

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 = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
```

```
perf =

    0.0323
```



See Also

narnet | narxnet | preparets | 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:

`Q`	Number of targets to divide up.
`trainRatio`	Ratio of targets for training. Default = `0.7`.
`valRatio`	Ratio of targets for validation. Default = `0.15`.
`testRatio`	Ratio of targets for testing. Default = `0.15`.

and returns

`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

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

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

Description

`[trainInd,valInd,testInd] = divideind(Q,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,

`Q`	Number of targets to divide up
`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

and returns

`trainInd`	Training indices (unchanged)
`valInd`	Validation indices (unchanged)
`testInd`	Test indices (unchanged)

Examples

```
[trainInd,valInd,testInd] = ...
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

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

Description

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

`Q`	Number of targets to divide up.
`trainRatio`	Ratio of vectors for training. Default = `0.7`.
`valRatio`	Ratio of vectors for validation. Default = `0.15`.
`testRatio`	Ratio of vectors for testing. Default = `0.15`.

and returns

`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

Examples

```
[trainInd,valInd,testInd] = 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,

`Q`	Number of targets to divide up.
`trainRatio`	Ratio of vectors for training. Default = `0.7`.
`valRatio`	Ratio of vectors for validation. Default = `0.15`.
`testRatio`	Ratio of vectors for testing. Default = `0.15`.

and returns

`trainInd`	Training indices
`valInd`	Validation indices
`testInd`	Test indices

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:

`Q`	Number of targets to divide up.

and returns

`trainInd`	Training indices equal to `1:Q`
`valInd`	Empty validation indices, `[]`
`testInd`	Empty test indices, `[]`

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 = dotprod(W,P,FP)
dim = dotprod('size',S,R,FP)
dw = dotprod('dw',W,P,Z,FP)
info = dotprod('code')
```

Description

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

`Z = dotprod(W,P,FP)` takes these inputs,

`W`	S-by-R weight matrix
`P`	R-by-Q matrix of Q input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

and returns the S-by-Q dot product of `W` and `P`.

`dim = dotprod('size',S,R,FP)` takes the layer dimension S, input dimension R, and function parameters, and returns the weight size [S-by-R].

`dw = dotprod('dw',W,P,Z,FP)` returns the derivative of `Z` with respect to `W`.

`info = dotprod('code')` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0

`'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 = rand(4,3);
P = rand(3,1);
Z = 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 = elliotsig(N)
```

Description

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

`A = 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 `[-inf inf]` 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 = elliotsig(n);
```

Plot the transfer function:

```
n = -5:0.01:5;
plot(n, elliotsig(n))
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 = 'elliotsig';
```

See Also

elliot2sig | logsig | tansig

Introduced in R2012b

elliot2sig

Elliot 2 symmetric sigmoid transfer function

Syntax

```
A = 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 `tansig`, but is not as smooth at the center.

`A = 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 `[-inf inf]` 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 = elliot2sig(n);
```

Plot the transfer function:

```
n = -5:0.01:5;
plot(n, elliot2sig(n))
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 `bttderiv`), 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,

`layerdelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

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

`errsurf(P,T,WV,BV,F)` takes these arguments,

`P`	1-by-`Q` matrix of input vectors
`T`	1-by-`Q` matrix of target vectors
`WV`	Row vector of values of `W`
`BV`	Row vector of values of `B`
`F`	Transfer function (string)

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`

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,

`x`	Neural network time series data
`ts`	Number of timesteps
`v`	Value

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.

```
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.

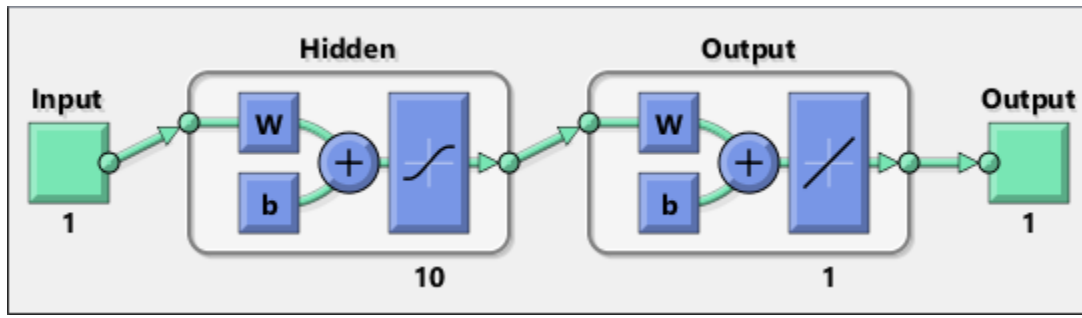
Examples

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] = fixunknowns(X)
[y,ps] = fixunknowns(X,FP)
Y = fixunknowns('apply',X,PS)
X = fixunknowns('reverse',Y,PS)
name = fixunknowns('name')
fp = fixunknowns('pdefaults')
pd = fixunknowns('pdesc')
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] = fixunknowns(X)` takes these inputs,

`X`	`N`-by-`Q` matrix

and returns

`Y`	`M`-by-`Q` matrix with `M - N` rows added
`PS`	Process settings that allow consistent processing of values

`[y,ps] = fixunknowns(X,FP)` takes an empty struct `FP` of parameters.

`Y = 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)
```

Definitions

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.

```
p2 =
   1  2  3  2  2
   1  0  1  1  0
   3  1 -1  2  4
```

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 = fixunknowns('apply',p1new,ps);
```

The function `fixunkowns` 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

```
fpderiv('dperf_dwb',net,X,T,Xi,Ai,EW)
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.

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

`net`	Neural network
`X`	Inputs, an `R`-by-`Q` matrix (or `N`-by-`TS` cell array of `Ri`-by-`Q` matrices)
`T`	Targets, an `S`-by-`Q` matrix (or `M`-by-`TS` cell array of `Si`-by-`Q` matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

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).

`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,

`net`	Neural network
`toMatrix`	True if result is to be in matrix form
`columnSample`	True if samples are to be represented as columns, false if rows
`cellTime`	True if time series are to be represented as a cell array, false if represented with a matrix

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.

```
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 = {rands(2,3);rands(2,3);rands(2,3)}
columnSamples = true;
cellTime = true;
[y,wasMatrix] = tonndata(x)
x2 = fromnndata(y,wasMatrix,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 = 2x2 cell array
    {[2]}    {[5]}
    {[8]}    {[6]}
```

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

```
ans = 3x4 cell array
    {[11]}    {[12]}    {[13]}    {[14]}
```

```
{[21]}    {[22]}    {[23]}    {[24]}
{[31]}    {[32]}    {[33]}    {[34]}
```

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])
```

```
ans = 2×3

    0.1000    0.2000    0.3000
    0.2000    0.2500    0.3000
```

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

```
ans = 2x2 cell array
    {[     1]}    {[0.6667]}
    {[0.6000]}    {[     2]}
```

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

```
ans = 3x4 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}    {[0.4000]}
```

```
    {[0.0500]}    {[0.1000]}    {[0.1500]}    {[0.2000]}
    {[0.0333]}    {[0.0667]}    {[0.1000]}    {[0.1333]}
```

See Also

gadd | gmultiply | gnegate | gsqrt | gsubtract

Introduced in R2010b

gensim

Generate Simulink block for neural network simulation

Syntax

```
gensim(net,st)
```

To Get Help

Type `help network/gensim`.

Description

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

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

`net`	Neural network
`st`	Sample time (default = 1)

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 neural network

Syntax

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

Description

`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 = bodyfatodeGen(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`.

```
genFunction(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.

```
genFunction(maglevNet,'maglevFcn','MatrixOnly','yes');
x1 = cell2mat(X(1,:)); % Convert each input to matrix
x2 = cell2mat(X(2,:));
xi1 = cell2mat(Xi(1,:)); % Convert each input state to matrix
xi2 = cell2mat(Xi(2,:));
[y3,xf1,xf2] = maglevFcn(x1,x2,xi1,xi2);
accuracy3 = max(abs(cell2mat(y)-y3))

x1Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 1
x2Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 2
xi1Type = coder.typeof(double(0),[1 2]); % Coder type of input 1 states
xi2Type = coder.typeof(double(0),[1 2]); % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen -args {x1Type x2Type xi1Type xi2Type}
[y4,xf1,xf2] = maglevNetCodeGen(x1,x2,xi1,xi2);
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 Neural Network 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 Trained Neural Network Functions”.

See Also

`gensim`

Topics

“Deploy Trained 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 = [1 2 3; 4 7 4]
y = getelements(x,[1 3])
```

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

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

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

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

```
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] = getsiminit(sysName,netName,net)
```

Description

`[xi,ai] = getsiminit(sysName,netName,net)` takes these arguments,

`sysName`	The name of the Simulink system containing the neural network block
`netName`	The name of the Simulink neural network block
`net`	The original neural network

and returns,

`xi`	Initial input delay states
`ai`	Initial layer delay states

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.

```
[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.

```
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.

```
[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`.)

```
[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:

```
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);
wb = 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 = 2x2 cell array
    {[ 1]}    {[6]}
    {[15]}    {[8]}
```

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

```
ans = 3x4 cell array
    {[10]}    {[20]}    {[30]}    {[ 40]}
```

```
{[20]}    {[40]}    {[60]}    {[ 80]}
{[30]}    {[60]}    {[90]}    {[120]}
```

See Also

gadd | gdivide | gnegate | gsqrt | gsubtract

Introduced in R2010b

gnegate

Generalized negation

Syntax

```
gnegate(x)
```

Description

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

Examples

Negate a Cell Array

This example shows how to negate a cell array:

```
x = {[1 2; 3 4],[1 -3; -5 2]};
y = gnegate(x);
y{1}, y{2}
```

```
ans = 2×2

    -1    -2
    -3    -4


ans = 2×2

    -1     3
     5    -2
```

See Also

gadd | gdivide | gmultiply | gsqrt | gsubtract

Introduced in R2010b

gpu2nndata

Reformat neural data back from GPU

Syntax

```
X = gpu2nndata(Y,Q)
X = gpu2nndata(Y)
X = 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 = 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 = 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 = 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{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 = nndata([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

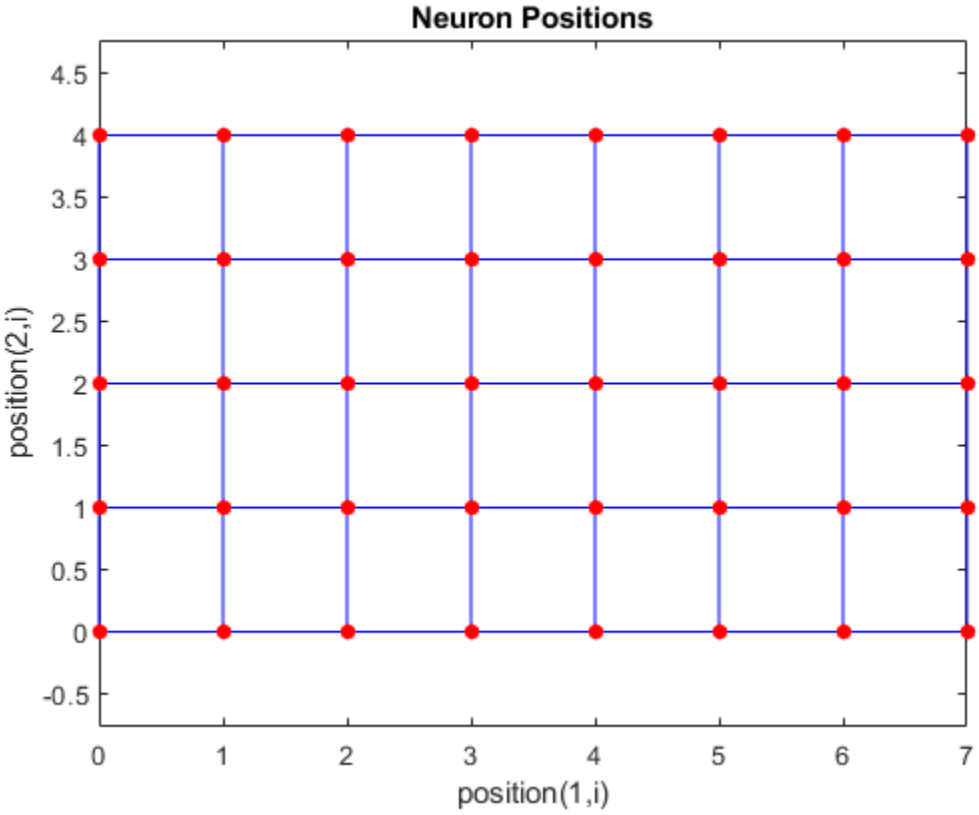
and returns an `N`-by-`S` matrix of `N` coordinate vectors where `N` is the number of dimensions and `S` is the product of `dimensions`.

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 array
    {[     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 = 2x2 cell array
    {[ 0]}    {[-1]}
    {[-2]}    {[ 2]}
```

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

```
ans = 3x4 cell array
    {[ -9]}    {[ -8]}    {[ -7]}    {[ -6]}
```

```
{[-19]}    {[-18]}    {[-17]}    {[-16]}
{[-29]}    {[-28]}    {[-27]}    {[-26]}
```

See Also

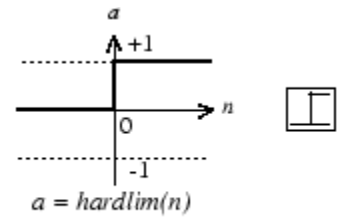
`gadd` | `gdivide` | `gmultiply` | `gnegate` | `gsqrt`

Introduced in R2010b

hardlim

Hard-limit transfer function

Graph and Symbol



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,

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, the `S`-by-`Q` Boolean matrix with 1s where `N` ≥ 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 `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.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

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

0 otherwise

See Also

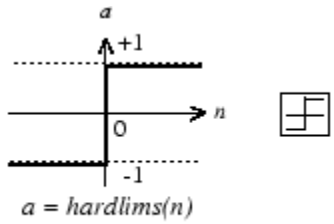
`hardlims` | `sim`

Introduced before R2006a

hardlims

Symmetric hard-limit transfer function

Graph and Symbol



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,

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

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.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

`hardlims(n)` = 1 if `n` ≥ 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:

`dimensions`	Row vector of dimension sizes

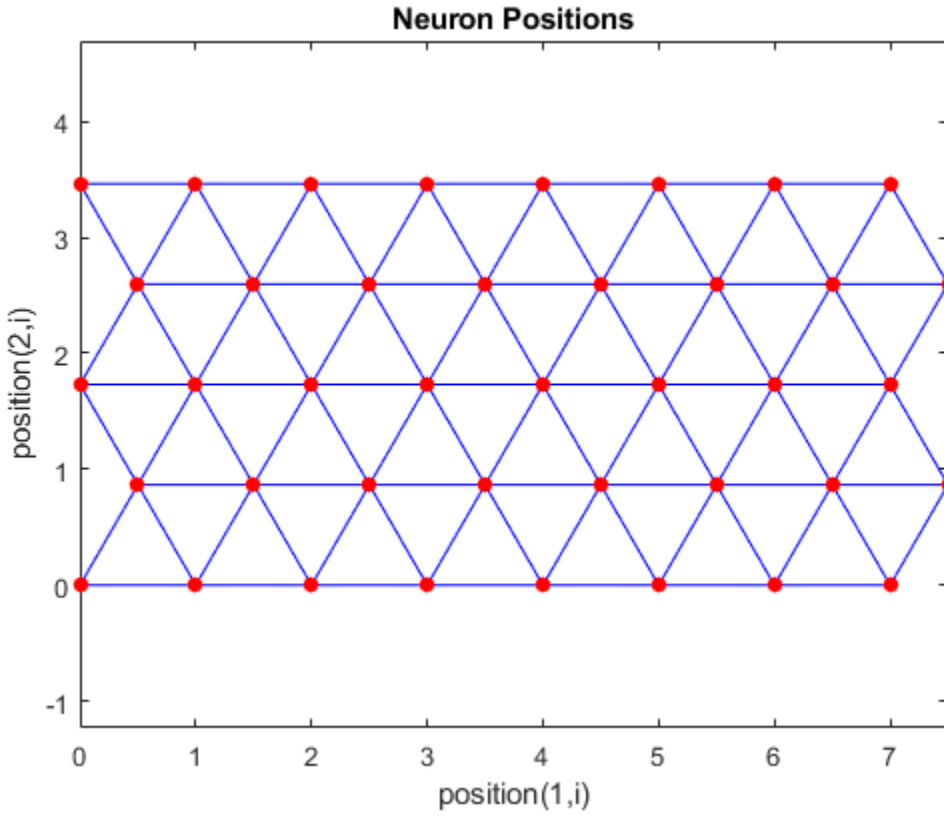
and returns an `N`-by-`S` matrix of `N` coordinate vectors where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with Hexagonal Pattern

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

```
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,

`ind`	Row vector of indices

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.

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

```
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.

```
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 = [0 1 0 1; 0 0 1 1];
t = [0 0 0 1];
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.

```
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,

S	Number of rows (neurons)
PR	R-by-2 matrix of `R = [Pmin Pmax]` (default = `[1 1]`)

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 `initcon`,

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

To initialize the network, call `init`.

Algorithms

`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.

`initcon` gets initial bias values by assuming that each neuron has responded to equal numbers of vectors in the past.

See Also

`competlayer` | `init` | `initlay` | `initwb` | `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 `net.layers{i}.initFcn`.

`net = initlay(net)` takes

`net`	Neural network

and returns the network with each layer updated.

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

`'pnames'`	Names of initialization parameters
`'pdefaults'`	Default initialization parameters

`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 `[-inf, inf]`. 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,

`net`	Neural network
`i`	Index of a layer

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,

`net`	Neural network
`i`	Index of a layer

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 such functions are `rands` and `midpoint`.)

To initialize the network, call `init`.

Algorithms

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

See Also

`init` | `initlay` | `initnw` | `linearlayer` | `perceptron`

Introduced before R2006a

initzero

Zero weight and bias initialization function

Syntax

```
W = initzero(S,PR)
b = initzero(S,[1 1])
```

Description

`W = initzero(S,PR)` takes two arguments,

S	Number of rows (neurons)
PR	R-by-2 matrix of input value ranges = `[Pmin Pmax]`

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

`b = 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 = initzero(5,[0 1; -2 2])
b = 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,

`flag`	True if all network inputs and outputs are configured (have non-zero sizes)
`inputflags`	Vector of true/false values for each configured/unconfigured input
`outputflags`	Vector of true/false values for each configured/unconfigured output

Examples

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

```
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,

`layerDelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns a layer recurrent neural network.

Examples

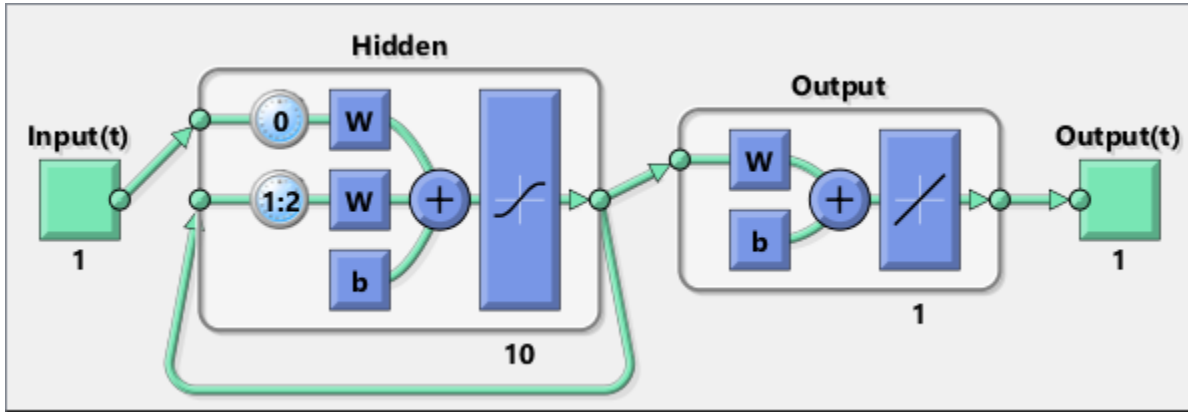
Recurrent Neural Network

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,

`B`	S-by-1 bias vector
`P`	1-by-Q ones vector
`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

dB	S-by-1 weight (or bias) change matrix
LS	New learning state

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

LP.lr - 0.001	Learning rate

info = learncon('*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

Neural Network Toolbox 2.0 compatibility: The LP.lr described above equals 1 minus the bias time constant used by trainc in the Neural Network 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.

```
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)*c + lr*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:

`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 output gradient with respect to performance x 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
LS	New learning state

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

LP.lr - 0.01	Learning rate

info = learngd('*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.

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

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

```
dW = learngd([],[],[],[],[],[],[],gW,[],[],lp,[])
```

Algorithms

learngd 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 | learngdm | 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,

`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 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

Learning occurs according to `learngdm`'s learning parameters, shown here with their default values.

`LP.lr - 0.01`	Learning rate
`LP.mc - 0.9`	Momentum constant

`info = learngdm('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 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:

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

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

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

`learngdm` returns the weight change and a new learning state.

Algorithms

`learngdm` 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*dWprev + (1-mc)*lr*gW
```

The previous weight change `dWprev` is stored and read from the learning state `LS`.

See Also

`adapt` | `learngd` | `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,

`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 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

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`.

```
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.

```
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'`.

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

Hebb, D.O., *The Organization of Behavior*, New York, Wiley, 1949

See Also

`adapt` | `learnhd` | `train`

Introduced before R2006a

learnhd

Hebb with decay weight learning rule

Syntax

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

Description

`learnhd` is the Hebb weight learning function.

`[dW,LS] = 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 `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

LS	New learning state

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

`LP.dr - 0.01`	Decay rate
`LP.lr - 0.1`	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.

```
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.

```
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,

`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 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

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

`LP.lr - 0.01`	Learning rate

`info = learnis('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`.

```
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.

```
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'-w)
```

References

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland, Reidel Press, 1982

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,

`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 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

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`.

```
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.

```
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'`.

4 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 `dW` for a given neuron from the neuron's input `P`, output `A`, and learning rate `LR` according to the Kohonen learning rule:

`dw = lr*(p'-w)`, if `a ~= 0`; `= 0`, otherwise

References

Kohonen, T., *Self-Organizing and Associative Memory*, New York, Springer-Verlag, 1984

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,

`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 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

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,:) = +lr*(p-w(i,:))` if `gA(i) = 0;= -lr*(p-w(i,:))` if `gA(i) = -1`

See Also

`adapt` | `learnlv2` | `train`

Introduced before R2006a

learnlv2

LVQ2.1 weight learning function

Syntax

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

LS	New learning state

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

`LP.lr - 0.01`	Learning rate
`LP.window - 0.25`	Window size (0 to 1, typically 0.2 to 0.3)

`info = learnlv2('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 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`.

```
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.

```
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. 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 `di` between the winning neuron and the input `p` is roughly equal to the distance `dj` from the runnerup neuron to the input `p` according to the given window,

```
min(di/dj, dj/di) > (1-window)/(1+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,:) = - lp.lr*(p'-w(i,:))
dw(j,:) = + lp.lr*(p'-w(j,:))
```

See Also

`adapt` | `learnlv1` | `train`

Introduced before R2006a

learnos

Outstar weight learning function

Syntax

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

Description

`learnos` is the outstar weight learning function.

`[dW,LS] = 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 `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

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

`LP.lr - 0.01`	Learning rate

`info = learnos('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`.

```
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.

```
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

Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland, Reidel Press, 1982

See Also

`adapt` | `learnis` | `learnk` | `train`

Introduced before R2006a

learnp

Perceptron weight and bias learning function

Syntax

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

Description

`learnp` is the perceptron weight/bias learning function.

`[dW,LS] = learnp(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 `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 = learnp('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 error `E` for a layer with a two-element input and three neurons.

```
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.

```
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 = 0, if e = 0
     = p', if e = 1
     = -p', if e = -1
```

This can be summarized as

```
dw = e*p'
```

References

Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C., Spartan Press, 1961

See Also

`adapt` | `learnpn` | `train`

Introduced before R2006a

learnpn

Normalized perceptron weight and bias learning function

Syntax

```
[dW,LS] = learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = 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] = 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:

'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 error E for a layer with a two-element input and three neurons.

```
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.

```
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) + ... + p(R)^2)
dw = 0,  if e = 0
```

```
    = pn', if e = 1
    = -pn', if e = -1
```

The expression for `dW` can be summarized as

```
dw = e*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,

`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

Learning occurs according to `learnsom`'s learning parameters, shown here with their default values.

`LP.order_lr`	`0.9`	Ordering phase learning rate
`LP.order_steps`	`1000`	Ordering phase steps
`LP.tune_lr`	`0.02`	Tuning phase learning rate
`LP.tune_nd`	`1`	Tuning phase neighborhood distance

`info = learnsom('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 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.

```
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.

```
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,  if a(i,q) = 1
         = 0.5, if a(j,q) = 1 and D(i,j) <= nd
         = 0, 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] = learnsomb(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnsomb('code')
```

Description

`learnsomb` is the batch self-organizing map weight learning function.

`[dW,LS] = learnsomb(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 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 the following:

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

LS	New learning state

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

`LP.init_neighborhood`	3	Initial neighborhood size
`LP.steps`	100	Ordering phase steps

`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.

```
p = rand(2,1);
a = rand(6,1);
w = rand(6,2);
pos = hextop(2,3);
d = linkdist(pos);
lp = learnsomb('pdefaults');
```

Because `learnsom` only needs these values to calculate a weight change (see Algorithm).

```
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] = learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
info = 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] = 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 `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

Learning occurs according to the learnwh learning parameter, shown here with its default value.

LP.lr — 0.01	Learning rate

info = learnwh('*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 error E for a layer with a two-element input and three neurons. You also define the learning rate LR learning parameter.

```
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.

```
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.

2. Set `net.adaptFcn` to `'trains'`. `net.adaptParam` automatically becomes `trains`'s default parameters.
3. Set each `net.inputWeights{i,j}.learnFcn` to `'learnwh'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnwh'`.
5. 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*e*pn'
```

References

Widrow, B., and M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record*, New York IRE, pp. 96–104, 1960

Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York, Prentice-Hall, 1985

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,

`inputDelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`widrowHoffLR`	Widrow-Hoff learning rate (default = 0.01)

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`.

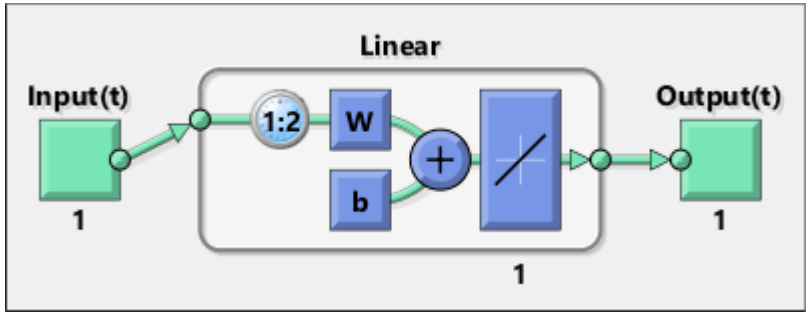
Examples

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};
net = linearlayer(1:2,0.01);
[Xs,Xi,Ai,Ts] = preparets(net,x,t);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi);
perf = perform(net,Ts,Y)
```

```
perf =

    0.2396
```



See Also

narnet | narxnet | preparets | removedelay | timedelaynet

Introduced in R2010b

linkdist

Link distance function

Syntax

```
d = linkdist(pos)
```

Description

`linkdist` is a layer distance function used to find the distances between the layer's neurons given their positions.

`d = linkdist(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.

```
pos = rand(3,10);
D = linkdist(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 `Pi` and `Pj` from a set of `S` vectors is

```
Dij = 0, if i == j
     = 1, if (sum((Pi-Pj).^2)).^0.5 is <= 1
     = 2, if k exists, Dik = Dkj = 1
     = 3, if k1, k2 exist, Dik1 = Dk1k2 = Dk2j = 1
     = N, if k1..kN exist, Dik1 = Dk1k2 = ...= DkNj = 1
     = S, if none of the above conditions apply
```

See Also

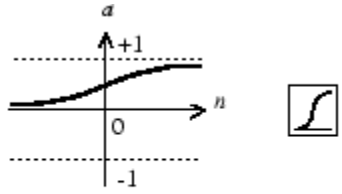
`dist` | `mandist` | `selforgmap` | `sim`

Introduced before R2006a

logsig

Log-sigmoid transfer function

Graph and Symbol



a = logsig(n)

Log-Sigmoid Transfer Function

Syntax

```
A = logsig(N,FP)
dA_dN = logsig('dn',N,A,FP)
info = logsig('code')
```

Description

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

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

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

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,

`hiddenSize`	Size of hidden layer (default = 10)
`lvqLR`	LVQ learning rate (default = 0.01)
`lvqLF`	LVQ learning function (default = `'learnlv1'`)

and returns an LVQ neural network.

The other option for the `lvq` learning function is `learnlv2`.

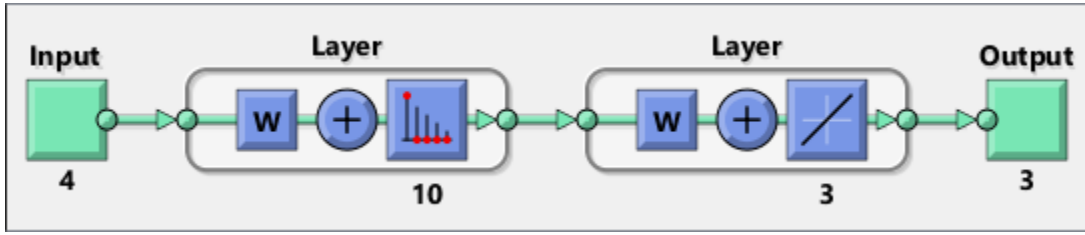
Examples

Train a Learning Vector Quantization Network

Here, an LVQ network is trained to classify iris flowers.

```
[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

```
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,

`E`	Matrix or cell array of error vectors
`Y`	Matrix or cell array of output vectors (ignored)
`X`	Vector of all weight and bias values (ignored)
`FP`	Function parameters (ignored)

and returns the mean absolute error.

`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:

```
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.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = net(p)
e = t-y
perf = mae(e)
```

Note that `mae` can be called with only one argument because the other arguments are ignored. `mae` supports those arguments to conform to the standard performance function argument list.

Network Use

You can create a standard network that uses `mae` with `perceptron`.

To prepare a custom network to be trained with `mae`, set `net.performFcn` to `'mae'`. This automatically sets `net.performParam` to the empty matrix `[]`, because `mae` has no performance parameters.

In either case, calling `train` or `adapt`, results in `mae` being used to calculate performance.

See Also

`mse` | `perceptron`

Introduced before R2006a

mandist

Manhattan distance weight function

Syntax

```
Z = mandist(W,P)
D = mandist(pos)
```

Description

`mandist` is the Manhattan distance weight function. Weight functions apply weights to an input to get weighted inputs.

`Z = 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.

`mandist` is also a layer distance function, which can be used to find the distances between neurons in a layer.

`D = mandist(pos)` takes one argument,

`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 = sum(abs(x-y))
```

See Also

`dist` | `linkdist` | `sim`

Introduced before R2006a

mapminmax

Process matrices by mapping row minimum and maximum values to [-1 1]

Syntax

```
[Y,PS] = mapminmax(X,YMIN,YMAX)
[Y,PS] = mapminmax(X,FP)
Y = mapminmax('apply',X,PS)
X = mapminmax('reverse',Y,PS)
dx_dy = mapminmax('dx_dy',X,Y,PS)
```

Description

`mapminmax` processes matrices by normalizing the minimum and maximum values of each row to [`YMIN`, `YMAX`].

`[Y,PS] = mapminmax(X,YMIN,YMAX)` takes `X` and optional parameters

`X`	`N`-by-`Q` matrix
`YMIN`	Minimum value for each row of `Y` (default is –1)
`YMAX`	Maximum value for each row of `Y` (default is +1)

and returns

`Y`	`N`-by-`Q` matrix
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = mapminmax(X,FP)` takes parameters as a struct: `FP.ymin`, `FP.ymax`.

`Y = mapminmax('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = mapminmax('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`dx_dy = mapminmax('dx_dy',X,Y,PS)` returns the reverse derivative.

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].

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0]
[y1,PS] = mapminmax(x1)
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0]
y2 = mapminmax('apply',x2,PS)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = mapminmax('reverse',y1,PS)
```

Definitions

Normalize Inputs and Targets Using `mapminmax`

Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function `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] = mapminmax(p);
[tn,ts] = mapminmax(t);
net = train(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 `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.

```
an = sim(net,pn);
a = mapminmax('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 `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 `ps`. The following code applies a new set of inputs to the network already trained.

```
pnewn = mapminmax('apply',pnew,ps);
anewn = sim(net,pnewn);
anew = mapminmax('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. (If `xmax=xmin` or if either `xmax` or `xmin` are non-finite, then `y=x` and no change occurs.)

```
y = (ymax-ymin)*(x-xmin)/(xmax-xmin) + ymin;
```

See Also

`fixunknowns` | `mapstd` | `processpca`

Introduced in R2006a

mapstd

Process matrices by mapping each row's means to 0 and deviations to 1

Syntax

```
[Y,PS] = mapstd(X,ymean,ystd)
[Y,PS] = mapstd(X,FP)
Y = mapstd('apply',X,PS)
X = mapstd('reverse',Y,PS)
dx_dy = 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] = mapstd(X,ymean,ystd)` takes `X` and optional parameters,

`X`	`N`-by-`Q` matrix
`ymean`	Mean value for each row of `Y` (default is 0)
`ystd`	Standard deviation for each row of `Y` (default is 1)

and returns

`Y`	`N`-by-`Q` matrix
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = mapstd(X,FP)` takes parameters as a struct: `FP.ymean`, `FP.ystd`.

`Y = mapstd('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = mapstd('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`dx_dy = 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.

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0]
[y1,PS] = mapstd(x1)
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0]
y2 = mapstd('apply',x2,PS)
```

Reverse the processing of `y1` to get `x1` again.

```
x1_again = mapstd('reverse',y1,PS)
```

Definitions

Normalize Network Inputs and Targets Using `mapstd`

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. The function `mapstd` normalizes the inputs and targets so that they will have zero mean and unity standard deviation. The following code illustrates the use of `mapstd`.

```
[pn,ps] = mapstd(p);
[tn,ts] = 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 `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)*(ystd/xstd) + ymean;
```

See Also

`fixunknowns` | `mapminmax` | `processpca`

Introduced in R2006a

maxlinlr

Maximum learning rate for linear layer

Syntax

```
lr = maxlinlr(P)
lr = maxlinlr(P,'bias')
```

Description

`maxlinlr` is used to calculate learning rates for `linearlayer`.

`lr = maxlinlr(P)` takes one argument,

P	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 = 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 = 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,

`m`	Mean value of all absolute finite values
`n`	Number of finite values

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] = meansqr(x)
```

Description

`[m,n] = meansqr(x)` takes a matrix or cell array of matrices and returns,

`m`	Mean value of all squared finite values
`n`	Number of finite values

If `x` contains no finite values, the mean returned is 0.

Examples

```
m = meansqr([1 2;3 4])
[m,n] = meansqr({[1 2; NaN 4], [4 5; 2 3]})
```

See Also

`meanabs` | `sumabs` | `sumsqr`

Introduced in R2010b

midpoint

Midpoint weight initialization function

Syntax

```
W = midpoint(S,PR)
```

Description

`midpoint` is a weight initialization function that sets weight (row) vectors to the center of the input ranges.

`W = midpoint(S,PR)` takes two arguments,

S	Number of rows (neurons)
PR	R-by-Q matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R matrix with rows set to `(Pmin+Pmax)'/2`.

Examples

Here initial weight values are calculated for a five-neuron layer with input elements ranging over `[0 1]` and `[-2 2]`.

```
W = midpoint(5,[0 1; -2 2])
```

See Also

`init` | `initlay` | `initwb`

Introduced before R2006a

minmax

Ranges of matrix rows

Syntax

```
pr = minmax(P)
```

Description

`pr = minmax(P)` takes one argument,

P	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 `m`th matrix is an `Ri`-by-2 matrix of the minimum and maximum values of elements for the matrix on the `i`th row of P.

Examples

```
P = [0 1 2; -1 -2 -0.5]
pr = minmax(P)
P = {[0 1; -1 -2] [2 3 -2; 8 0 2]; [1 -2] [9 7 3]};
pr = minmax(P)
```

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:

`net`	Neural network
`t`	Matrix or cell array of targets
`y`	Matrix or cell array of outputs
`ew`	Error weights (optional)

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 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.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10);
net.performParam.regularization = 0.01;
```

MSE is the default performance function for `feedforwardnet`.

```
net.performFcn
```

```
ans =
'mse'
```

Train the network and evaluate performance.

```
net = train(net, x, t);
y = net(x);
perf = perform(net, t, y)
```

```
perf = 20.7769
```

Alternatively, you can call `mse` directly.

```
perf = mse(net, t, y, 'regularization', 0.01)
```

```
perf = 20.7769
```

See Also

mae

Introduced before R2006a

narnet

Nonlinear autoregressive neural network

Syntax

```
narnet(feedbackDelays,hiddenSizes,trainFcn)
```

Description

NAR (nonlinear autoregressive) neural networks can be trained to predict a time series from that series past values.

`narnet(feedbackDelays,hiddenSizes,trainFcn)` takes these arguments,

`feedbackDelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns a NAR neural network.

Examples

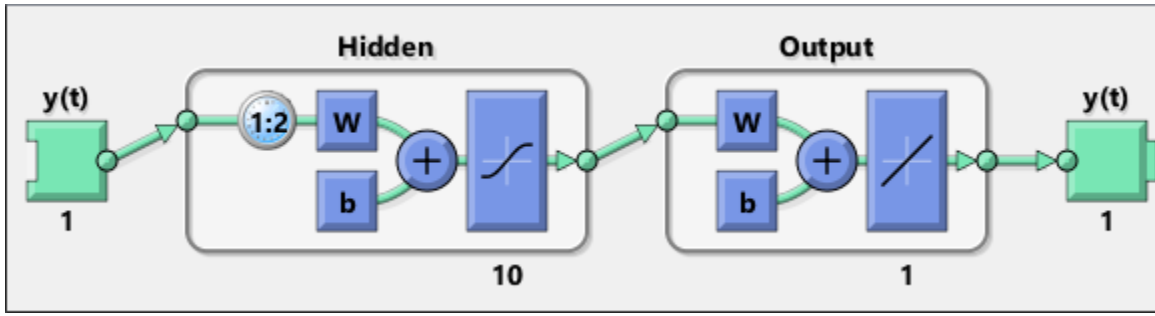
Train NAR Network and Predict on New Data

Load the simple time-series prediction data and create a NAR network.

```
T = simplenar_dataset;
net = narnet(1:2,10);
```

Prepare the time series data using `preparets` and train the network.

```
[Xs,Xi,Ai,Ts] = preparets(net,{},{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```



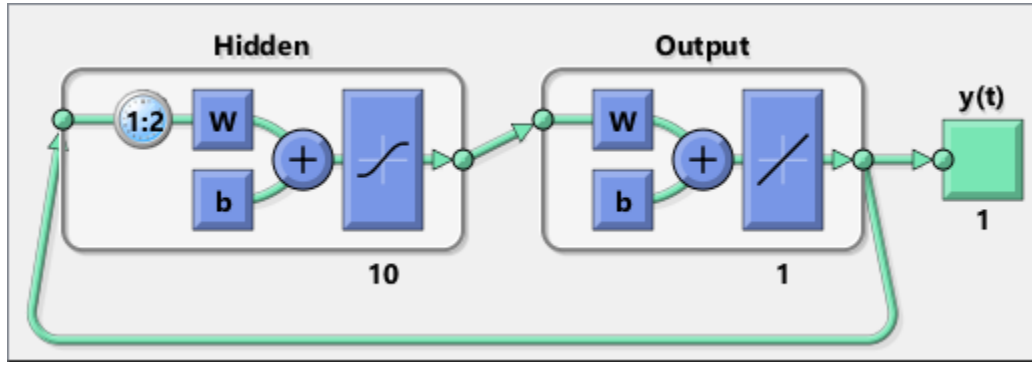
Calculate the network performance.

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y)
```

```
perf =

   1.0100e-09
```

To predict the output for the next 20 time steps, first simulate the network in closed loop form.

```
[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 `Xic` and `Aic`.

```
y2 = netc(cell(0,20),Xic,Aic)
```

```
y2 =

  1x20 cell array

  Columns 1 through 5

    {[0.8346]}    {[0.3329]}    {[0.9084]}    {[1.0000]}    {[0.3190]}

  Columns 6 through 10

    {[0.7329]}    {[0.9801]}    {[0.6409]}    {[0.5146]}    {[0.9746]}

  Columns 11 through 15

    {[0.9077]}    {[0.2807]}    {[0.8651]}    {[0.9897]}    {[0.4093]}

  Columns 16 through 20

    {[0.6838]}    {[0.9976]}    {[0.7007]}    {[0.4311]}    {[0.9660]}
```

See Also

`narnet` | `narxnet` | `preparets` | `removedelay` | `timedelaynet`

Introduced in R2010b

narxnet

Nonlinear autoregressive neural network with external input

Syntax

```
narxnet(inputDelays,feedbackDelays,hiddenSizes,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,trainFcn)` takes these arguments,

`inputDelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`feedbackDelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns a NARX neural network.

Examples

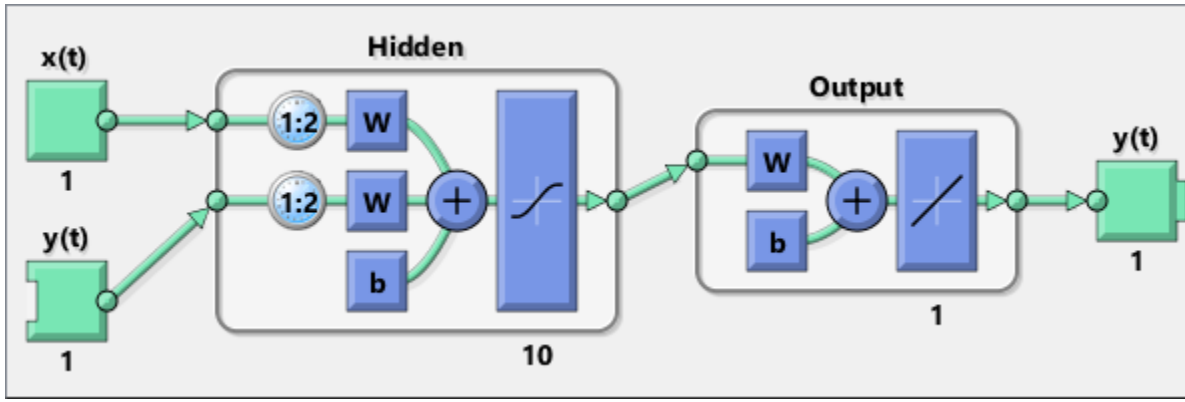
Train NARX Network and Predict on New Data

Partition the training data. 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 network, and simulate it on the first 80 observations

```
net = narxnet(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```



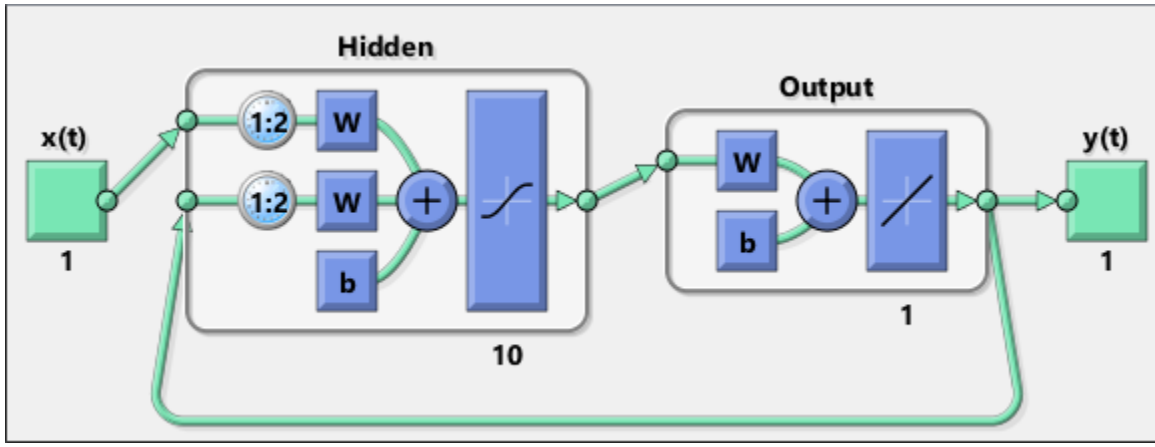
Calculate the network performance.

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y)
```

```
perf =

    0.0153
```

Run the prediction for 20 time steps ahead in closed loop mode.

```
[netc,Xic,Aic] = closeloop(net,Xf,Af);
view(netc)
```




```
y2 = netc(Xnew,Xic,Aic)

y2 =

  1x20 cell array

  Columns 1 through 5

    {[-0.0156]}    {[0.1133]}    {[-0.1472]}    {[-0.0706]}    {[0.0355]}

  Columns 6 through 10

    {[-0.2829]}    {[0.2047]}    {[-0.3809]}    {[-0.2836]}    {[0.1886]}

  Columns 11 through 15

    {[-0.1813]}    {[0.1373]}    {[0.2189]}    {[0.3122]}    {[0.2346]}

  Columns 16 through 20

    {[-0.0156]}    {[0.0724]}    {[0.3395]}    {[0.1940]}    {[0.0757]}
```

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 network 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 = negdist(W,P)
dim = negdist('size',S,R,FP)
dw = negdist('dz_dw',W,P,Z,FP)
```

Description

`negdist` is a weight function. Weight functions apply weights to an input to get weighted inputs.

`Z = negdist(W,P)` takes these inputs,

`W`	`S`-by-`R` weight matrix
`P`	`R`-by-`Q` matrix of `Q` input (column) vectors
`FP`	Row cell array of function parameters (optional, ignored)

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

`dim = negdist('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [`S`-by-`R`].

`dw = negdist('dz_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 = 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 = netinv(N,FP)
```

Description

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

`A = netinv(N,FP)` takes inputs

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `1/N`.

`info = 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 [`min max`] output range.

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

`netinv('fullderiv')` returns 1 or 0, depending on whether `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`.

```
n = rand(5,10);
a = netinv(n);
```

Assign this transfer function to layer `i` of a network.

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

`Zi`	`S`-by-`Q` matrices in a row cell array

and returns an element-wise product of `Z1` to `Zn`.

`info = netprod('code')` returns information about this function. The following codes are supported:

`'deriv'`	Name of derivative function
`'fullderiv'`	Full `N`-by-`S`-by-`Q` derivative = 1, element-wise `S`-by-`Q` derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

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 = netsum({Z1,Z2,...,Zn},FP)
info = netsum('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 = netsum({Z1,Z2,...,Zn},FP)` takes `Z1` to `Zn` and optional function parameters,

`Zi`	`S`-by-`Q` matrices in a row cell array
`FP`	Row cell array of function parameters (ignored)

and returns the elementwise sum of `Z1` to `Zn`.

`info = netsum('code')` returns information about this function. The following codes are supported:

`netsum('name')` returns the name of this function.

`netsum('type')` returns the type of this function.

`netsum('fpnames')` returns the names of the function parameters.

`netsum('fpdefaults')` returns default function parameter values.

`netsum('fpcheck', FP)` throws an error for illegal function parameters.

`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`.

```
z1 = [1, 2, 4; 3, 4, 1]
z2 = [-1, 2, 2; -5, -6, 1]
b = [0; -1]
n = netsum({z1, z2, concur(b, 3)})
```

Assign this net input function to the first layer of a network.

```
net = feedforwardnet();
net.layers{1}.netInputFcn = 'netsum';
```

See Also

`cascadeforwardnet` | `feedforwardnet` | `netinv` | `netprod`

Introduced before R2006a

network

Create custom neural network

Syntax

```
net = network
net =
network(numInputs,numLayers,biasConnect,inputConnect,layerConnect,ou
tputConnect)
```

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,ou
tputConnect)` takes these optional arguments (shown with default values):

`numInputs`	Number of inputs, 0
`numLayers`	Number of layers, 0
`biasConnect`	`numLayers`-by-1 Boolean vector, zeros

`inputConnect`	`numLayers`-by-`numInputs` Boolean matrix, zeros
`layerConnect`	`numLayers`-by-`numLayers` Boolean matrix, zeros
`outputConnect`	1-by-`numLayers` Boolean vector, zeros

and returns

`net`	New network with the given property values

Properties

Architecture Properties

`net.numInputs`	0 or a positive integer	Number of inputs.
`net.numLayers`	0 or a positive integer	Number of layers.
`net.biasConnect`	`numLayer`-by-1 Boolean vector	If `net.biasConnect(i)` is 1, then layer `i` has a bias, and `net.biases{i}` is a structure describing that bias.
`net.inputConnect`	`numLayer`-by-`numInputs` Boolean vector	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.
`net.layerConnect`	`numLayer`-by-`numLayers` Boolean vector	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.
`net.numInputs`	0 or a positive integer	Number of inputs.
`net.numLayers`	0 or a positive integer	Number of layers.
`net.biasConnect`	`numLayer`-by-1 Boolean vector	If `net.biasConnect(i)` is 1, then layer `i` has a bias, and `net.biases{i}` is a structure describing that bias.

net.inputConnect	numLayer-by-numInputs Boolean vector	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.
net.layerConnect	numLayer-by-numLayers Boolean vector	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.
net.outputConnect	1-by-numLayers Boolean vector	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.
net.numOutputs	0 or a positive integer (read only)	Number of network outputs according to net.outputConnect.
net.numInputDelays	0 or a positive integer (read only)	Maximum input delay according to all net.inputWeights{i,j}.delays.
net.numLayerDelays	0 or a positive number (read only)	Maximum layer delay according to all net.layerWeights{i,j}.delays.

Subobject Structure Properties

net.inputs	numInputs-by-1 cell array	net.inputs{i} is a structure defining input i.
net.layers	numLayers-by-1 cell array	net.layers{i} is a structure defining layer i.
net.biases	numLayers-by-1 cell array	If net.biasConnect(i) is 1, then net.biases{i} is a structure defining the bias for layer i.
net.inputWeights	numLayers-by-numInputs cell array	If net.inputConnect(i,j) is 1, then net.inputWeights{i,j} is a structure defining the weight to layer i from input j.
net.layerWeights	numLayers-by-numLayers cell array	If net.layerConnect(i,j) is 1, then net.layerWeights{i,j} is a structure defining the weight to layer i from layer j.

`net.outputs`	1-by-`numLayers` cell array	If `net.outputConnect(i)` is 1, then `net.outputs{i}` is a structure defining the network output from layer `i`.

Function Properties

`net.adaptFcn`	Name of a network adaption function or `''`
`net.initFcn`	Name of a network initialization function or `''`
`net.performFcn`	Name of a network performance function or `''`
`net.trainFcn`	Name of a network training function or `''`

Parameter Properties

`net.adaptParam`	Network adaption parameters
`net.initParam`	Network initialization parameters
`net.performParam`	Network performance parameters
`net.trainParam`	Network training parameters

Weight and Bias Value Properties

`net.IW`	`numLayers`-by-`numInputs` cell array of input weight values
`net.LW`	`numLayers`-by-`numLayers` cell array of layer weight values
`net.b`	`numLayers`-by-1 cell array of bias values

Other Properties

`net.userdata`	Structure you can use to store useful values

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.

```
net = network
net.numInputs = 1
net.numLayers = 2
```

Alternatively, you can create the same network with one line of code.

```
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.

```
net = network(1,2,[1;0],[1; 0],[0 0; 1 0],[0 1])
```

You can view the network subobjects with the following code.

```
net.inputs{1}
net.layers{1}, net.layers{2}
net.biases{1}
net.inputWeights{1,1}, net.layerWeights{2,1}
net.outputs{2}
```

You can alter the properties of any of the network subobjects. This code changes the transfer functions of both layers:

```
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'logsig';
```

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 `net.IW`. If the values are not yet set, these result is empty.

```
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 `net.LW`. Again, if the values are not yet set, the result is empty.

```
net.LW{2,1}
```

You can view the bias values for the first layer as follows.

```
net.b{1}
```

To change the number of elements in input 1 to 2, set each element's range:

```
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 (`grnn`s) are a kind of radial basis network that is often used for function approximation. `grnn`s can be designed very quickly.

`net = newgrnn(P,T,spread)` takes three inputs,

`P`	R-by-`Q` matrix of `Q` input vectors
`T`	S-by-`Q` matrix of `Q` target class vectors
`spread`	Spread of radial basis functions (default = 1.0)

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

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York, Van Nostrand Reinhold, 1993, pp. 155–61

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,

`P`	`R`-by-`Q` matrix of `Q` input vectors
`T`	`S`-by-`Q` matrix of `Q` target class vectors
`Pi`	`1`-by-`ID` cell array of initial input delay states

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:

`P`	`Ni`-by-`TS` cell array	Each element `P{i,ts}` is an `Ri`-by-`Q` input matrix
`T`	`Nt`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix
`Pi`	`Ni`-by-`ID` cell array	Each element `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 `Pi1` for input 1 and three initial delays states `Pi2` for input 2.

```
P1 = {1 2 1 3 3 2}; Pi1 = {1 3 0};
P2 = {1 2 1 1 2 1}; Pi2 = {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],[Pi1; Pi2]);
Y = sim(net,[P1; P2],[Pi1; Pi2]);
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,

`P`	R-by-`Q` matrix of `Q` input vectors
`T`	S-by-`Q` matrix of `Q` target class vectors
`spread`	Spread of radial basis functions (default = 0.1)

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 = sim(net,P)
Yc = 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/spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/– `spread`. The second-layer weights `W2` are set to `T`.

References

Wasserman, P.D., *Advanced Methods in Neural Computing*, New York, Van Nostrand Reinhold, 1993, pp. 35–55

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,

`P`	`R`-by-`Q` matrix of `Q` input vectors
`T`	`S`-by-`Q` matrix of `Q` target class vectors
`goal`	Mean squared error goal (default = 0.0)
`spread`	Spread of radial basis functions (default = 1.0)
`MN`	Maximum number of neurons (default is `Q`)
`DF`	Number of neurons to add between displays (default = 25)

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,

`P`	RxQ matrix of Q R-element input vectors
`T`	SxQ matrix of Q S-element target class vectors
`spread`	Spread of radial basis functions (default = 1.0)

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 `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.

`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 network fitting tool

Syntax

```
nftool
```

Description

`nftool` opens the neural network fitting tool 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` | `nprtool` | `ntstool`

Introduced in R2006a

nncell2mat

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,

`y`	Cell array formed by concatenating matrices
`i`	Array of row sizes
`ji`	Array of column sizes

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

Crross correlation between neural network time series

Syntax

```
nncorr(a,b,maxlag,'flag')
```

Description

`nncorr(a,b,maxlag,'flag')` takes these arguments,

`a`	Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of `N`.
`b`	Matrix or cell array, with columns interpreted as timesteps, and having a total number of matrix rows of `M`.
`maxlag`	Maximum number of time lags
`flag`	Type of normalization (default = `'none'`)

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.

```
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.

```
b = nndata(2,1,20)
ab = nncorr(a,b,8)
```

See Also

`confusion` | `regression`

Introduced in R2010b

nndata

Create neural network data

Syntax

```
nndata(N,Q,TS,v)
```

Description

`nndata(N,Q,TS,v)` takes these arguments,

`N`	Vector of `M` element sizes
`Q`	Number of samples
`TS`	Number of timesteps
`v`	Scalar value

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 `getelements`, `getsamples`, `gettimesteps`, and `getsignals`.

You can set subsets of neural network data with `setelements`, `setsamples`, `settimesteps`, and `setsignals`.

You can concatenate subsets of neural network data with `catelements`, `catsamples`, `cattimesteps`, and `catsignals`.

Examples

Here four samples of five timesteps, for a 2-element signal consisting of zero values is created:

```
x = nndata(2,4,5,0)
```

To create random data with the same dimensions:

```
x = nndata(2,4,5)
```

Here static (1 timestep) data of 12 samples of 4 elements is created.

```
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 `i`th 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:

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

See Also

gpu2nndata

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,

`x`	Neural network data
`i`	Index of signal (default = 1)
`q`	Index of sample (default = 1)

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.

```
x = nndata([4;3],3,10);
```

Now the second signal of the first series is converted to Simulink form.

```
y_2_1 = nndata2sim(x,2,1)
```

See Also

`nndata` | `nnsize` | `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,

`N`	Vector containing the number of element sizes for each of `M` signals
`Q`	Number of samples
`TS`	Number of timesteps
`M`	Number of signals

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 `Sx1` 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 = [1 2 3; 4 7 4]
[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 Neural Network Toolbox”.

See Also

`nctool` | `nftool` | `nprtool` | `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

```
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.

```
[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.

```
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 = normprod(W,P,FP)
dim = normprod('size',S,R,FP)
dw = normprod('dz_dw',W,P,Z,FP)
```

Description

`normprod` is a weight function. Weight functions apply weights to an input to get weighted inputs.

`Z = normprod(W,P,FP)` takes these inputs,

`W`	`S`-by-`R` weight matrix
`P`	`R`-by-`Q` matrix of `Q` input (column) vectors
`FP`	Row cell array of function parameters (optional, ignored)

and returns the `S`-by-`Q` matrix of normalized dot products.

`dim = normprod('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size [`S`-by-`R`].

`dw = normprod('dz_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 = w*p/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

```
m = [1 2; 3 4];
normr(m)
ans =
      0.4472     0.8944
      0.6000     0.8000
```

See Also

`normc`

Introduced before R2006a

nprtool

Neural network pattern recognition tool

Syntax

```
nprtool
```

Description

`nprtool` opens the neural network 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,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

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.

```
[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.

$$y_1 = y(x + 2dx)$$
$$y_2 = y(x + dx)$$
$$y_3 = y(x - dx)$$
$$y_4 = y(x - 2dx)$$
$$\frac{dy}{dx} = \frac{-y1 + 8y_2 - 8y_3 + y_4}{12dx}$$

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,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

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).

`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.

```
[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

```
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:

```
x = [1 2 3; 4 7 4]
n = numelements(x)
```

This code calculates the number of elements represented by cell data:

```
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

```
x = [1 2; 3 NaN]
n = numfinite(x)

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` | `numsignals` | `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:

```
x = [1 2 3; 4 7 4]
n = numsignals(x)
```

This code calculates the number of signals represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numsignals(x)
```

See Also

`catsignals` | `getsignals` | `nndata` | `nnsize` | `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:

```
x = [1 2 3; 4 7 4]
n = numtimesteps(x)
```

This code calculates the number of time steps represented by cell data:

```
x = {[1:3; 4:6] [7:9; 10:12]; [13:15] [16:18]}
n = numtimesteps(x)
```

See Also

`cattimesteps` | `gettimesteps` | `nndata` | `nnsize` | `numelements` | `numsamples` | `numsignals` | `settimesteps`

Introduced in R2010b

openloop

Convert neural network closed-loop feedback to open loop

Syntax

```
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.

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
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);
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,

`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainscg'`)
`performFcn`	Performance function (default = `'crossentropy'`)

and returns a pattern recognition neural network.

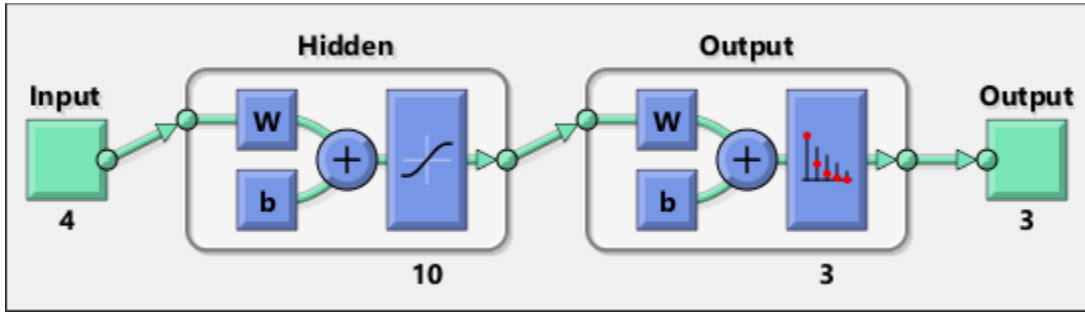
Examples

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);
```



See Also

`competlayer` | `lvqnet` | `network` | `nprtool` | `selforgmap`

Topics

“Classify Patterns with a Shallow Neural Network”
“Neural Network Object Properties”
“Neural Network Subobject Properties”

Introduced in R2010b

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,

`hardlimitTF`	Hard limit transfer function (default = `'hardlim'`)
`perceptronLF`	Perceptron learning rule (default = `'learnp'`)

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`.

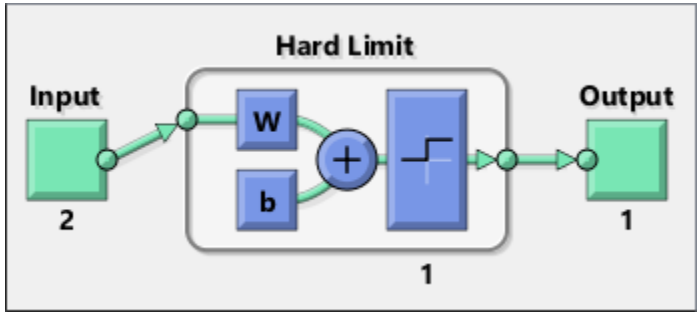
Note Neural Network Toolbox supports perceptrons for historical interest. For better results, you should instead use `patternnet`, which can solve nonlinearly separable problems. Sometimes the term “perceptrons” refers to feed-forward pattern recognition networks; but the original perceptron, described 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 = [0 0 1 1; 0 1 0 1];
t = [0 1 1 1];
net = perceptron;
net = train(net,x,t);
view(net)
y = net(x);
```



See Also

narnet | narxnet | patternnet | preparets | removedelay | timedelaynet

Introduced in R2010b

perform

Calculate network performance

Syntax

```
perform(net,t,y,ew)
```

Description

`perform(net,t,y,ew)` takes these arguments,

`net`	Neural network
`t`	Target data
`y`	Output data
`ew`	Error weights (default = `{1}`)

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 may 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.

```
[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,...,ta
rgetsn,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,...,ta rgetsn,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
```

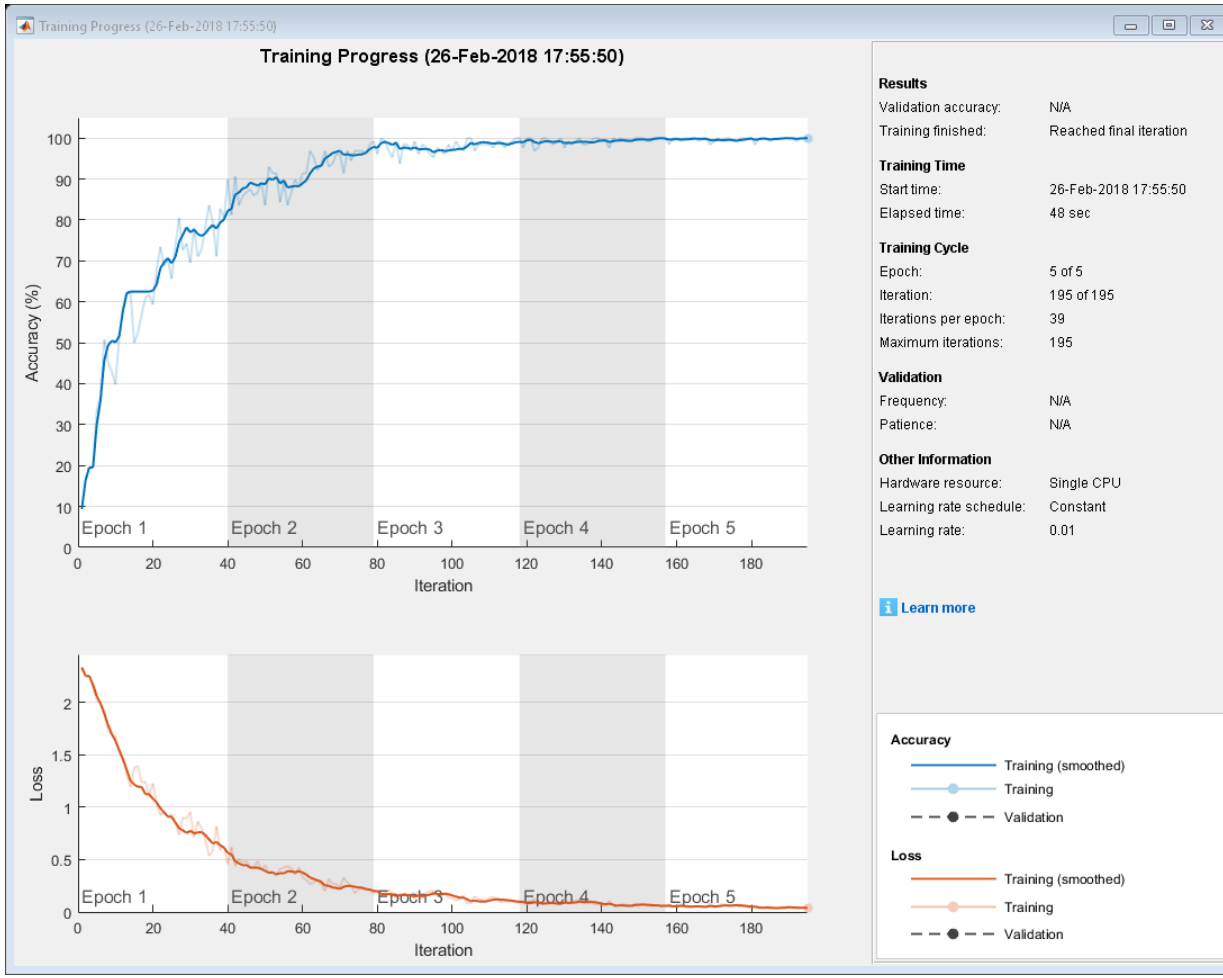
```
  Name           Size             Bytes  Class          Attributes

  YTrain      5000x1              6142  categorical
```

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 prediced labels `YPredicted`.

```
plotconfusion(YTest,YPredicted)
```


Confusion Matrix
Output Class
Target Class
0 1 2 3 4 5 6 7 8 9
499 10.0% | 0 0.0% | 1 0.0% | 1 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 99.2% 0.8%
0 0.0% | 481 9.6% | 2 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 99.6% 0.4%
0 0.0% | 7 0.1% | 493 9.9% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 0 0.0% | 98.4% 1.6%
0 0.0% | 0 0.0% | 2 0.0% | 495 9.9% | 0 0.0% | 1 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 99.2% 0.8%
0 0.0% | 3 0.1% | 1 0.0% | 0 0.0% | 500 10.0% | 0 0.0% | 1 0.0% | 0 0.0% | 2 0.0% | 0 0.0% | 98.6% 1.4%
0 0.0% | 1 0.0% | 0 0.0% | 2 0.0% | 0 0.0% | 498 10.0% | 2 0.0% | 1 0.0% | 0 0.0% | 0 0.0% | 98.8% 1.2%
0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 495 9.9% | 0 0.0% | 0 0.0% | 0 0.0% | 100% 0.0%
0 0.0% | 8 0.2% | 1 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 0 0.0% | 498 10.0% | 0 0.0% | 1 0.0% | 98.0% 2.0%
1 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 0 0.0% | 1 0.0% | 0 0.0% | 1 0.0% | 495 9.9% | 2 0.0% | 98.8% 1.2%
0 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 0 0.0% | 0 0.0% | 1 0.0% | 0 0.0% | 2 0.0% | 495 9.9% | 99.2% 0.8%
99.8% 0.2% | 96.2% 3.8% | 98.6% 1.4% | 99.0% 1.0% | 100% 0.0% | 99.6% 0.4% | 99.0% 1.0% | 99.6% 0.4% | 99.0% 1.0% | 99.0% 1.0% | 99.0% 1.0%

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.

```
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.

```
net = patternnet(10);
net = train(net,XTrain,YTrain);
```

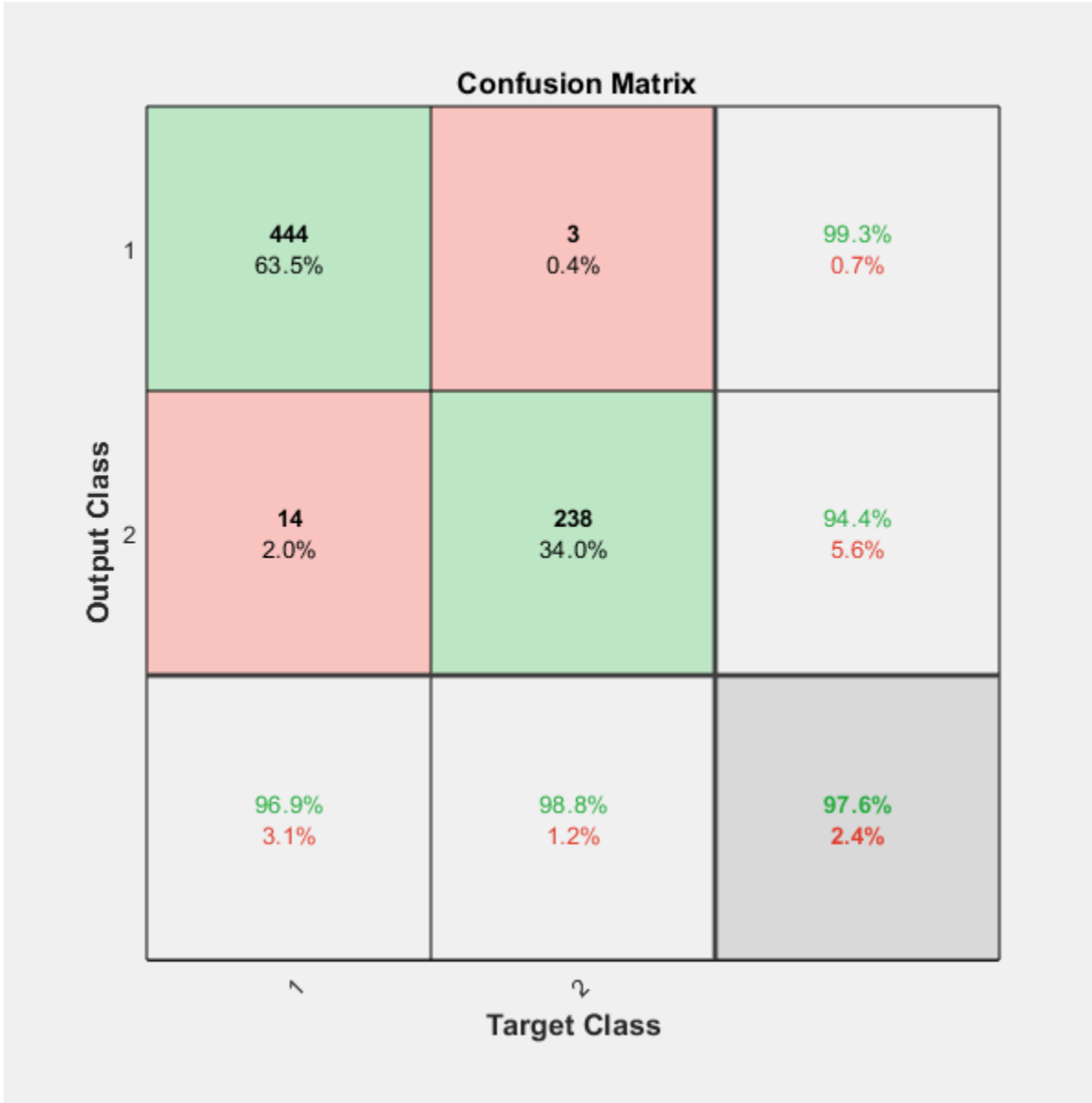
Estimate the cancer status using the trained network. Each column of the matrix `YPredicted` contains the predicted probabilities of each observation belonging to class 1 and class 2, respectively.

```
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    0.0
    0.0001    0.0001    0.0001    0.9422    0.0007    0.0001    0.9988    0.9999    0.9
```

Plot the confusion matrix. To create the plot, `plotconfusion` labels each observation according to the highest class probability.

```
plotconfusion(YTrain,YPredicted)
```


Confusion Matrix
444
63.5%
3
0.4%
99.3%
0.7%
14
2.0%
238
34.0%
94.4%
5.6%
96.9%
3.1%
98.8%
1.2%
97.6%
2.4%
Output Class
1
2
1
2
Target Class

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 = plotep(W,B,E)
H = plotep(W,B,E,H)
```

Description

`plotep` is used to show network learning on a plot created by `plotes`.

`H = plotep(W,B,E)` takes these arguments,

`W`	Current weight value
`B`	Current bias value
`E`	Current error

and returns a cell array `H`, containing information for continuing the plot.

`H = 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.

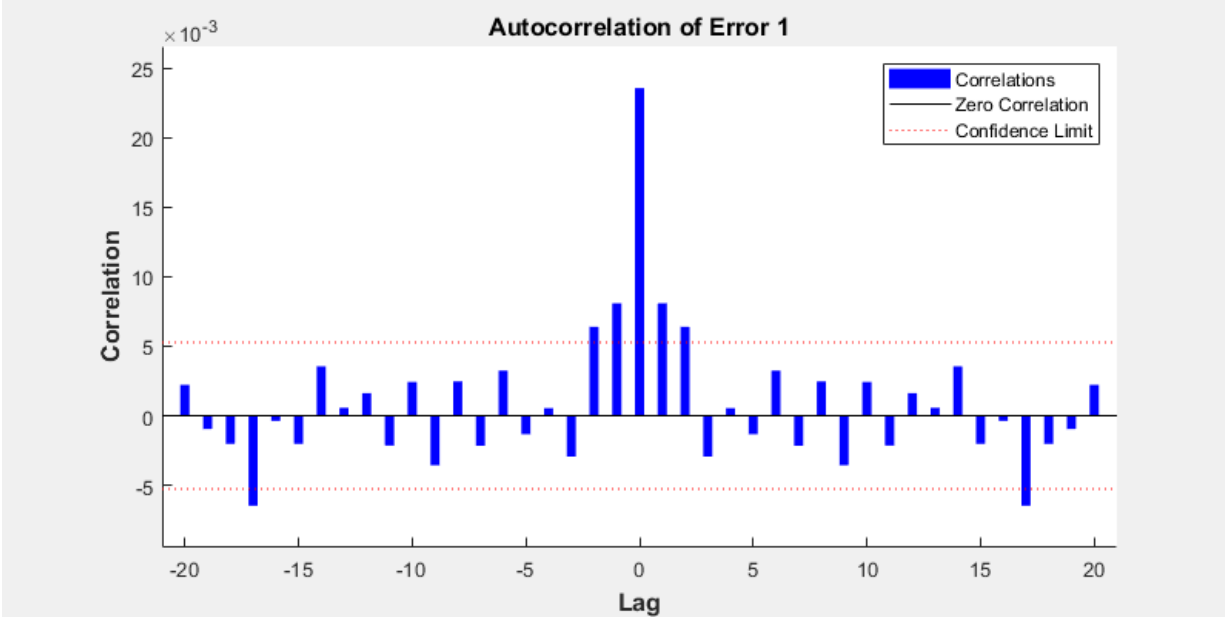
`ploterrcorr(errors,'outputIndex',outIdx)` uses the optional property name/value pair to define which output error autocorrelation is plotted. The default is 1.

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.

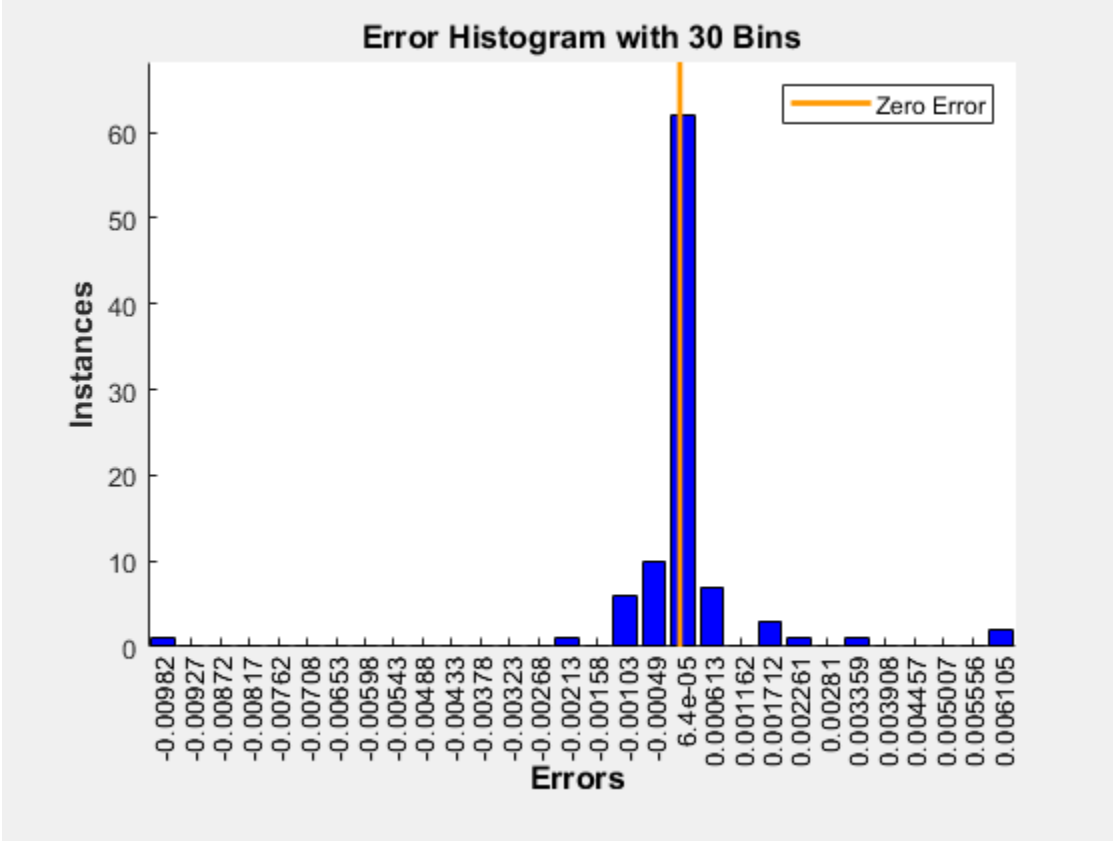
`ploterrhist(...,'bins',bins)` takes an optional property name/value pair which defines the number of bins to use in the histogram plot. The default is 20.

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,

`WV`	1-by-`N` row vector of values of `W`
`BV`	1-by-`M` row vector of values of `B`
`ES`	`M`-by-`N` matrix of error vectors
`V`	View (default = `[-37.5, 30]`)

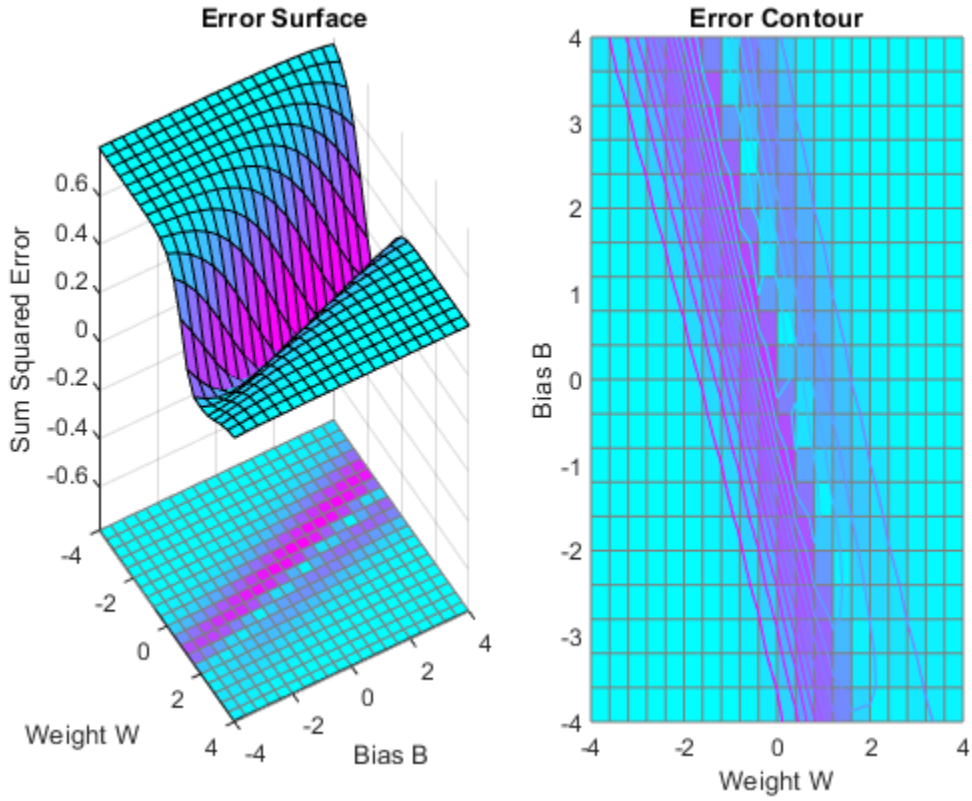
and plots the error surface with a contour underneath.

Calculate the error surface `ES` with `errsurf`.

Examples

Plot Error Surface of Single-Input Neuron

```
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

`errsurf`

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.

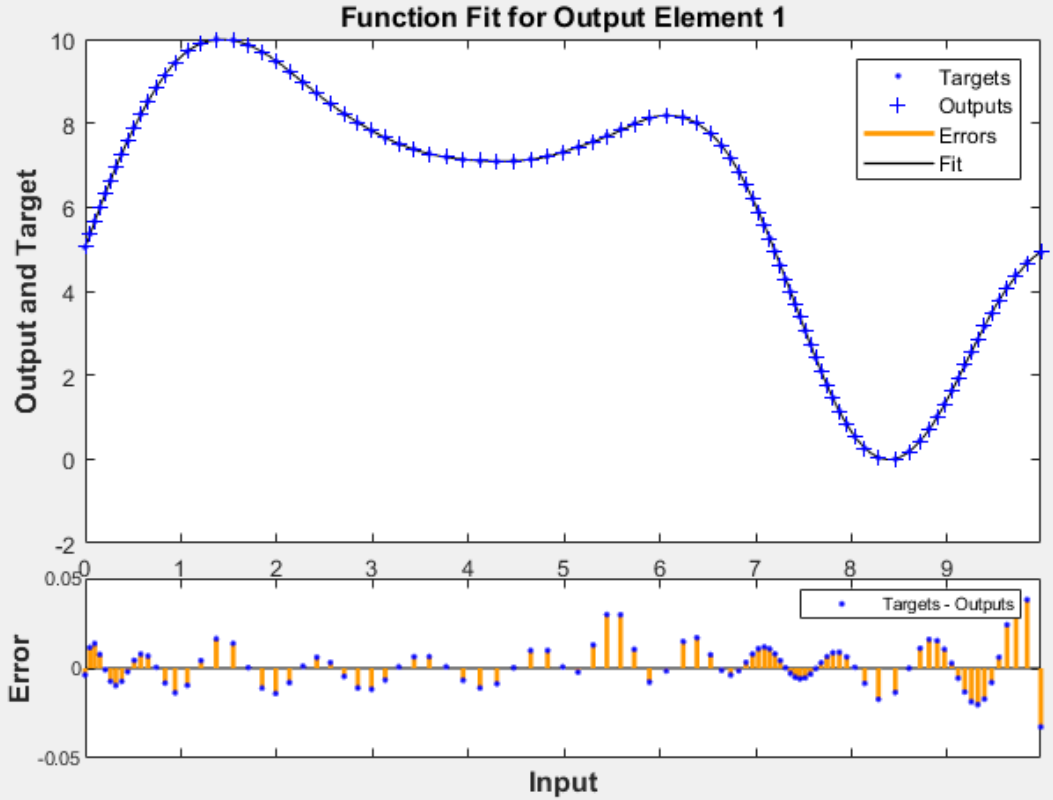
`plotfit(targets1,inputs1,'name1',...)` displays a series of plots.

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.

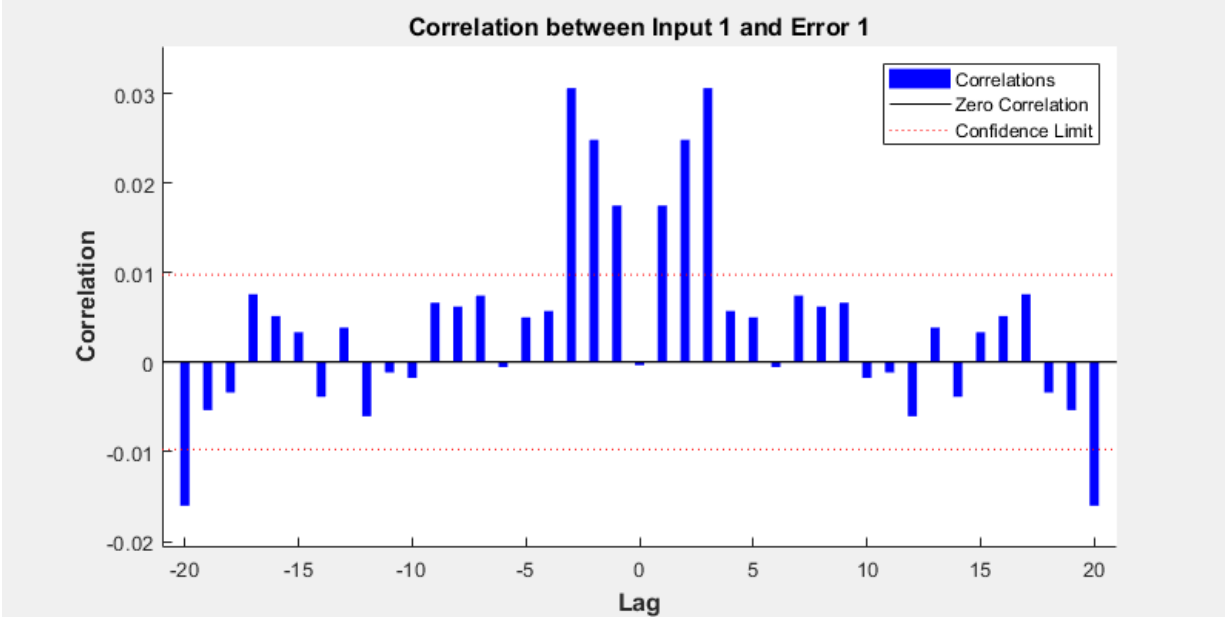
`plotinerrcorr(...,'outputIndex',outputIndex)` optionally defines which error element is being correlated and plotted. The default is 1.

Examples

Plot Cross-Correlation of Inputs to Errors

Here a NARX network is used to solve a time series problem.

```
[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,

`W`	S-by-R weight matrix (R must be 3 or less)
`B`	S-by-1 bias vector

and returns a handle to a plotted classification line.

`plotpc(W,B,H)` takes an additional input,

`H`	Handle to last plotted line

and deletes the last line before plotting the new one.

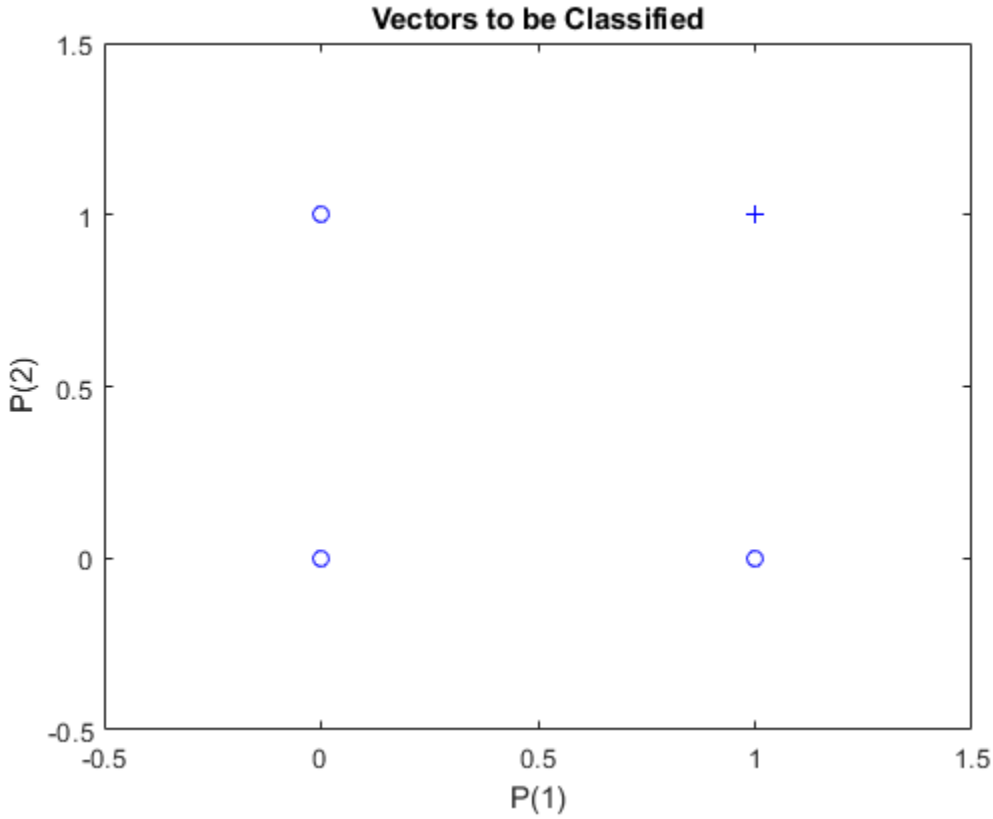
This function does not change the current axis and is intended to be called after `plotpv`.

Examples

Plot Classification Line

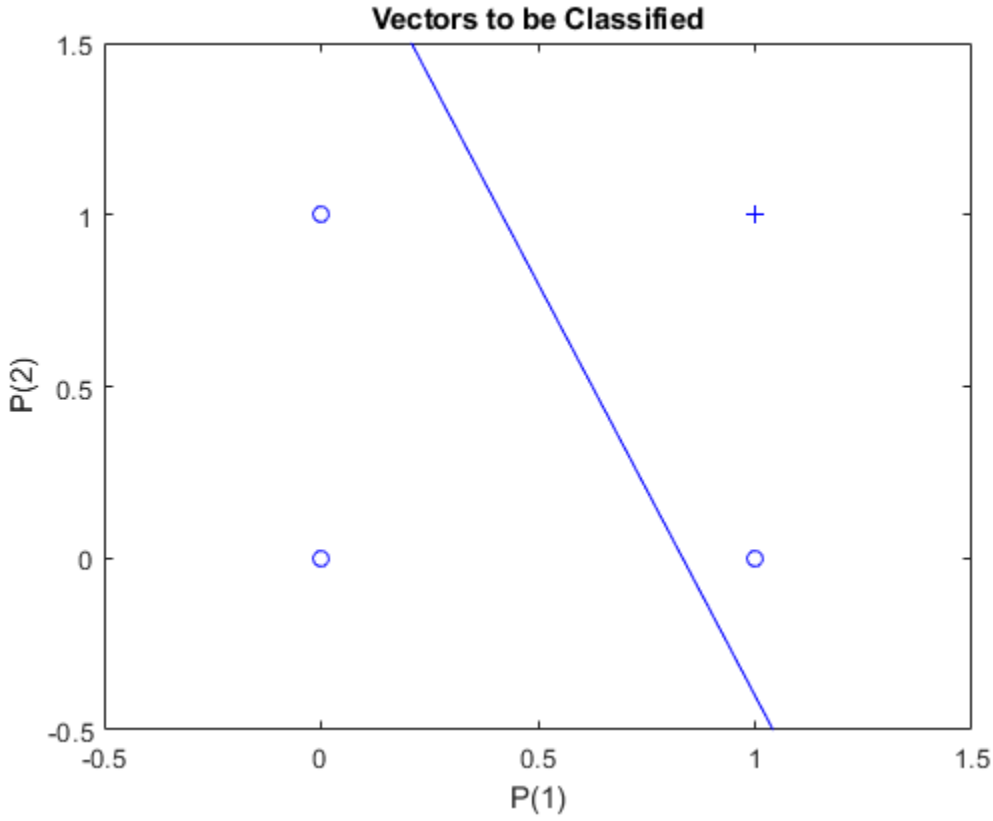
The code below defines and plots the inputs and targets for a perceptron:

```
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.

```
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

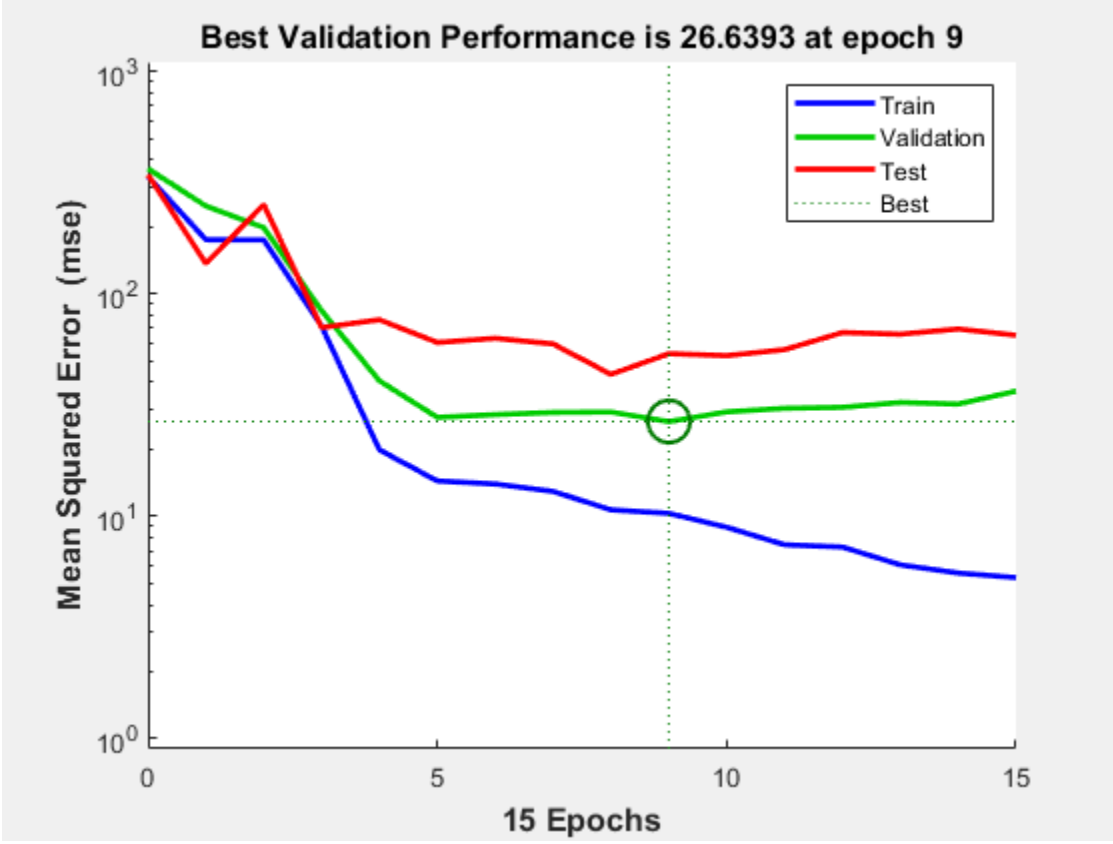
`plotperform(TR)` plots error vs. epoch for the training, validation, and test performances of the training record `TR` returned by the function `train`.

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

```
plotpv(P,T)
plotpv(P,T,V)
```

Description

`plotpv(P,T)` takes these inputs,

`P`	R-by-Q matrix of input vectors (R must be 3 or less)
`T`	S-by-Q matrix of binary target vectors (S must be 3 or less)

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]`

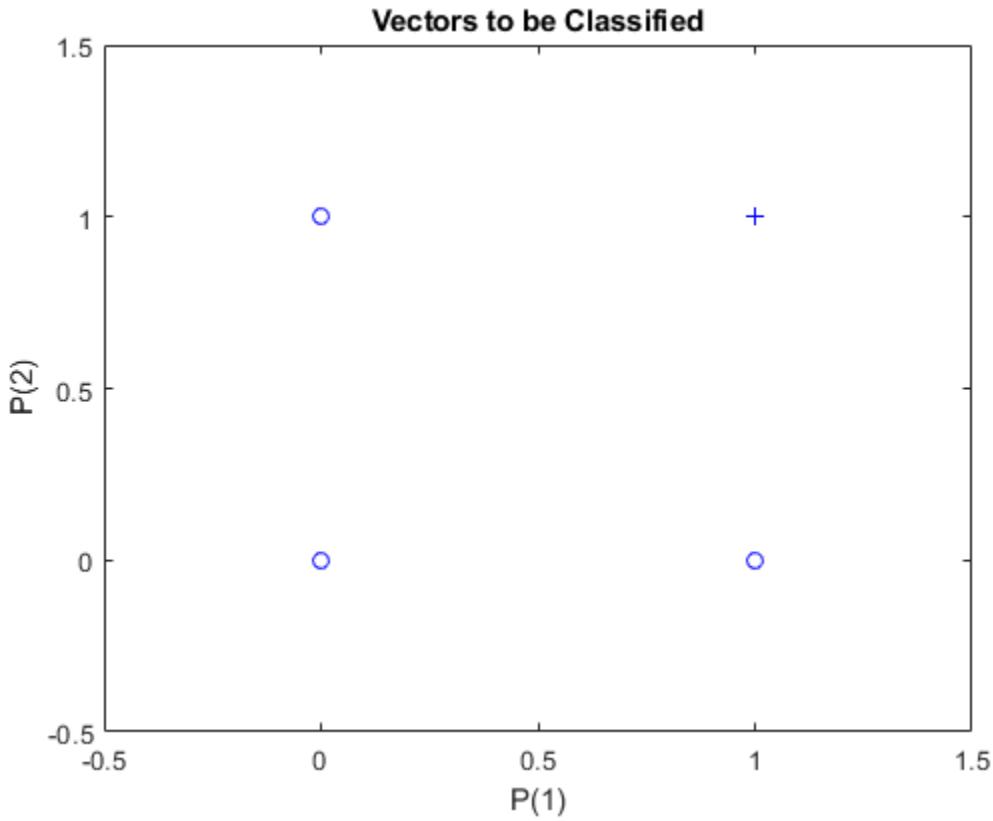
and plots the column vectors with limits set by `V`.

Examples

Plot Inputs and Targets for Perceptron

This example shows how to define and plot the inputs and targets for a perceptron.

```
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

```
plotregression(targets,outputs)
plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)
```

Description

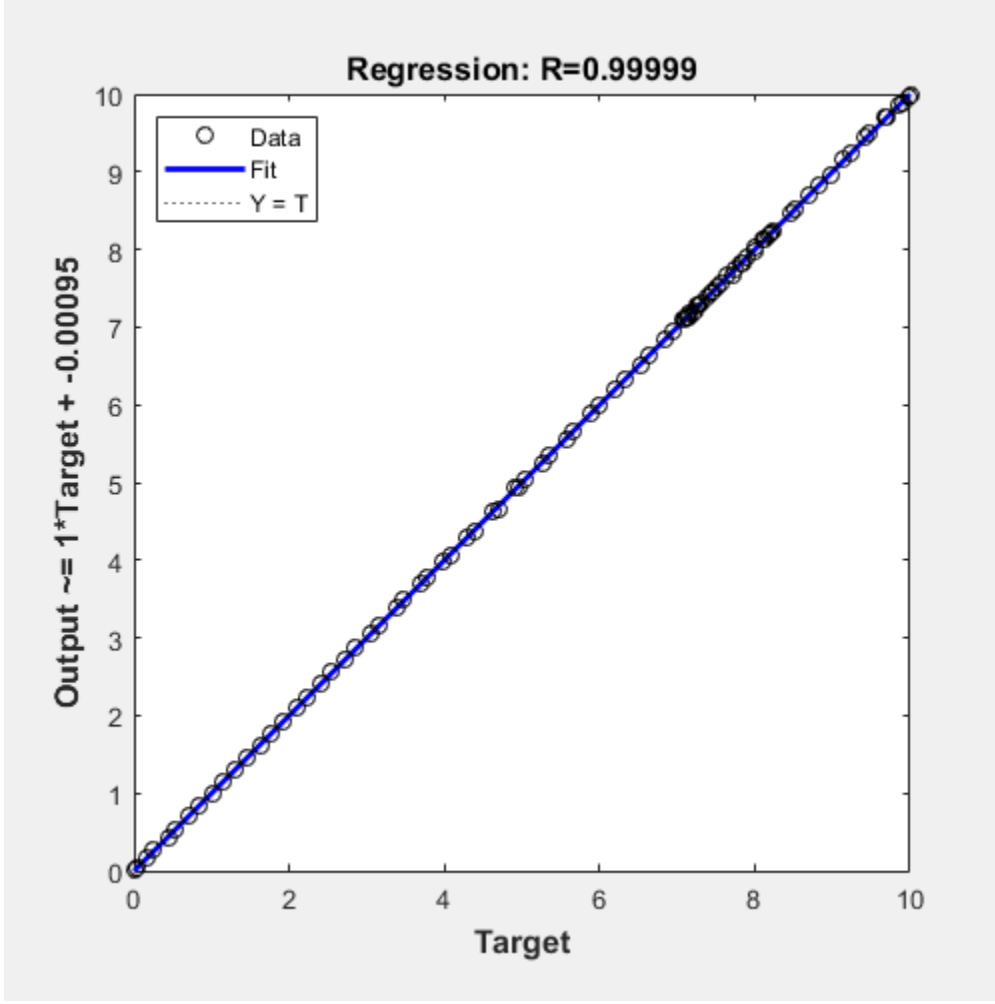
`plotregression(targets,outputs)` plots the linear regression of `targets` relative to `outputs`.

`plotregression(targs1,outs1,'name1',targs2,outs2,'name2',...)` generates multiple plots.

Examples

Plot Linear Regression

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

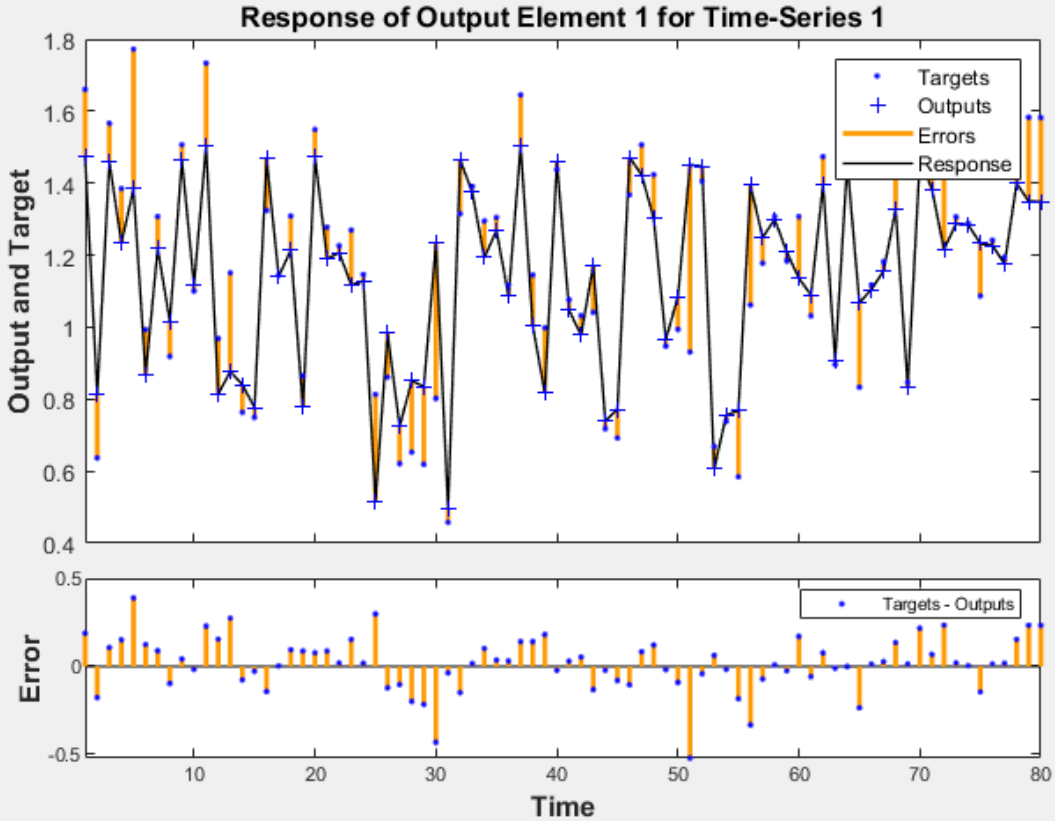
`plotresponse(...,'outputIndex',outputIndex)` optionally defines which error element is being correlated and plotted. The default is 1.

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] = preparets(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

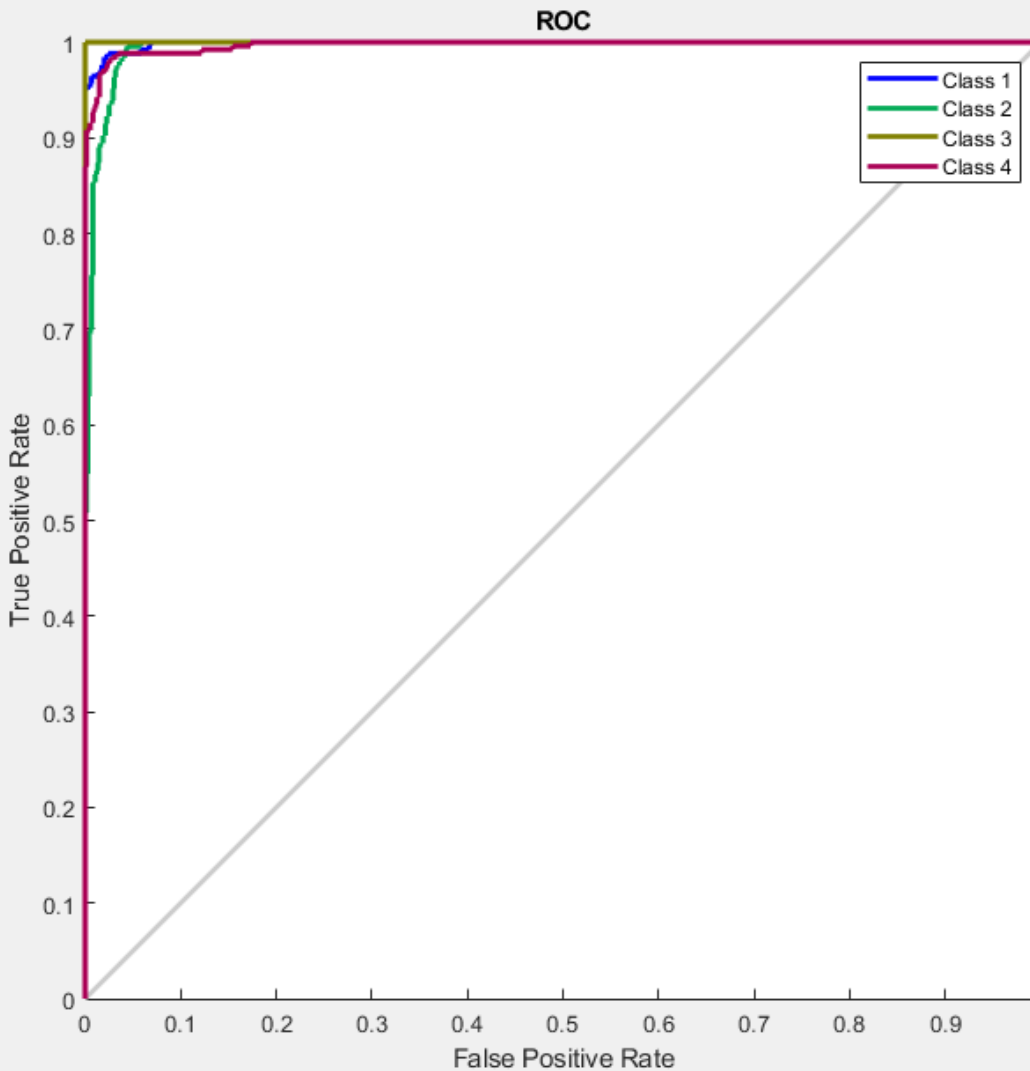
`plotroc(targets,outputs)` plots the receiver operating characteristic for each output class. The more each curve hugs the left and top edges of the plot, the better the classification.

`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)
```


ROC
True Positive Rate
False Positive Rate
Class 1
Class 2
Class 3
Class 4
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

See Also

`roc`

Introduced in R2008a

plotsom

Plot self-organizing map

Syntax

```
plotsom(pos)
plotsom(W,D,ND)
```

Description

`plotsom(pos)` takes one argument,

`POS`	`N`-by-`S` matrix of `S` `N`-dimension neural positions

and plots the neuron positions with red dots, linking the neurons within a Euclidean distance of 1.

`plotsom(W,D,ND)` takes three arguments,

`W`	`S`-by-`R` weight matrix
`D`	`S`-by-`S` distance matrix
`ND`	Neighborhood distance (default = 1)

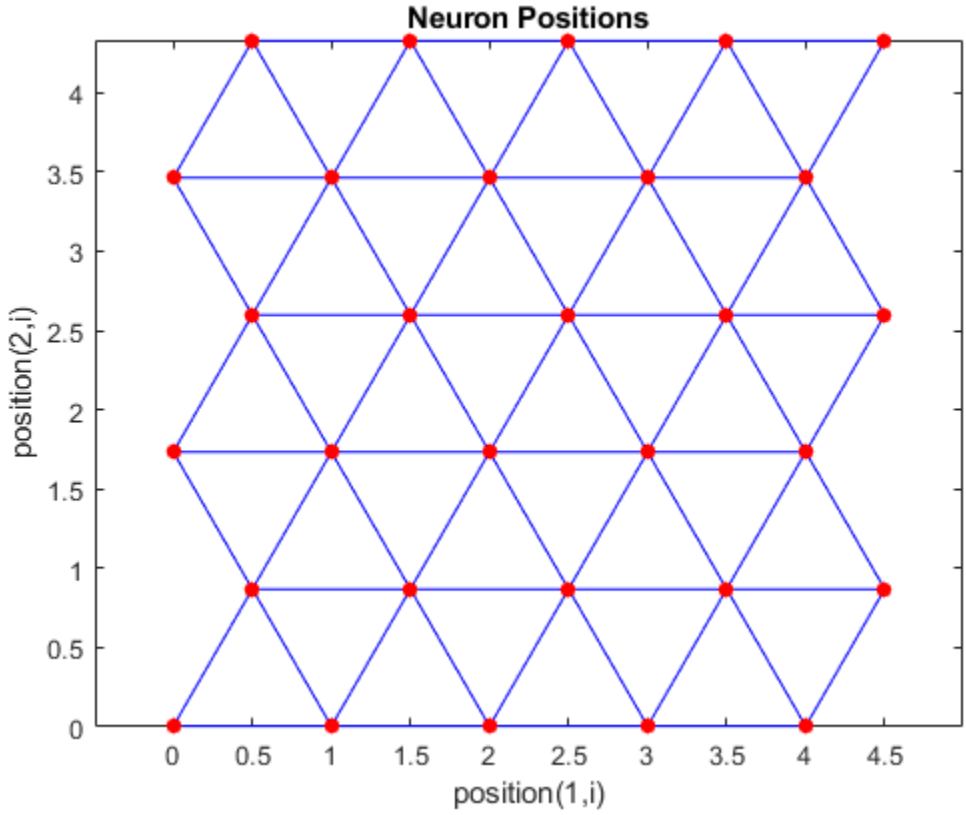
and plots the neuron's weight vectors with connections between weight vectors whose neurons are within a distance of 1.

Examples

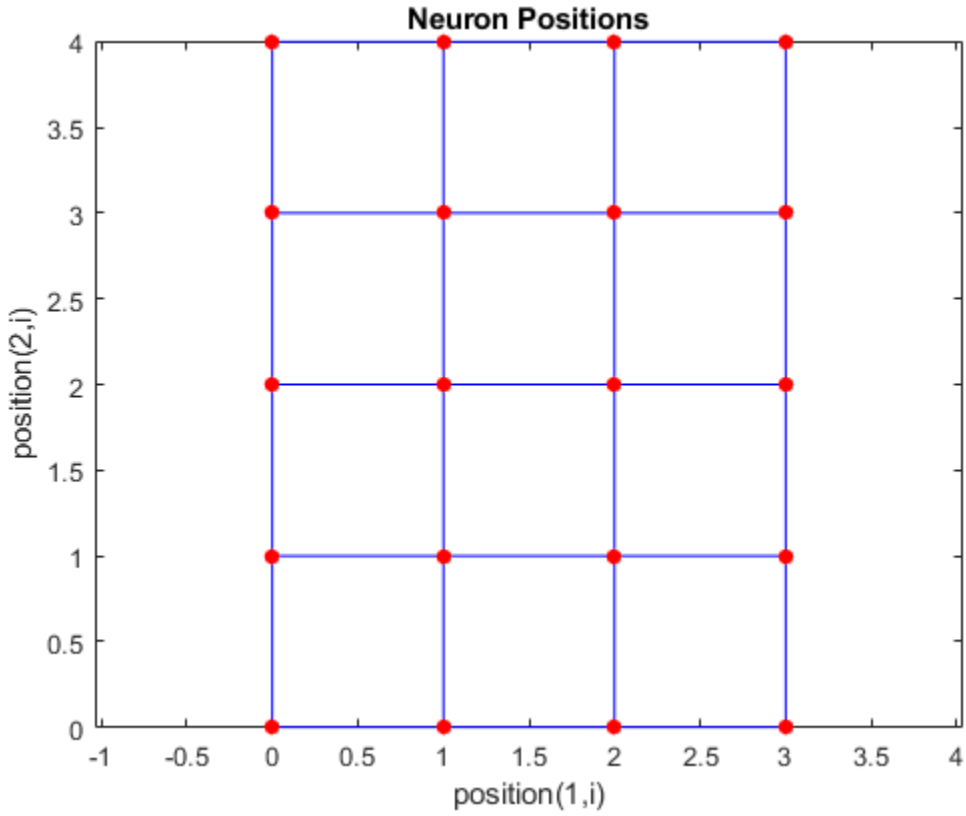
Plot Self-Organizing Maps

These examples generate plots of various layer topologies.

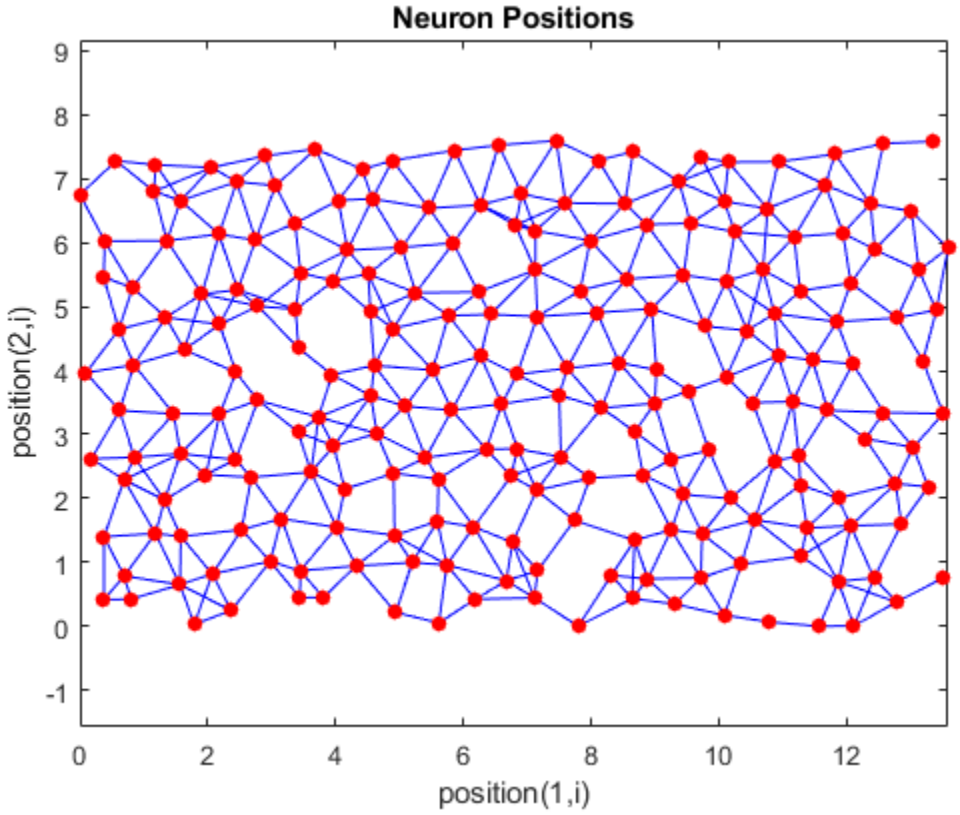
```
pos = hextop([5 6]);
plotsom(pos)
```



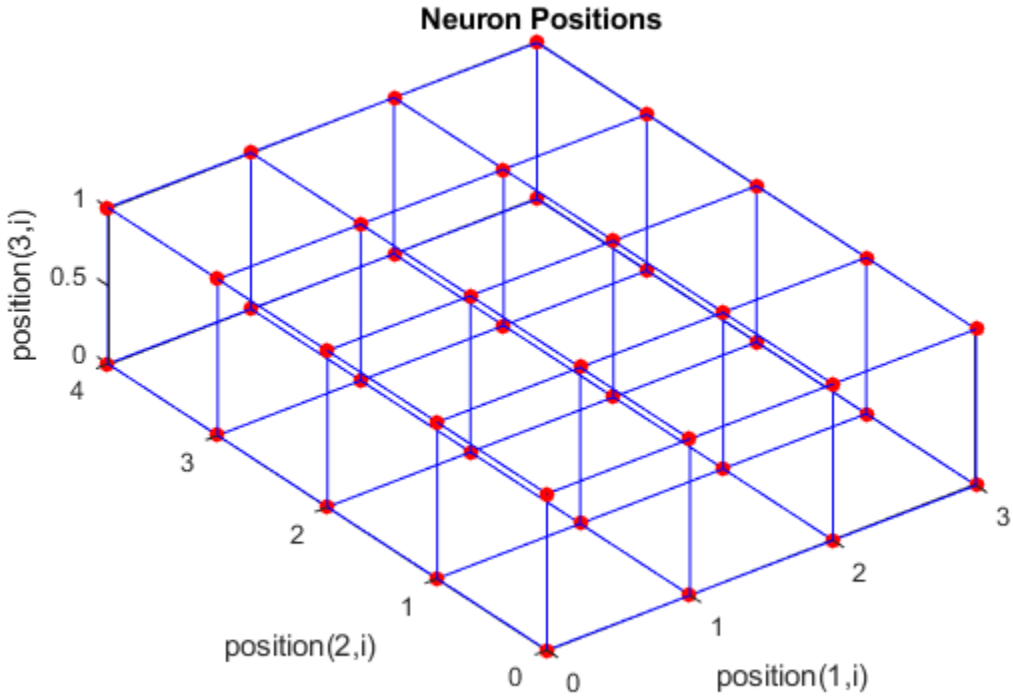

```
pos = gridtop([4 5]);
plotsom(pos)
```



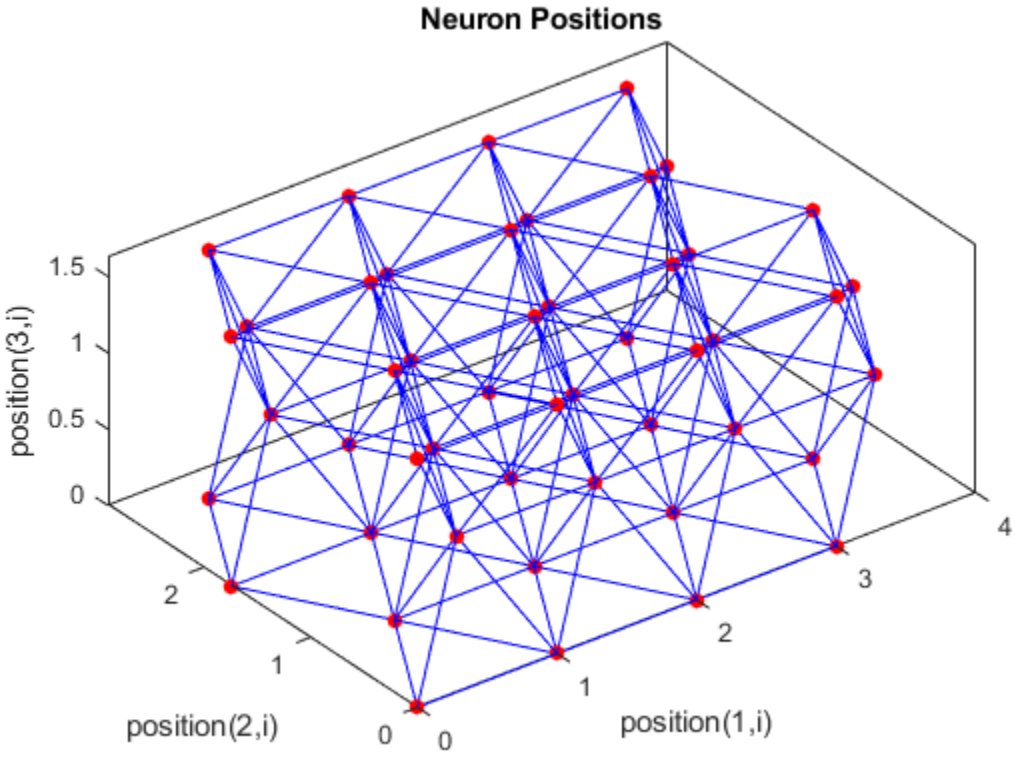

```
pos = randtop([18 12]);
plotsom(pos)
```




```
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

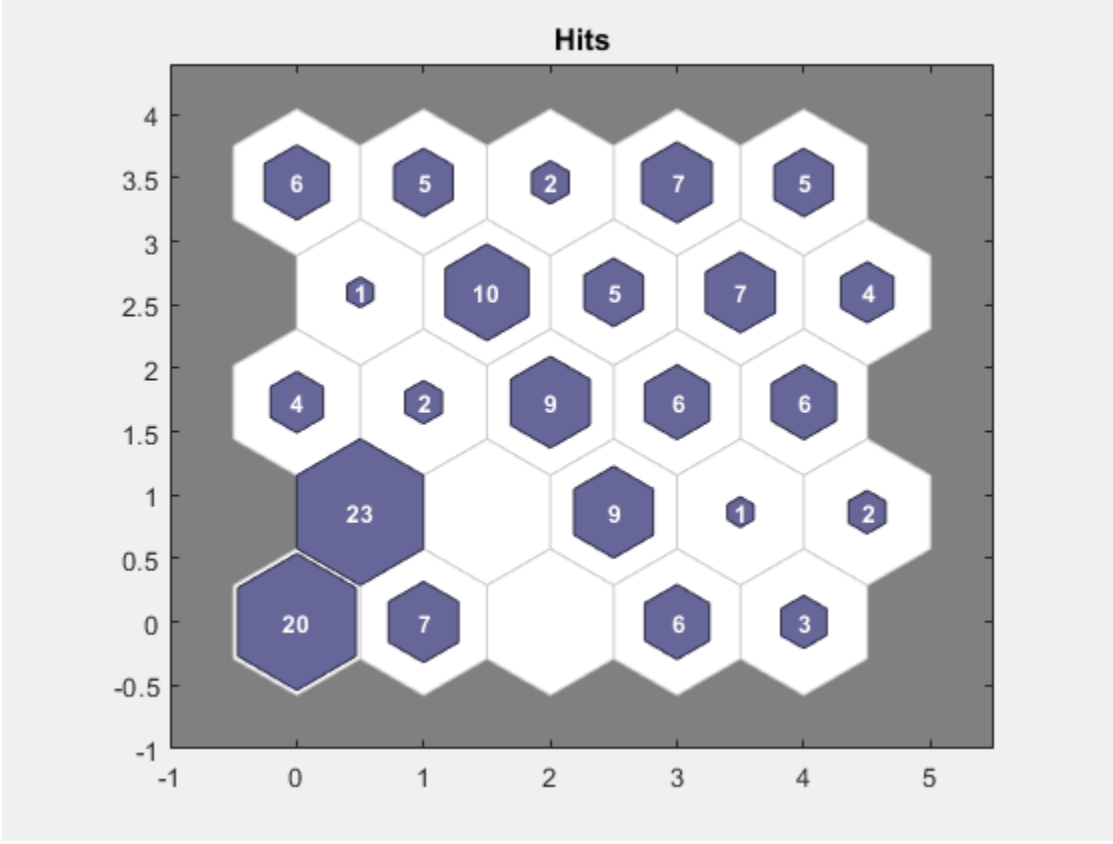
`plotsomhits(net,inputs)` plots a `SOM` layer, with each neuron showing the number of input vectors that it classifies. The relative number of vectors for each neuron is shown via the size of a colored patch.

This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

Examples

Plot SOM Sample Hits

```
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

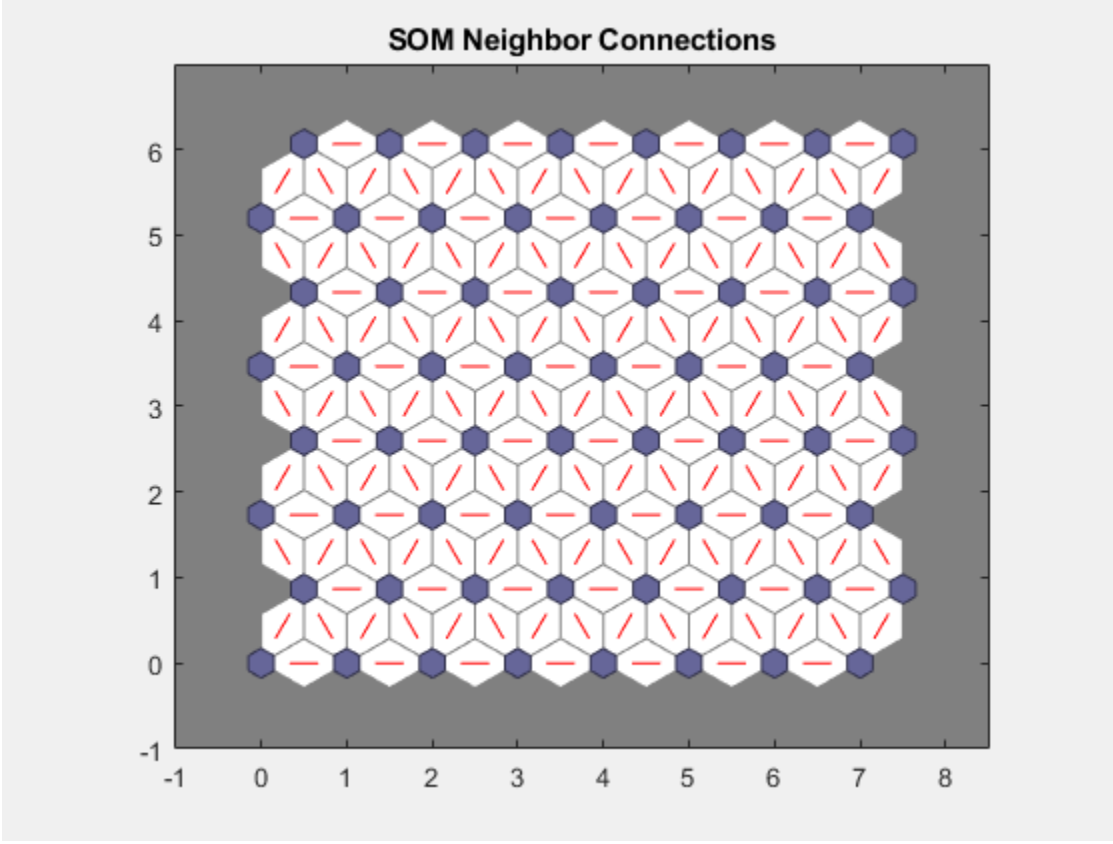
`plotsomnc(net)` plots a `SOM` layer showing neurons as gray-blue patches and their direct neighbor relations with red lines.

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 `i`th subplot shows the weights from the `i`th input to the layer's neurons, with the most negative connections shown as blue, zero connections as black, and the strongest positive connections as red.

The plot is only shown for layers organized in one or two dimensions.

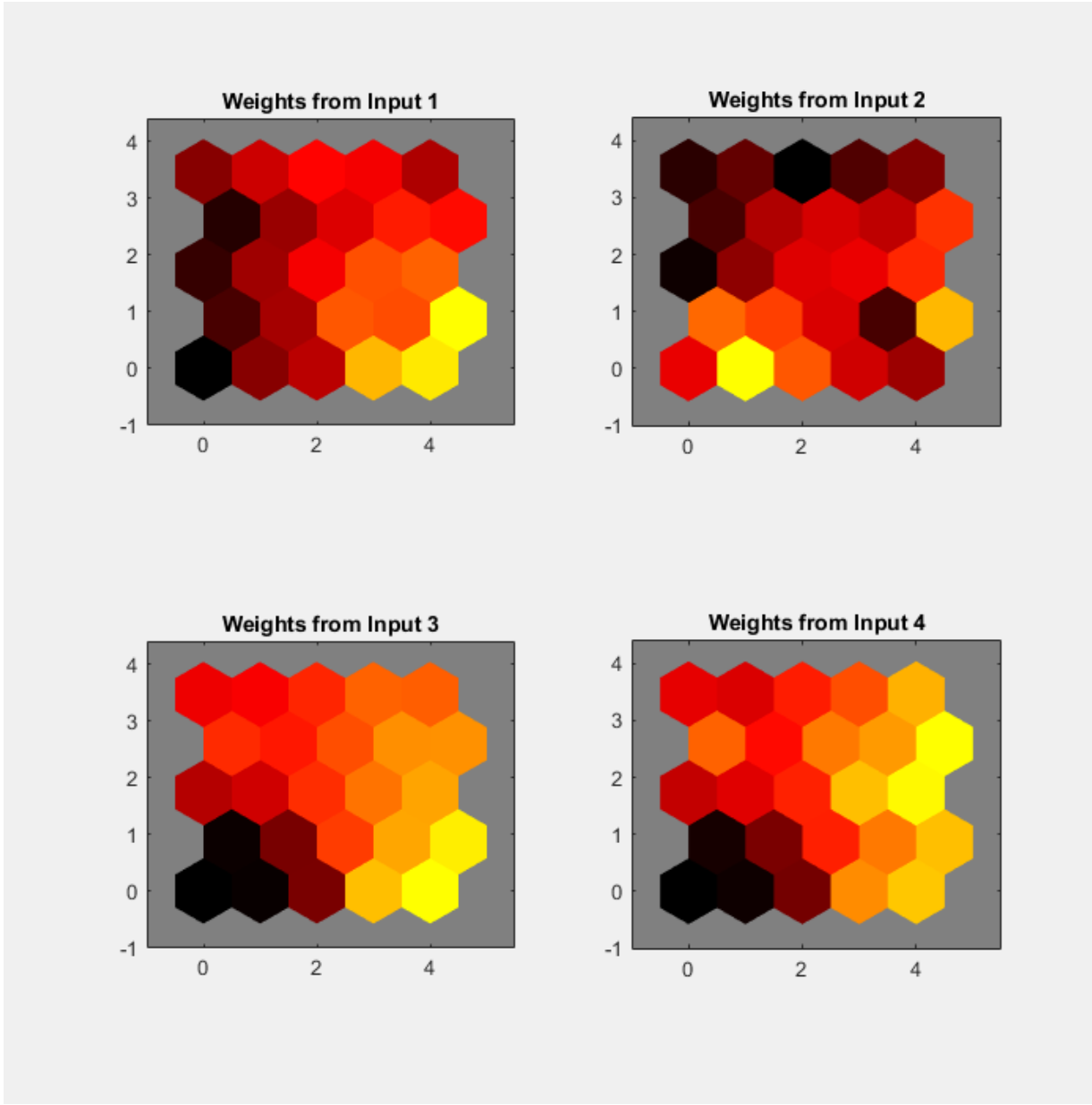
This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

This function can also be called with standardized plotting function arguments used by the function `train`.

Examples

Plot SOM Weight Planes

```
x = iris_dataset;
net = selforgmap([5 5]);
net = train(net,x);
plotsomplanes(net)
```


Weights from Input 1
Weights from Input 2
Weights from Input 3
Weights from Input 4
4
3
2
1
0
-1
0
2
4

See Also

`plotsomhits` | `plotsomnc` | `plotsomnd`

Introduced in R2008a

plotsompos

Plot self-organizing map weight positions

Syntax

```
plotsompos(net)
plotsompos(net,inputs)
```

Description

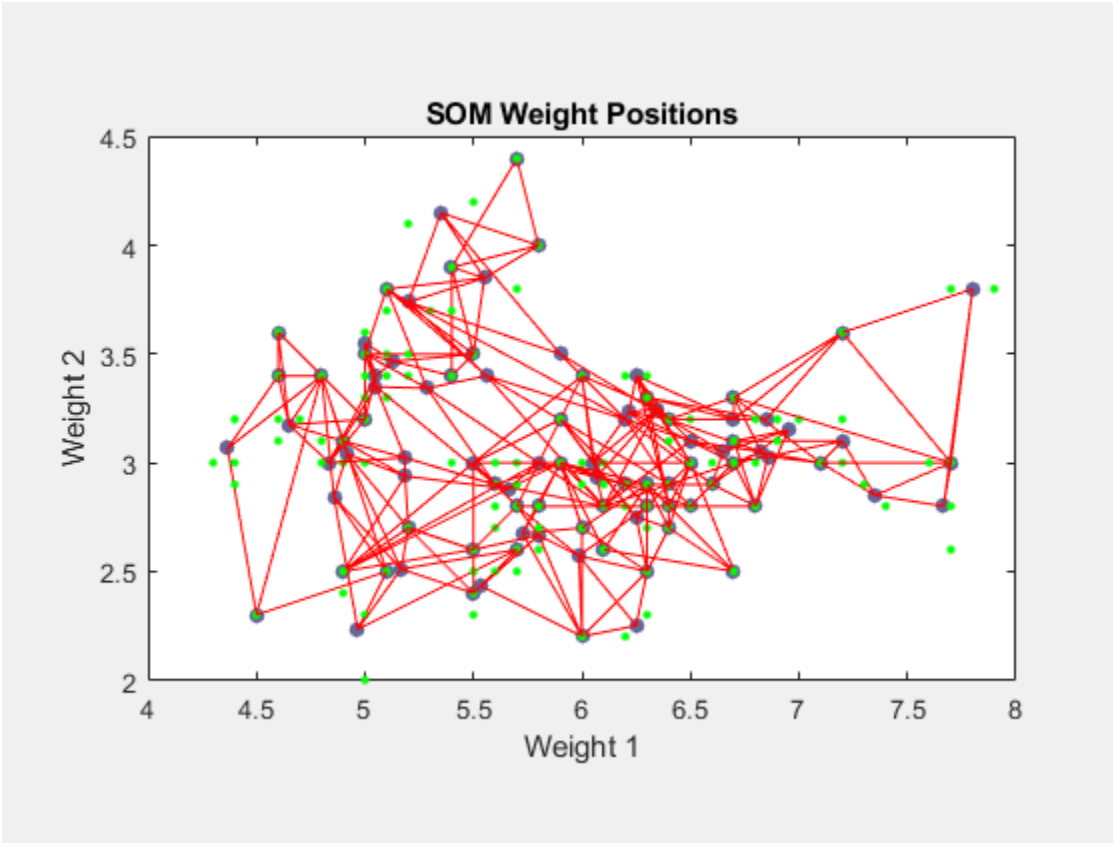
`plotsompos(net)` plots the input vectors as green dots and shows how the `SOM` classifies the input space by showing blue-gray dots for each neuron’s weight vector and connecting neighboring neurons with red lines.

`plotsompos(net,inputs)` plots the input data alongside the weights.

Examples

Plot SOM Weight Positions

```
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

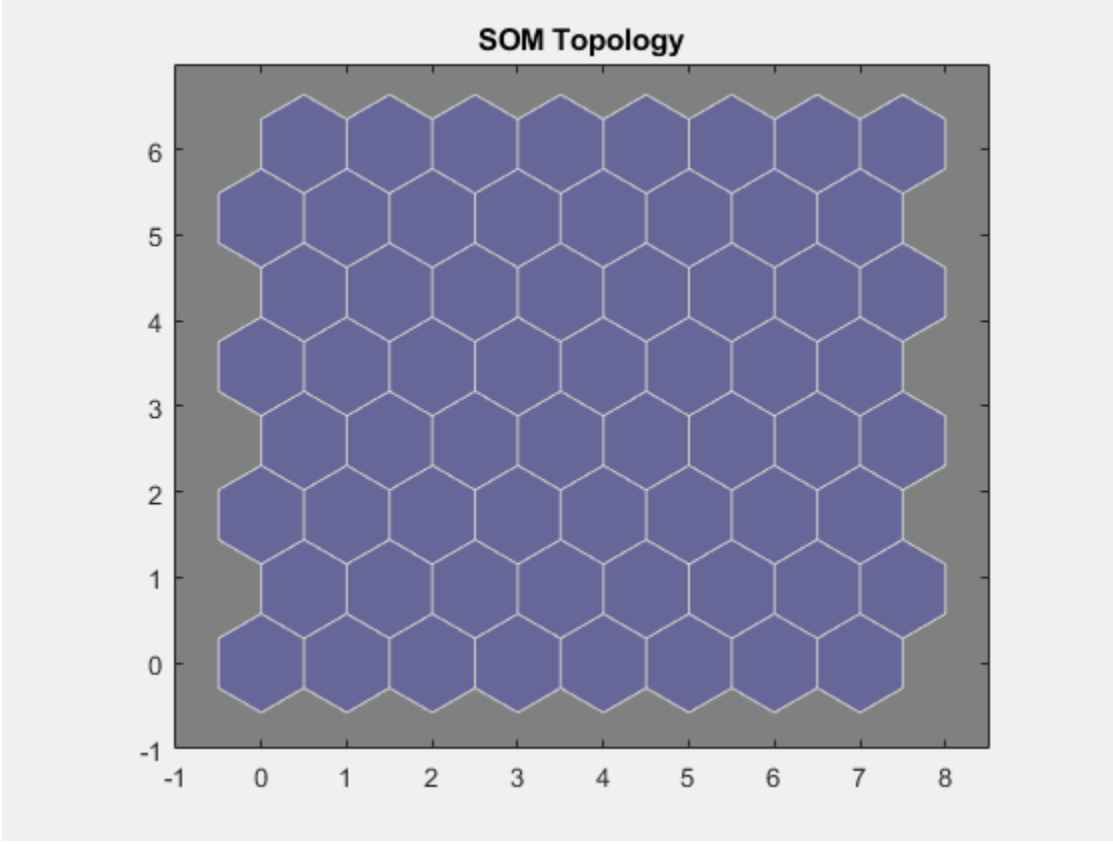
`plotsomtop(net)` plots the topology of a `SOM` layer.

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

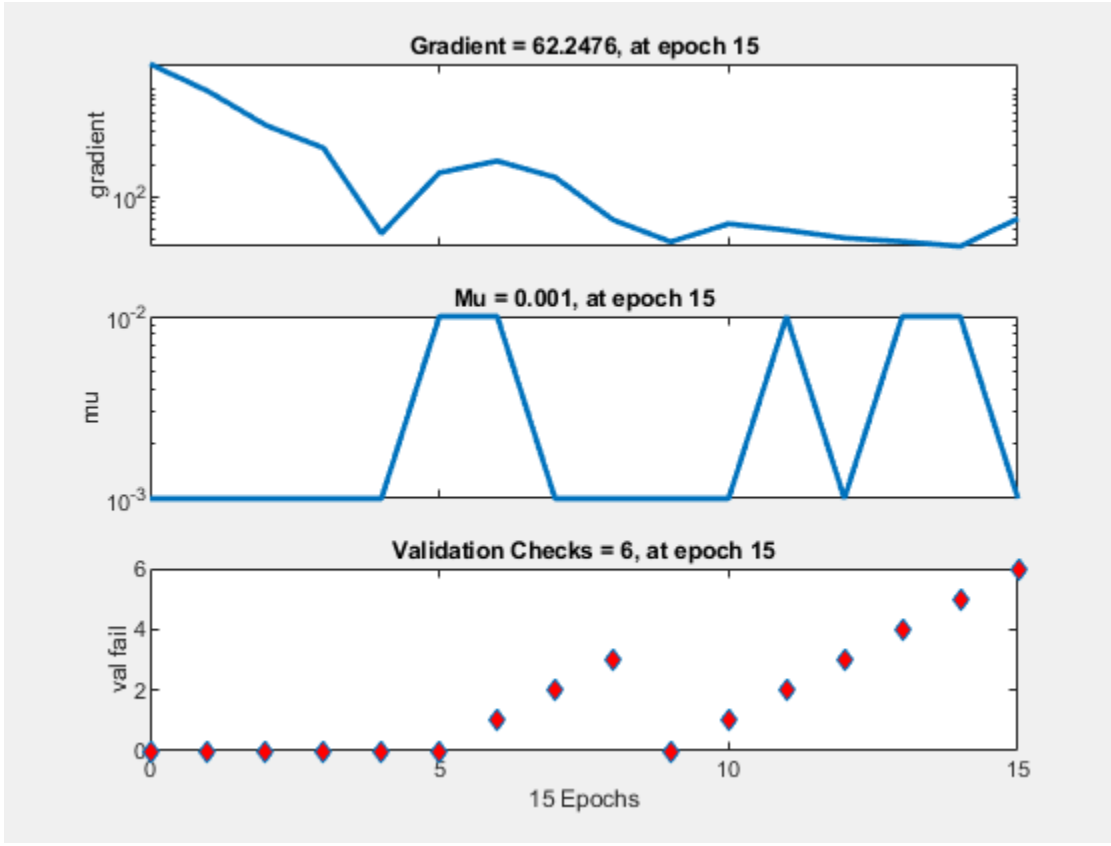
`plottrainstate(tr)` plots the training state from a training record `tr` returned by `train`.

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,

`M`	R-by-Q matrix of Q column vectors with R elements
`T`	The line plotting type (optional; default = `'-'`)

and plots the column vectors of `M`.

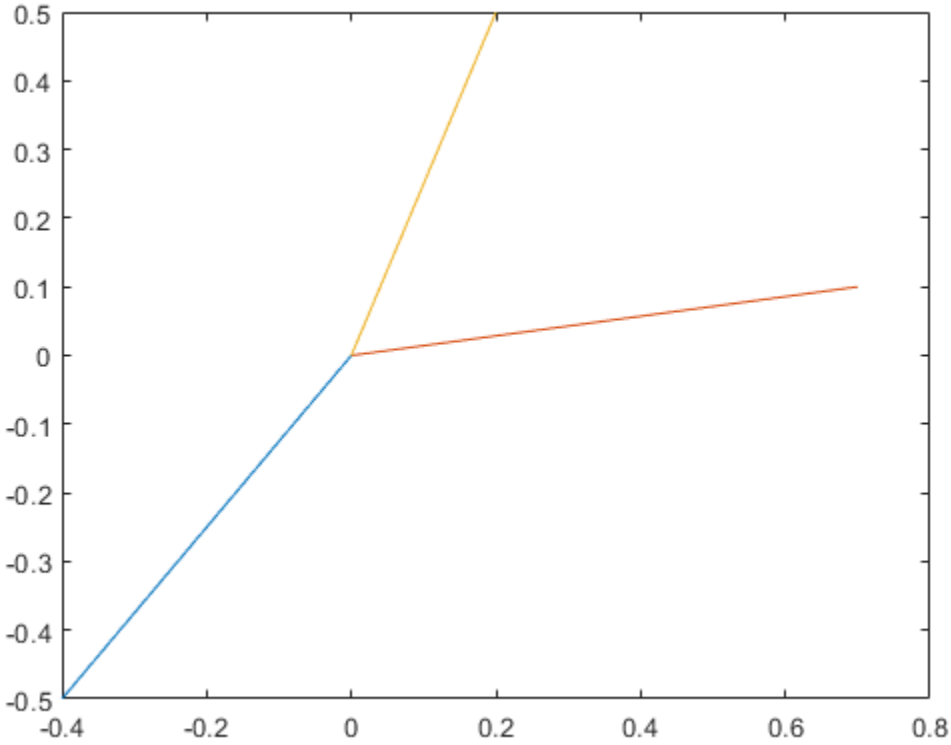
R must be 2 or greater. If R is greater than 2, only the first two rows of `M` are used for the plot.

Examples

Plot Vectors

This example shows how to plot three 2-element vectors.

```
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,

`X`	Matrix of (column) vectors
`C`	Row vector of color coordinates
`M`	Marker (default = `'+'`)

and plots each `i`th vector in `X` with a marker `M`, using the `i`th value in `C` as the color coordinate.

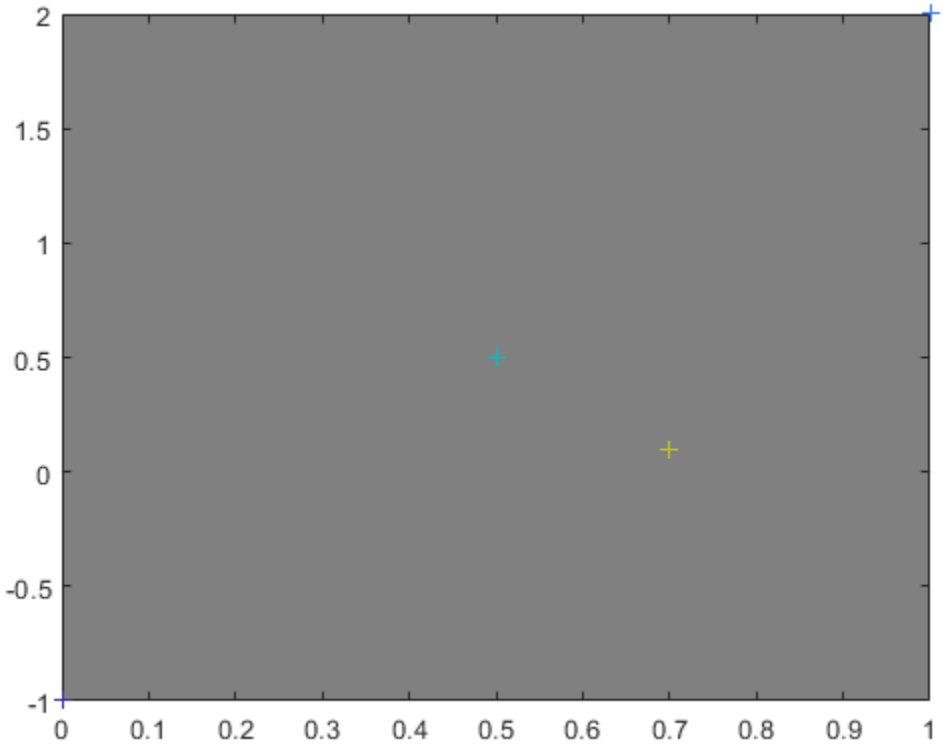
`plotvec(X)` only takes a matrix `X` and plots each `i`th vector in `X` with marker `'+'` using the index `i` as the color coordinate.

Examples

Plot Vectors with Different Colors

This example shows how to plot four 2-element vectors.

```
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.

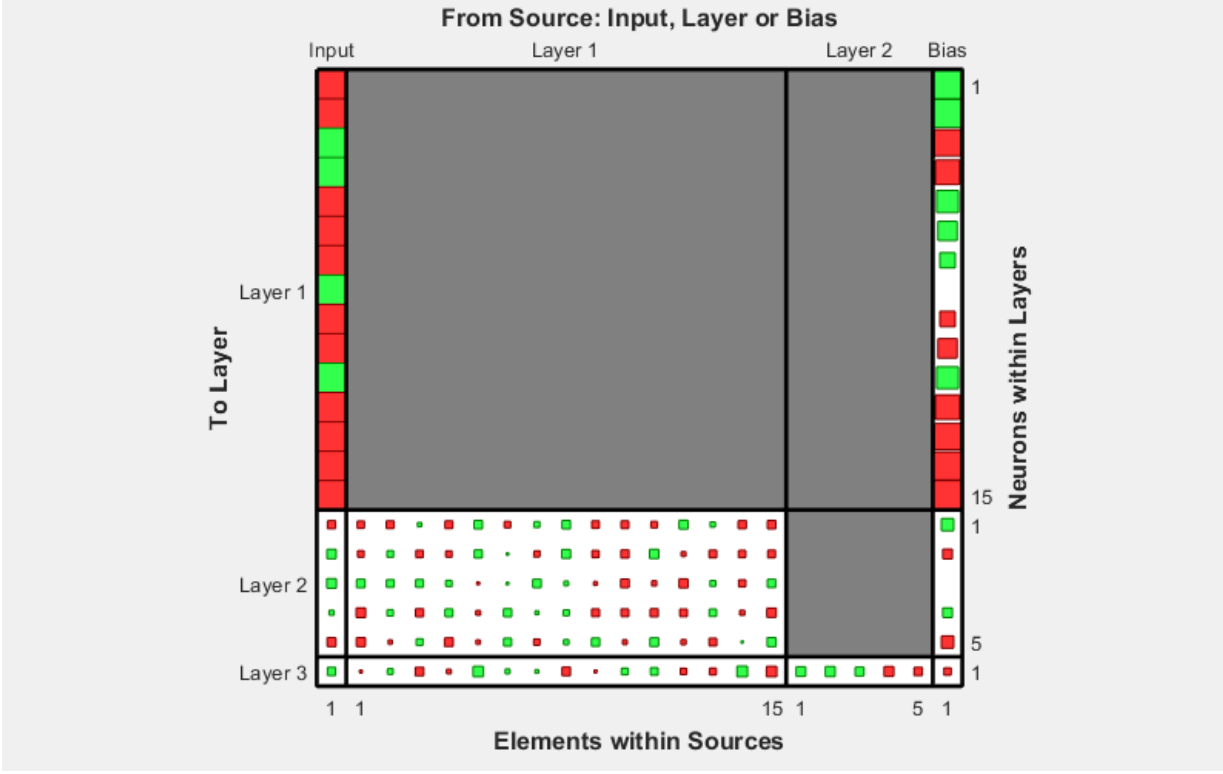
Examples

Plot Weights and Biases

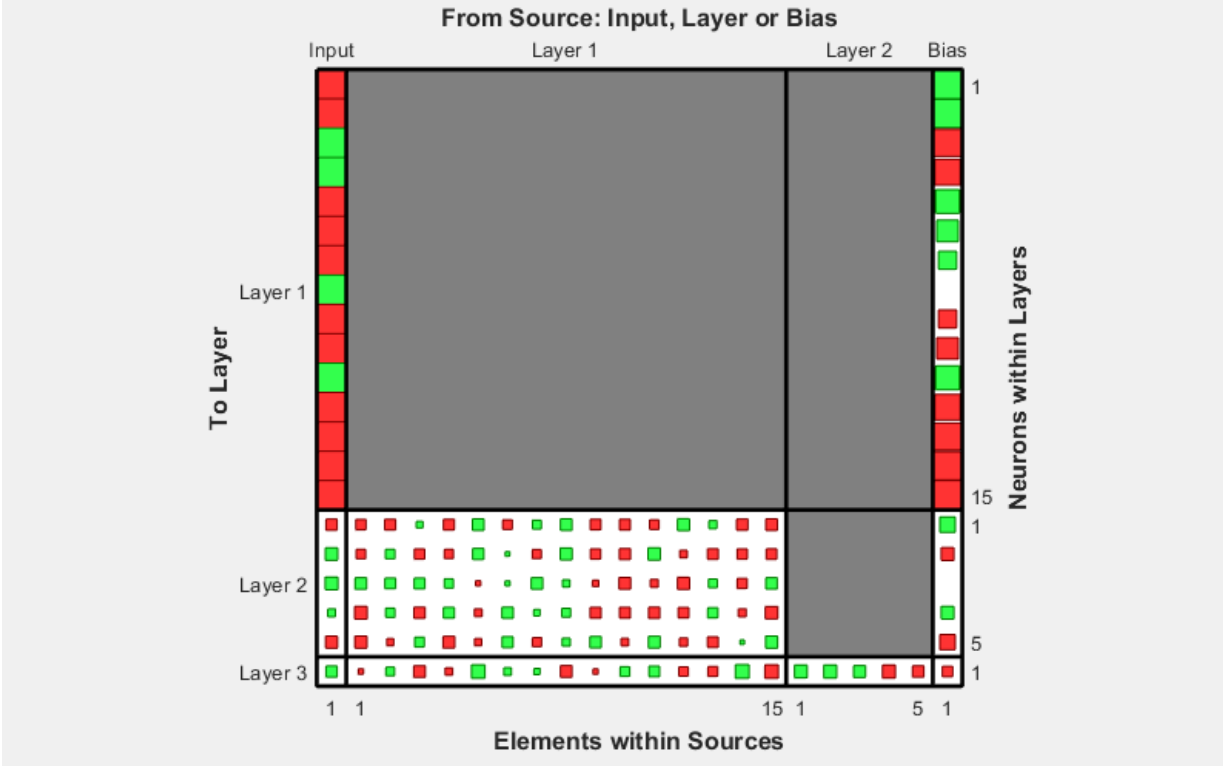
Here a cascade-forward network is configured for particular data and its weights and biases are plotted in several ways.

```
[x,t] = simplefit_dataset;
net = cascadeforwardnet([15 5]);
net = configure(net,x,t);
plotwb(net)
```

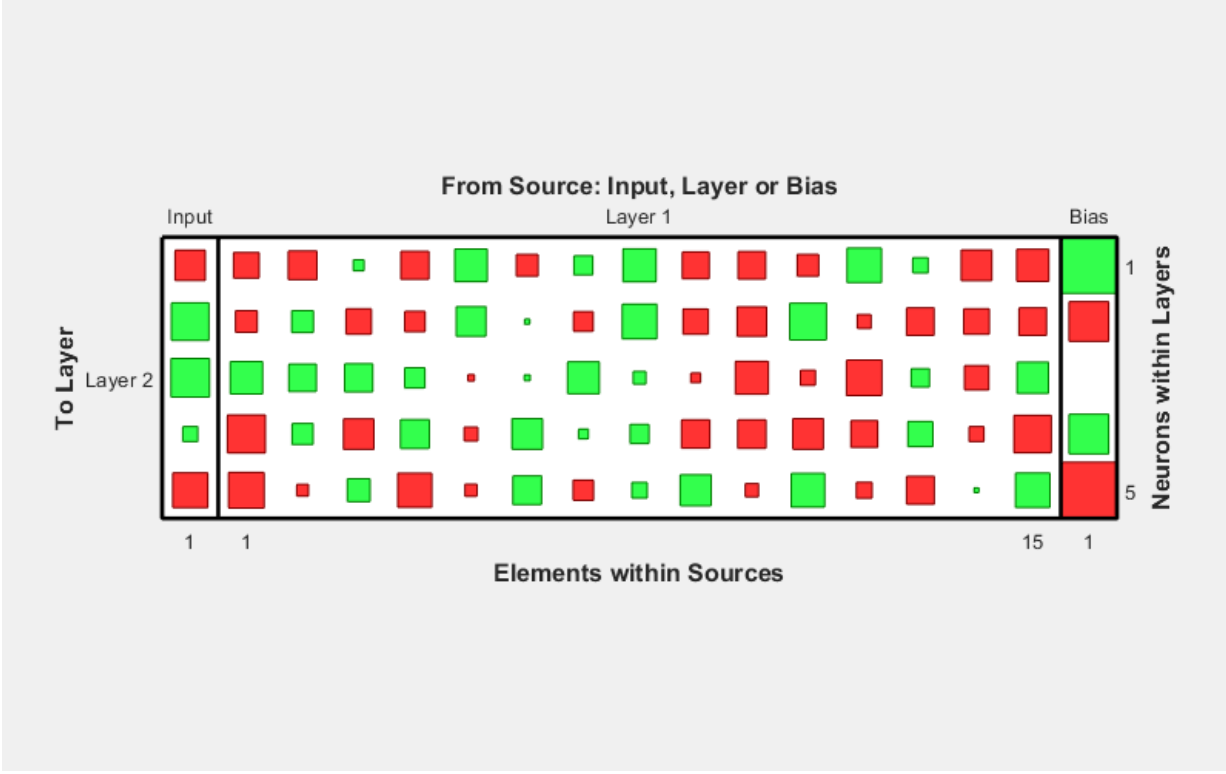

```
plotwb(net,'root',3)
```



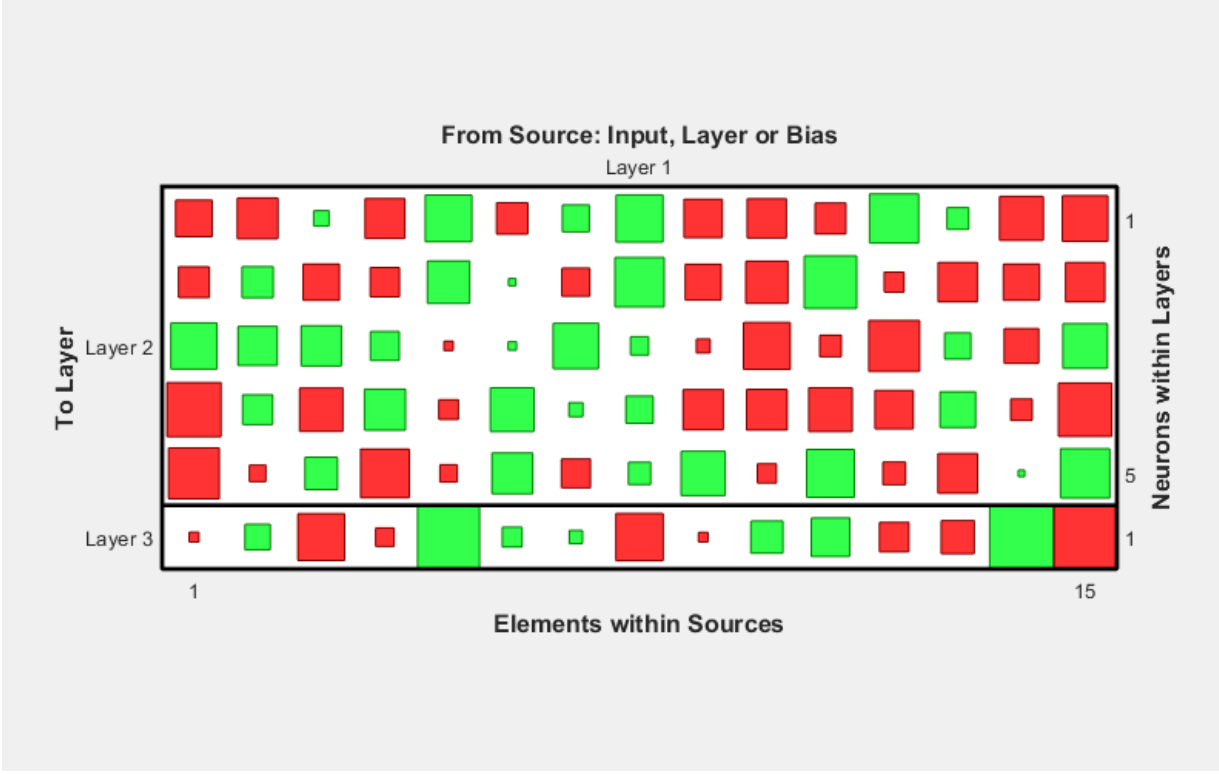

```
plotwb(net,'root',4)
```



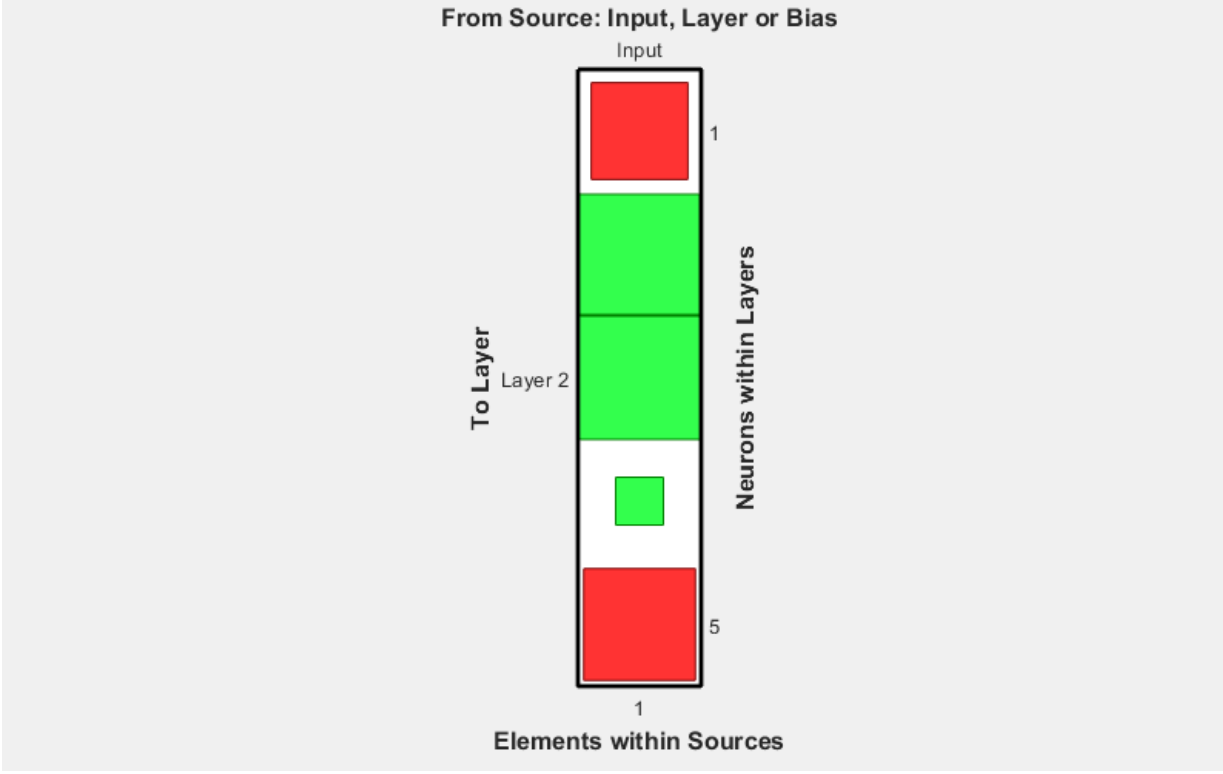

```
plotwb(net,'toLayers',2)
```




```
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,

X	M-by-N matrix
R	(Optional) radius to normalize columns to (default = 1)

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

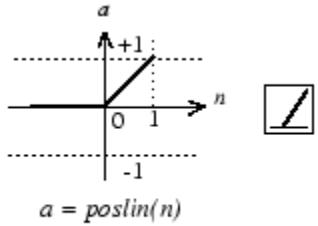
normc | normr

Introduced before R2006a

poslin

Positive linear transfer function

Graph and Symbol



Positive Linear Transfer Function

Syntax

```
A = poslin(N,FP)
info = poslin('code')
```

Description

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

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

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, the S-by-Q matrix of `N`'s elements clipped to `[0, inf]`.

`info = poslin('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.

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

Assign this transfer function to layer `i` of a network.

```
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.

```
poslin(n) = n, if n >= 0
          = 0, if n <= 0
```

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] = preparets(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 `adddelay`, this function can reformat the data accordingly.

`[Xs,Xi,Ai,Ts,EWs,shift] = preparets(net,Xnf,Tnf,Tf,EW)` takes these arguments,

`net`	Neural network
`Xnf`	Non-feedback inputs
`Tnf`	Non-feedback targets
`Tf`	Feedback targets
`EW`	Error weights (default = `{1}`)

and returns,

`Xs`	Shifted inputs
`Xi`	Initial input delay states

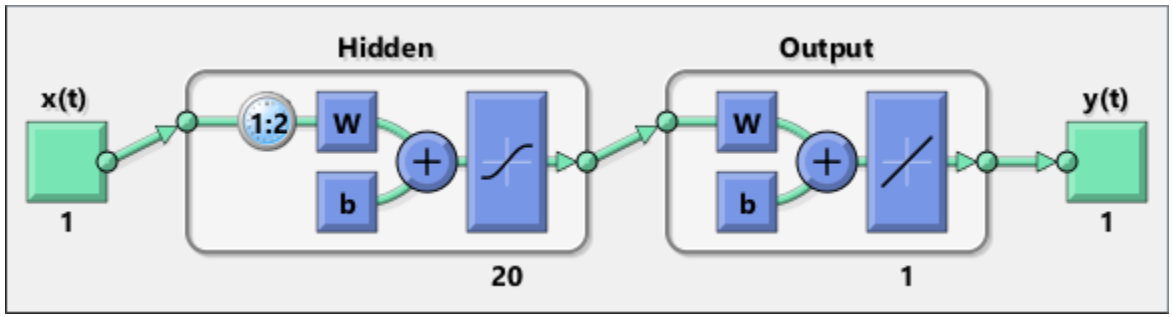
`Ai`	Initial layer delay states
`Ts`	Shifted targets
`EWs`	Shifted error weights
`shift`	The number of timesteps truncated from the front of `X` and `T` in order to properly fill `Xi` and `Ai`.

Examples

Prepare Data for Open- and Closed-Loop Networks

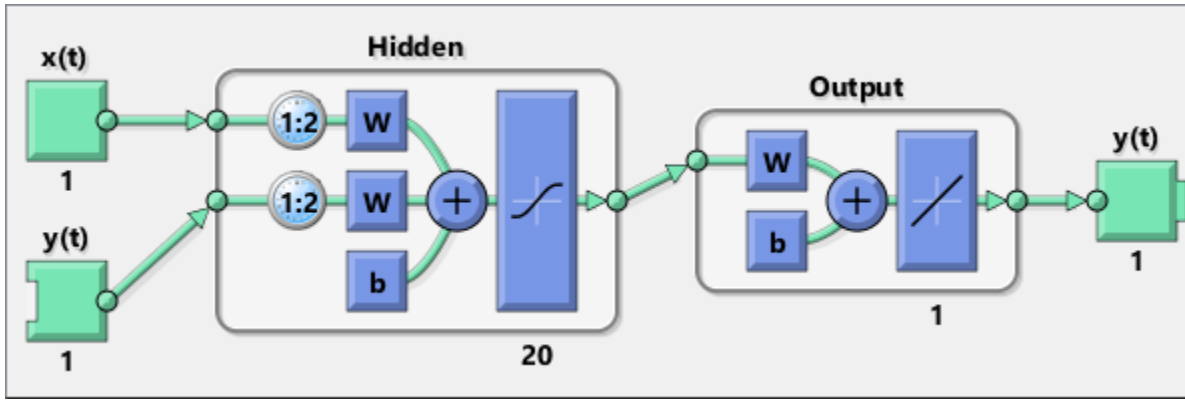
Here a time-delay network with 20 hidden neurons is created, trained and simulated.

```
[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);
```



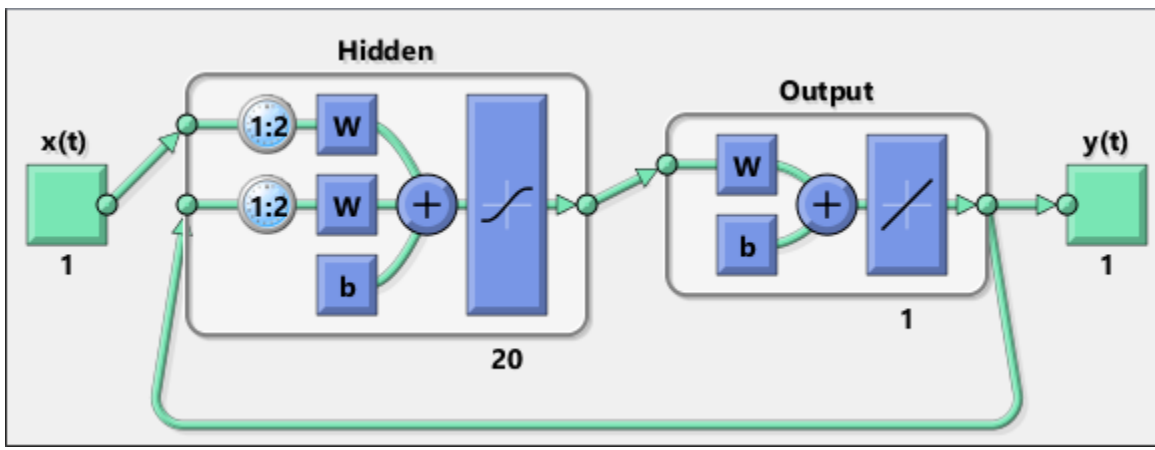
Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```
[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.

```
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] = processpca(X,maxfrac)
[Y,PS] = processpca(X,FP)
Y = processpca('apply',X,PS)
X = processpca('reverse',Y,PS)
name = processpca('name')
fp = processpca('pdefaults')
names = processpca('pdesc')
processpca('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] = processpca(X,maxfrac)` takes `X` and an optional parameter,

`X`	`N`-by-`Q` matrix
`maxfrac`	Maximum fraction of variance for removed rows (default is 0)

and returns

`Y`	`M`-by-`Q` matrix with `N - M` rows deleted
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = processpca(X,FP)` takes parameters as a struct: `FP.maxfrac`.

`Y = processpca('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = processpca('reverse',Y,PS)` returns `X`, given `Y` and settings `PS`.

`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.

```
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.

```
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.

```
x1_again = processpca('reverse',y1,ps)
```

Definitions

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`.

```
[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.

```
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:

```
net.inputs{1}.processFcns{end+1} = 'processpca';
```

Algorithms

Values in rows whose elements are not all the same value are set to

```
y = 2*(x-minx)/(maxx-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

```
[net,pi,pl,po] = prune(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.

`[net,pi,pl,po] = prune(net)` takes a neural network and returns

`net`	The same network with zero-sized subobjects removed
`pi`	Indices of pruned inputs
`pl`	Indices of pruned layers
`po`	Indices of pruned outputs

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` | `prunedata`

Introduced in R2010b

prunedata

Prune data for consistency with pruned network

Syntax

```
[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,

`pi`	Indices of pruned inputs
`pl`	Indices of pruned layers
`po`	Indices of pruned outputs
`X`	Input data
`Xi`	Initial input delay states
`Ai`	Initial layer delay states
`T`	Target data

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

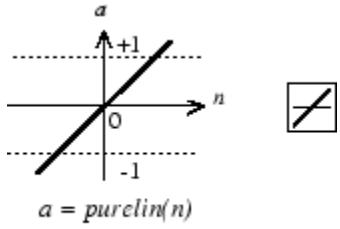
`gensim` | `prune`

Introduced in R2010b

purelin

Linear transfer function

Graph and Symbol



Linear Transfer Function

Syntax

```
A = purelin(N,FP)
info = purelin('code')
```

Description

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

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

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, an S-by-Q matrix equal to `N`.

`info = purelin('code')` returns useful information for each supported *`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 `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.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

```
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,

`X`	Matrix, vector, or scalar
`Q`	Minimum value

and returns values from `X` rounded to nearest multiple of `Q`.

Examples

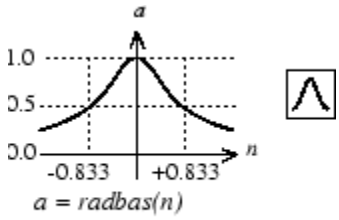
```
x = [1.333 4.756 -3.897];
y = quant(x,0.1)
```

Introduced before R2006a

radbas

Radial basis transfer function

Graph and Symbol



Radial Basis Function

Syntax

```
A = radbas(N,FP)
```

Description

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

`A = radbas(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 you create a plot of the `radbas` transfer function.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

```
a = radbas(n) = exp(-n^2)
```

See Also

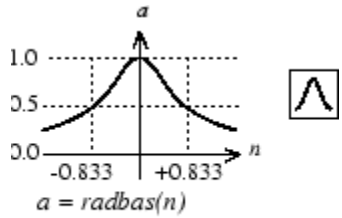
`radbasn` | `sim` | `tribas`

Introduced before 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 = rand(3,6)
a = radbasn(n)
```

Assign this transfer function to layer `i` of a network.

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

Algorithms

```
a = 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 = randnc(S,PR)
```

Description

`randnc` is a weight initialization function.

`W = randnc(S,PR)` takes two inputs,

S	Number of rows (neurons)
PR	R-by-2 matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R random matrix with normalized columns.

You can also call this in the form `randnc(S,R)`.

Examples

A random matrix of four normalized three-element columns is generated:

```
M = randnc(3,4)
M =
    -0.6007   -0.4715   -0.2724    0.5596
    -0.7628   -0.6967   -0.9172    0.7819
    -0.2395    0.5406   -0.2907    0.2747
```

See Also

`randnr`

Introduced before R2006a

randnr

Normalized row weight initialization function

Syntax

```
W = randnr(S,PR)
```

Description

`randnr` is a weight initialization function.

`W = randnr(S,PR)` takes two inputs,

S	Number of rows (neurons)
PR	R-by-2 matrix of input value ranges = `[Pmin Pmax]`

and returns an S-by-R random matrix with normalized rows.

You can also call this in the form `randnr(S,R)`.

Examples

A matrix of three normalized four-element rows is generated:

```
M = randnr(3,4)
M =
    0.9713    0.0800   -0.1838   -0.1282
    0.8228    0.0338    0.1797    0.5381
   -0.3042   -0.5725    0.5436    0.5331
```

See Also

`randnc`

Introduced before R2006a

rands

Symmetric random weight/bias initialization function

Syntax

```
W = rands(S,PR)
M = rands(S,R)
v = rands(S)
```

Description

`rands` is a weight/bias initialization function.

`W = rands(S,PR)` takes

S	Number of neurons
PR	R-by-2 matrix of R input ranges

and returns an S-by-R weight matrix of random values between –1 and 1.

`M = rands(S,R)` returns an S-by-R matrix of random values. `v = rands(S)` returns an S-by-1 vector of random values.

Examples

Here, three sets of random values are generated with `rands`.

```
rands(4,[0 1; -2 2])
rands(4)
rands(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 `'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 = randsmall(S,PR)
M = rands(S,R)
v = rands(S)
```

Description

`randsmall` is a weight/bias initialization function.

`W = randsmall(S,PR)` takes

S	Number of neurons
PR	R-by-2 matrix of R input ranges

and returns an S-by-R weight matrix of small random values between –0.1 and 0.1.

`M = rands(S,R)` returns an S-by-R matrix of random values. `v = rands(S)` returns an S-by-1 vector of random values.

Examples

Here three sets of random values are generated with `rands`.

```
randsmall(4,[0 1; -2 2])
randsmall(4)
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:

`dimensions`	Row vector of dimension sizes

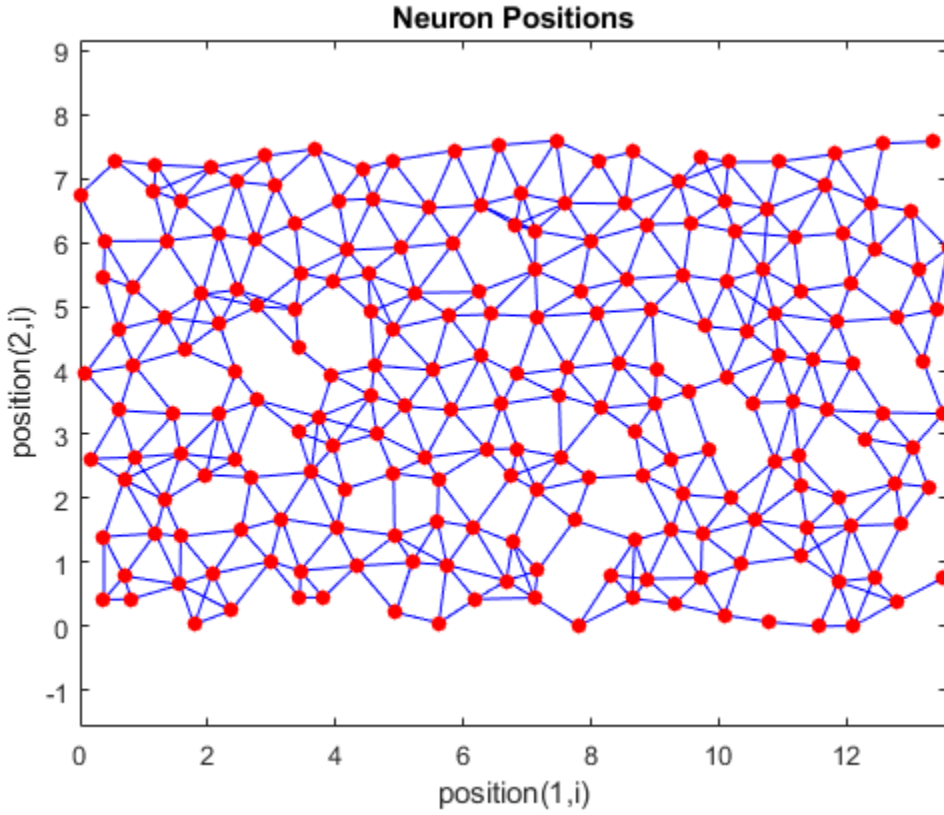
and returns an `N`-by-`S` matrix of `N` coordinate vectors, where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with 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] = regression(t,y)
[r,m,b] = regression(t,y,'one')
```

Description

`[r,m,b] = regression(t,y)` takes these arguments,

`t`	Target matrix or cell array data with a total of `N` matrix rows
`y`	Output matrix or cell array data of the same size

and returns these outputs,

`r`	Regression values for each of the `N` matrix rows
`m`	Slope of regression fit for each of the `N` matrix rows
`b`	Offset of regression fit for each of the `N` matrix rows

`[r,m,b] = regression(t,y,'one')` combines all matrix rows before regressing, and returns single scalar regression, slope, and offset values.

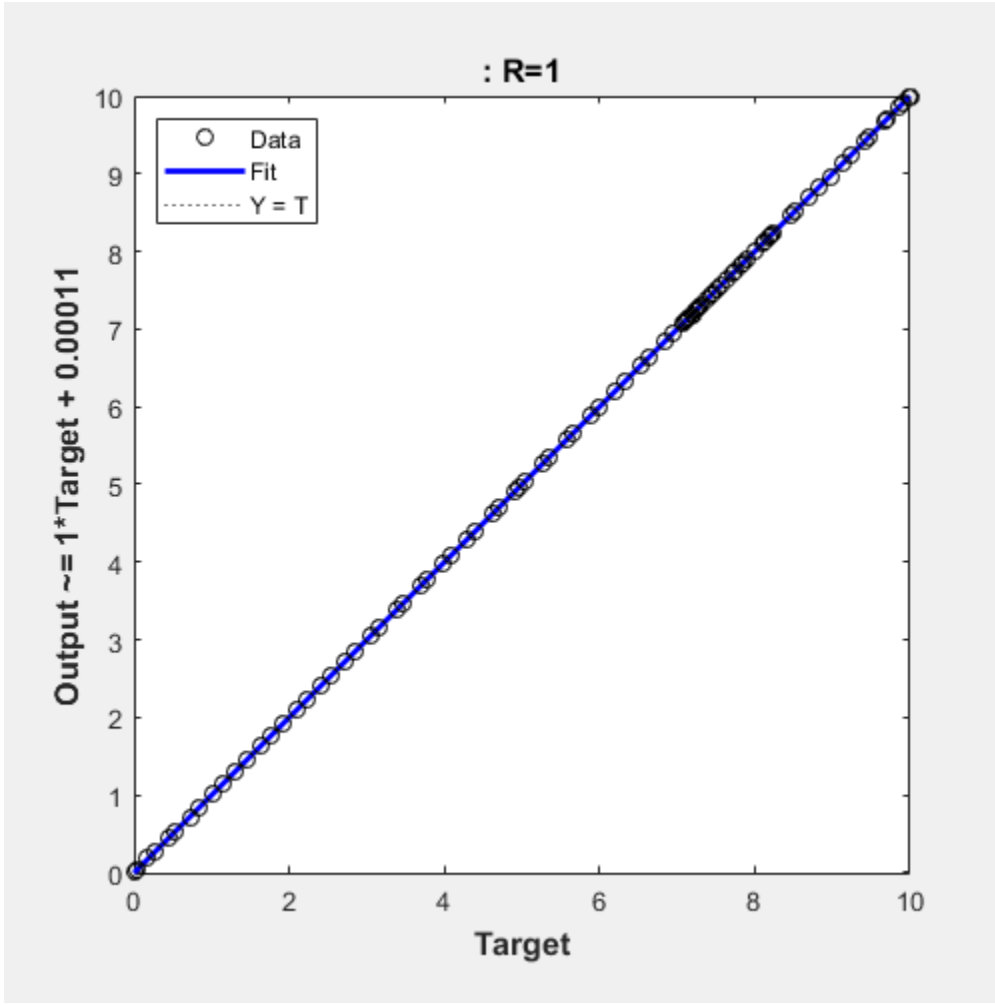
Examples

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] = simplefit_dataset;
net = 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

```
[Y,PS] = removeconstantrows(X,max_range)
[Y,PS] = removeconstantrows(X,FP)
Y = removeconstantrows('apply',X,PS)
X = removeconstantrows('reverse',Y,PS)
```

Description

`removeconstantrows` processes matrices by removing rows with constant values.

`[Y,PS] = removeconstantrows(X,max_range)` takes `X` and an optional parameter,

`X`	`N`-by-`Q` matrix
`max_range`	Maximum range of values for row to be removed (default is 0)

and returns

`Y`	`M`-by-`Q` matrix with `N - M` rows deleted
`PS`	Process settings that allow consistent processing of values

`[Y,PS] = removeconstantrows(X,FP)` takes parameters as a struct: `FP.max_range`.

`Y = removeconstantrows('apply',X,PS)` returns `Y`, given `X` and settings `PS`.

`X = 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 NaN; 1 1 1 2]`.

Examples

Format a matrix so that the rows with constant values are removed.

```
x1 = [1 2 4; 1 1 1; 3 2 2; 0 0 0];
[y1,PS] = removeconstantrows(x1);
```

```
y1 =
     1     2     4
     3     2     2

PS =
    max_range: 0
         keep: [1 3]
       remove: [2 4]
        value: [2x1 double]
        xrows: 4
        yrows: 2
    constants: [2x1 double]
    no_change: 0
```

Next, apply the same processing settings to new values.

```
x2 = [5 2 3; 1 1 1; 6 7 3; 0 0 0];
y2 = removeconstantrows('apply',x2,PS)
```

```
5     2     3
6     7     3
```

Reverse the processing of `y1` to get the original `x1` matrix.

```
x1_again = 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

```
net = removedelay(net,n)
```

Description

`net = removedelay(net,n)` takes these arguments,

`net`	Neural network
`n`	Number of delays

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.

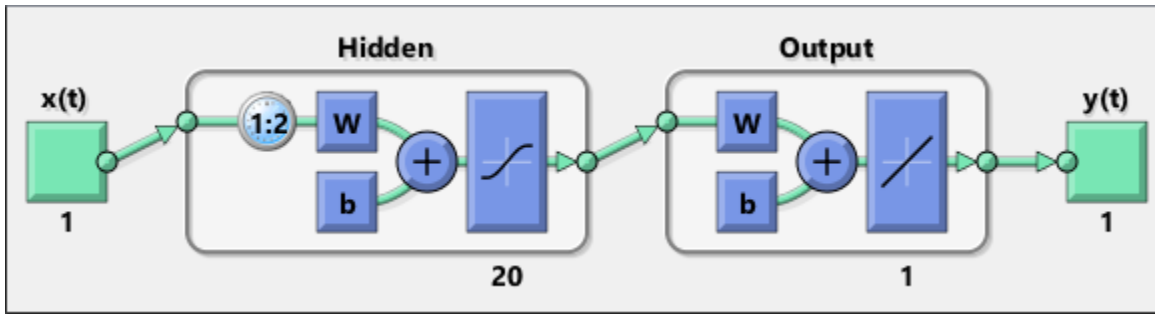
Examples

Remove and Add Delay to Network

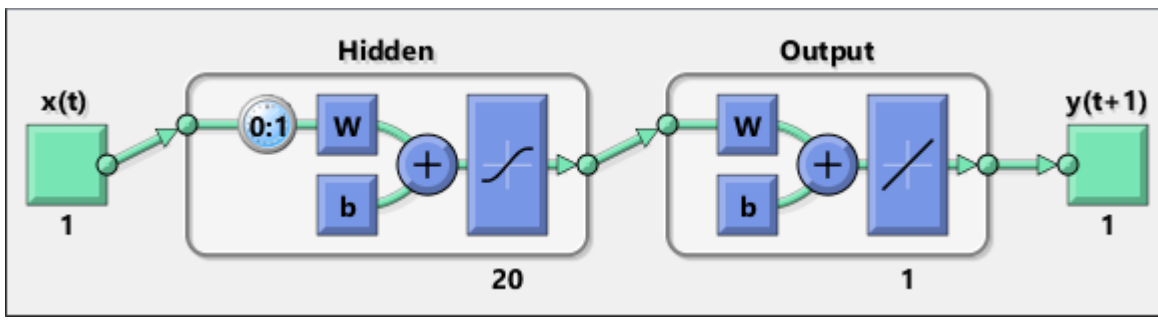
This example creates, trains, and simulates a time delay network in its original form, on an input time series `X` and target series `T`. Then the delay is removed and later added back. The first and third outputs will be identical, while the second result will include a new prediction for the following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

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

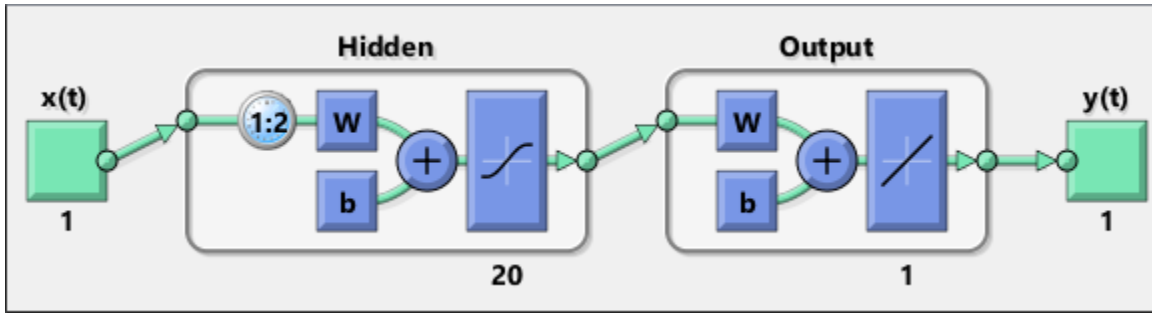
```
y1 = net1(Xs,Xi);
view(net1)
```




```
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,

`X`	`N`-by-`Q` matrix
`ind`	Vector of row indices to remove (default is `[]`)

and returns

`Y`	`M`-by-`Q` matrix, where `M == N-length(ind)`
`PS`	Process settings that allow consistent processing of values

`[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

```
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.

```
net = perceptron;
net.inputs{1}.size = 2;
net.layers{1}.size = 1;
```

The initial network has weights and biases with zero values.

```
net.iw{1,1}, net.b{1}
```

Change these values as follows:

```
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

```
[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`.

`[tpr,fpr,thresholds] = roc(targets,outputs)` takes these arguments:

`targets`	`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.
`outputs`	`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.

and returns these values:

`tpr`	1-by-`S` cell array of 1-by-`N` true-positive/positive ratios.
`fpr`	1-by-`S` cell array of 1-by-`N` false-positive/negative ratios.
`thresholds`	1-by-`S` cell array of 1-by-`N` thresholds over interval `[0,1]`.

`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

```
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,

`net`	Neural network
`t`	Matrix or cell array of target vectors
`y`	Matrix or cell array of output vectors
`ew`	Error weights (default = `{1}`)

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)`

```
[...] = sae(...,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 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

```
[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.

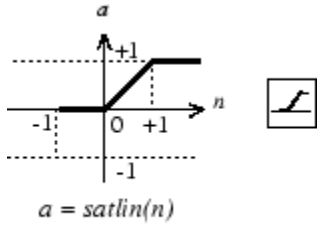
Then calling `train`, `adapt` or `perform` will result in `sae` being used to calculate performance.

Introduced in R2010b

satlin

Saturating linear transfer function

Graph and Symbol



Satlin Transfer Function

Syntax

```
A = satlin(N,FP)
```

Description

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

`A = satlin(N,FP)` takes one input,

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

and returns `A`, the `S`-by-`Q` matrix of `N`'s elements clipped to `[0, 1]`.

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

`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 `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.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

```
a = satlin(n) = 0, if n <= 0
n, if 0 <= n <= 1
1, if 1 <= n
```

See Also

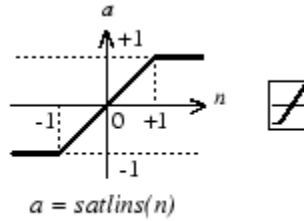
`poslin` | `purelin` | `satlins` | `sim`

Introduced before R2006a

satlins

Symmetric saturating linear transfer function

Graph and Symbol



Satlins Transfer Function

Syntax

```
A = satlins(N,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,

`N`	`S`-by-`Q` matrix of net input (column) vectors
`FP`	Struct of function parameters (optional, ignored)

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 = scalprod(W,P)
dim = scalprod('size',S,R,FP)
dw = scalprod('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 = scalprod(W,P)` takes these inputs,

`W`	1-by-1 weight matrix
`P`	R-by-Q matrix of Q input (column) vectors

and returns the R-by-Q scalar product of `W` and `P` defined by `Z = w*P`.

`dim = scalprod('size',S,R,FP)` takes the layer dimension `S`, input dimension `R`, and function parameters, and returns the weight size `[1-by-1]`.

`dw = scalprod('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(1,1);
P = rand(3,1);
Z = 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,distanceFc
n)
```

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,distanceFc n)` 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'`)

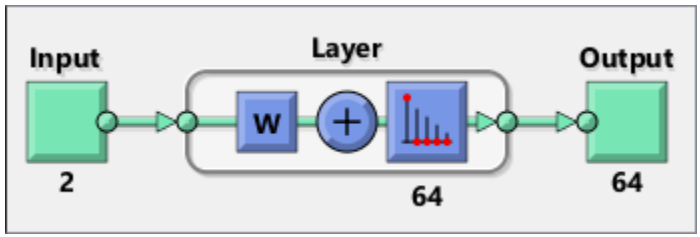
and returns a self-organizing map.

Examples

Use Self-Organizing Map to Cluster Data

Here a self-organizing map is used to cluster a simple set of data.

```
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] = separatewb(net,wb)
```

Description

`[b,IW,LW] = separatewb(net,wb)` takes two arguments,

`net`	Neural network
`wb`	Weight/bias vector

and returns

`b`	Cell array of bias vectors
`IW`	Cell array of input weight matrices
`LW`	Cell array of layer weight matrices

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] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
wb = formwb(net,net.b,net.iw,net.lw)
[b,iw,lw] = separatewb(net,wb)
```

See Also

`formwb` | `getwb` | `setwb`

Introduced in R2010b

seq2con

Convert sequential vectors to concurrent vectors

Syntax

```
b = seq2con(s)
```

Description

Neural Network Toolbox software represents batches of vectors with a matrix, and sequences of vectors with multiple columns of a cell array.

`seq2con` and `con2seq` allow concurrent vectors to be converted to sequential vectors, and back again.

`b = seq2con(s)` takes one input,

`s`	`N`-by-`TS` cell array of matrices with `M` columns

and returns

`b`	`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 = 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 = 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,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

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 = [1 2; 3 4; 7 4]
v = [10 11; 12 13];
y = 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 = 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,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

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:

```
x = [1 2 3; 4 7 4]
v = [10 11; 12 13];
y = setsamples(x,[1 3],v)
```

This code sets samples 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 = setsamples(x,[1 3],v)
```

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,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

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:

```
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,

`sysName`	The name of the Simulink system containing the neural network block
`netName`	The name of the Simulink neural network block
`net`	The original neural network
`xi`	Initial input delay states
`ai`	Initial layer delay states
`Q`	Sample number (default is 1)

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.

```
[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.

```
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.

```
[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`.)

```
[xi,ai] = getsiminit(sysName,netName,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,

`x`	Neural network matrix or cell array data
`i`	Indices
`v`	Neural network data to store into `x`

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 = 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

```
net = setwb(net,wb)
```

Description

This function sets a network’s weight and biases to a vector of values.

`net = setwb(net,wb)` takes the following inputs:

`net`	Neural network
`wb`	Vector of weight and bias values

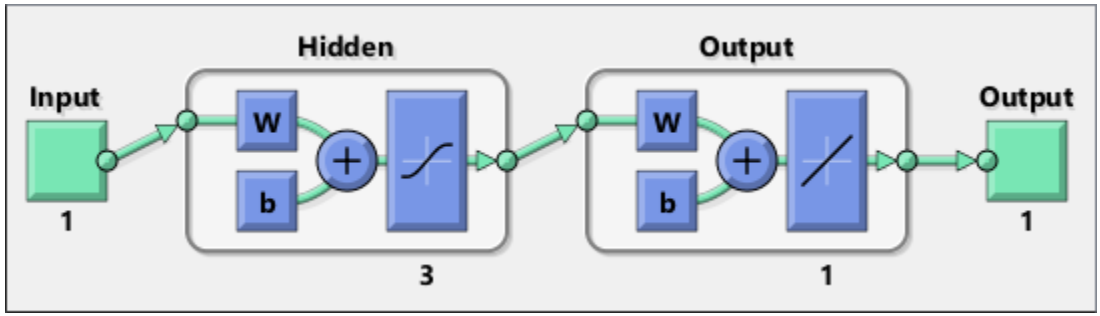
Examples

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] = simplefit_dataset;
net = feedforwardnet(3);
net = configure(net,x,t);
view(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.

```
net = setwb(net,rand(10,1));
```

View the weight and bias values

```
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] = sim(net,X,Xi,Ai,T)
[Y,Xf,Af] = sim(net,{Q TS},Xi,Ai)
[Y,...] = sim(net,...,'useParallel',...)
[Y,...] = sim(net,...,'useGPU',...)
[Y,...] = sim(net,...,'showResources',...)
[Ycomposite,...] = sim(net,Xcomposite,...)
[Ygpu,...] = sim(net,Xgpu,...)
```

To Get Help

Type `help network/sim`.

Description

`sim` simulates neural networks.

`[Y,Xf,Af] = sim(net,X,Xi,Ai,T)` takes

`net`	Network
`X`	Network inputs
`Xi`	Initial input delay conditions (default = zeros)
`Ai`	Initial layer delay conditions (default = zeros)
`T`	Network targets (default = zeros)

and returns

`Y`	Network outputs

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 = sim(net,x,xi,ai)
y = 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:

X	Ni-by-TS cell array	Each element X{i,ts} is an Ri-by-Q matrix.
Xi	Ni-by-ID cell array	Each element Xi{i,k} is an Ri-by-Q matrix.
Ai	Nl-by-LD cell array	Each element Ai{i,k} is an Si-by-Q matrix.
T	No-by-TS cell array	Each element X{i,ts} is a Ui-by-Q matrix.
Y	No-by-TS cell array	Each element Y{i,ts} is a Ui-by-Q matrix.
Xf	Ni-by-ID cell array	Each element Xf{i,k} is an Ri-by-Q matrix.
Af	Nl-by-LD cell array	Each element Af{i,k} is an Si-by-Q matrix.

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 Xi, Ai, Xf, and Af are ordered from oldest delay condition to most recent:

Xi{i,k}	=	Input i at time ts = k - ID
Xf{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 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:

`'useParallel','no'`	Calculations occur on normal MATLAB thread. This is the default `'useParallel'` setting.
`'useParallel','yes'`	Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.
`'useGPU','no'`	Calculations occur on the CPU. This is the default 'useGPU' setting.
`'useGPU','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.
`'useGPU','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','no'`	Do not display computing resources used at the command line. This is the default setting.
`'showResources','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.

`[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.

```
[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 `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);
```

Simulate in Parallel on a Parallel Pool

Parallel Computing Toolbox allows Neural Network Toolbox to 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

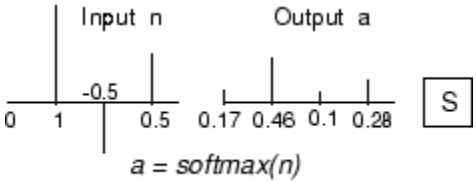
`nndata` | `nndata2sim`

Introduced in R2010b

softmax

Soft max transfer function

Graph and Symbol



Softmax Transfer Function

Syntax

```
A = softmax(N,FP)
```

Description

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

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

`N`	`S`-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`.

`info = softmax('code')` returns information about this function. The following codes are defined:

`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 = softmax(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 = 'softmax';
```

Algorithms

```
a = 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,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size
`tol`	Tolerance on search

`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	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
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the backstepping algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`low_lim`	Lower limit on change in step size
`up_lim`	Upper limit on change in step size
`maxstep`	Maximum step length
`minstep`	Minimum step length
`scale_tol`	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 `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`V`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

Definitions

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

Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ, Prentice-Hall, 1983

See Also

`srchcha` | `srchgol` | `srchhyb`

Introduced before R2006a

srchbre

1-D interval location using Brent’s method

Syntax

```
[a,gX,perf,retcode,delta,tol] =
srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)
```

Description

`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] = srchbre(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size

`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	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
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the Brent algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`bmax`	Largest step size
`scale_tol`	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 `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

Definitions

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

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

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.

`[a,gX,perf,retcode,delta,tol] = srchcha(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,tol,ch_perf)` takes these inputs,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size

`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	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
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the Charalambous algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`gama`	Parameter to avoid small reductions in performance, usually set to 0.1
`scale_tol`	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 `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

Definitions

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

Charalambous, C., “Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, Vol. 139, No. 3, June, 1992, pp. 301–310.

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,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size

`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	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
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the golden section algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`bmax`	Largest step size
`scale_tol`	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 `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.

`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

Definitions

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

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

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,

`net`	Neural network
`X`	Vector containing current values of weights and biases
`Pd`	Delayed input vectors
`Tl`	Layer target vectors
`Ai`	Initial input delay conditions
`Q`	Batch size
`TS`	Time steps
`dX`	Search direction vector
`gX`	Gradient vector
`perf`	Performance value at current `X`
`dperf`	Slope of performance value at current `X` in direction of `dX`
`delta`	Initial step size

`tol`	Tolerance on search
`ch_perf`	Change in performance on previous step

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point
`retcode`	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
`delta`	New initial step size, based on the current step size
`tol`	New tolerance on search

Parameters used for the hybrid bisection-cubic algorithm are

`alpha`	Scale factor that determines sufficient reduction in `perf`
`beta`	Scale factor that determines sufficiently large step size
`bmax`	Largest step size
`scale_tol`	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 `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

`Pd`	`No`-by-`Ni`-by-`TS` cell array	Each element `P{i,j,ts}` is a `Dij`-by-`Q` matrix.
`Tl`	`Nl`-by-`TS` cell array	Each element `P{i,ts}` is a `Vi`-by-`Q` matrix.
`Ai`	`Nl`-by-`LD` cell array	Each element `Ai{i,k}` is an `Si`-by-`Q` matrix.

where

`Ni`	=	`net.numInputs`
`Nl`	=	`net.numLayers`
`LD`	=	`net.numLayerDelays`
`Ri`	=	`net.inputs{i}.size`
`Si`	=	`net.layers{i}.size`
`Vi`	=	`net.targets{i}.size`
`Dij`	=	`Ri * length(net.inputWeights{i,j}.delays)`

Definitions

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,

`net`	Neural network
`t`	Matrix or cell array of target vectors
`y`	Matrix or cell array of output vectors
`ew`	Error weights (default = `{1}`)

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

```
[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

```
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,

`net`	Neural network
`X`	Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)
`T`	Targets, an SxQ matrix (or MxTS cell array of SixQ matrices)
`Xi`	Initial input delay states (optional)
`Ai`	Initial layer delay states (optional)
`EW`	Error weights (optional)

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,

`s`	Sum of all absolute finite values
`n`	Number of finite values

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] = sumsqr(x)
```

Description

`[s,n] = sumsqr(x)` takes a matrix or cell array of matrices and returns,

`s`	Sum of all squared finite values
`n`	Number of finite values

If `x` contains no finite values, the sum returned is 0.

Examples

```
m = sumsqr([1 2;3 4])
[m,n] = sumsqr({[1 2; NaN 4], [4 5; 2 3]})
```

See Also

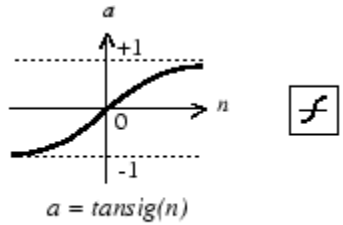
`meanabs` | `meansqr` | `sumabs`

Introduced before R2006a

tansig

Hyperbolic tangent sigmoid transfer function

Graph and Symbol



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,

`N`	S-by-Q matrix of net input (column) vectors
`FP`	Struct of function parameters (ignored)

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.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

```
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

Vogl, T.P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, “Accelerating the convergence of the backpropagation method,” *Biological Cybernetics*, Vol. 59, 1988, pp. 257–263

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,

`x`	Neural network time series data
`i`	Signal index
`ts`	Timestep index
`delays`	Row vector of increasing zero or positive delays

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.

```
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,

`inputDelays`	Row vector of increasing 0 or positive delays (default = 1:2)
`hiddenSizes`	Row vector of one or more hidden layer sizes (default = 10)
`trainFcn`	Training function (default = `'trainlm'`)

and returns a time delay neural network.

Examples

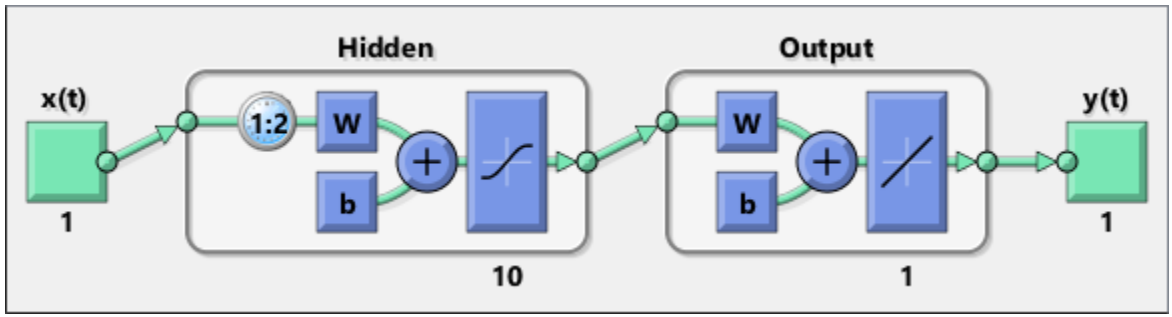
Train Time Delay Network and Predict on New Data

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.

```
net = timedelaynet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```

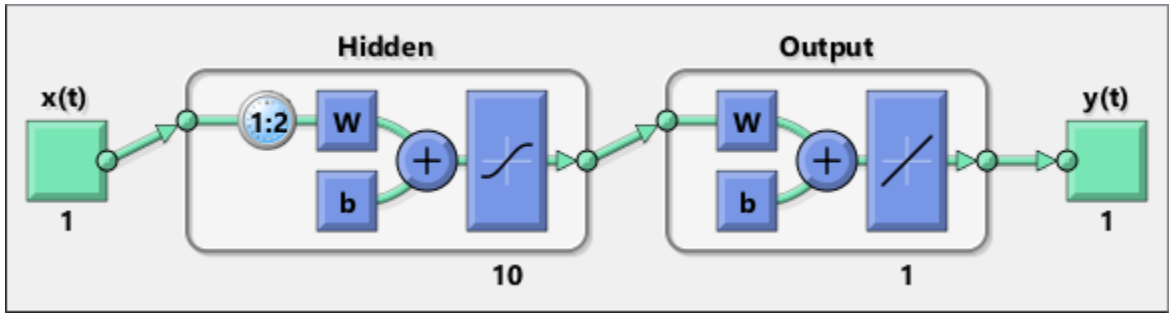


Calculate the network performance.

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y);
```

Run the prediction for 20 timesteps ahead in closed loop mode.

```
[netc,Xic,Aic] = closeloop(net,Xf,Af);
view(netc)
```



```
y2 = netc(Xnew,Xic,Aic);
```

See Also

distdelaynet | narnet | narxnet | preparets | removedelay

Introduced in R2010b

tonndata

Convert data to standard neural network cell array form

Syntax

```
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
```

Description

`[y,wasMatrix] = tonndata(x,columnSamples,cellTime)` takes these arguments,

`x`	Matrix or cell array of matrices
`columnSamples`	True if original samples are oriented as columns, false if rows
`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
`wasMatrix`	True if original data was a matrix (as apposed to cell array)

If `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 `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 `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.

```
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 neural network

For deep learning with convolutional neural networks, see `trainNetwork` instead.

Syntax

```
[net,tr] = train(net,X,T,Xi,Ai,EW)
[net,___] = train(___,'useParallel',___)
[net,___] = train(___,'useGPU',___)
[net,___] = train(___,'showResources',___)
[net,___] = train(Xcomposite,Tcomposite,___)
[net,___] = train(Xgpu,Tgpu,___)
net = train(___,'CheckpointFile','path/
name','CheckpointDelay',numDelays)
```

Description

`train` trains a network `net` according to `net.trainFcn` and `net.trainParam`.

`[net,tr] = train(net,X,T,Xi,Ai,EW)` takes

`net`	Network
`X`	Network inputs
`T`	Network targets (default = zeros)
`Xi`	Initial input delay conditions (default = zeros)
`Ai`	Initial layer delay conditions (default = zeros)
`EW`	Error weights

and returns

`net`	Newly trained network

tr	Training record (epoch and perf)

Note that T is optional and need only be used for networks that require targets. Xi is also optional and need only be used for networks that have input or layer delays.

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 is as follows:

X	R-by-Q matrix
T	U-by-Q matrix

The cell array format is more general, and more convenient for networks with multiple inputs and outputs, allowing sequences of inputs to be presented.

X	Ni-by-TS cell array	Each element X{i,ts} is an Ri-by-Q matrix.
T	No-by-TS cell array	Each element T{i,ts} is a Ui-by-Q matrix.
Xi	Ni-by-ID cell array	Each element Xi{i,k} is an Ri-by-Q matrix.
Ai	Nl-by-LD cell array	Each element Ai{i,k} is an Si-by-Q matrix.
EW	No-by-TS cell array	Each element EW{i,ts} is a Ui-by-Q matrix

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 `Xi` and `Ai` are ordered from the oldest delay condition to the most recent:

`Xi{i,k}`	=	Input `i` at time `ts = k - ID`
`Ai{i,k}`	=	Layer output `i` at time `ts = k - LD`

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`.

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:

`X`	(`sum` of `Ri`)-by-`Q` matrix
`T`	(`sum` of `Ui`)-by-`Q` matrix
`Xi`	(`sum` of `Ri`)-by-`(ID*Q)` matrix
`Ai`	(`sum` of `Si`)-by-`(LD*Q)` matrix
`EW`	(`sum` of `Ui`)-by-`Q` matrix

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.

The training record `TR` is 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`)

`[net,___] = train(___,'useParallel',___)`, `[net,___] = train(___,'useGPU',___)`, or `[net,___] = train(___,'showResources',___)` 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:

`'useParallel','no'`	Calculations occur on normal MATLAB thread. This is the default `'useParallel'` setting.
`'useParallel','yes'`	Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.
`'useGPU','no'`	Calculations occur on the CPU. This is the default 'useGPU' setting.
`'useGPU','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.
`'useGPU','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','no'`	Do not display computing resources used at the command line. This is the default setting.

`'showResources','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',N`	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.

`[net, ___ ] = train(Xcomposite,Tcomposite, ___ )` takes Composite data and returns Composite results. If Composite data is used, then `'useParallel'` is automatically set to `'yes'`.

`[net, ___ ] = train(Xgpu,Tgpu, ___ )` takes gpuArray data and returns gpuArray results. If gpuArray data is used, then `'useGPU'` is automatically set to `'yes'`.

`net = train( ___ ,'CheckpointFile','path/name','CheckpointDelay',numDelays)` periodically saves intermediate values of the neural network and training record during training to the specified file. This protects training results from power failures, computer lock ups, Ctrl+C, or any other event that halts the training process before `train` returns normally.

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).

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.

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.

Examples

Train and Plot Networks

Here input x and targets t define a simple function that you can plot:

```
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 Neural Network 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;
numWorkers = numel(Xc);
ind = [0 ceil((1:4)*(Q/4))];
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:

```
[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:

```
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.

```
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.

```
[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.

```
[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.

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

```
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:

`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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

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

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.showWindow`	`true`	Show training window
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchbac'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to determine tolerance for linear search.
net.trainParam.alpha	0.001	Scale factor that determines sufficient reduction in perf
net.trainParam.beta	0.1	Scale factor that determines sufficiently large step size
net.trainParam.delta	0.01	Initial step size in interval location step
net.trainParam.gama	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size
net.trainParam.batch_frag	0	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.

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

Train Neural Network Using `trainbfg` Train Function

This example shows how to train a neural network using the `trainbfg` train function.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainbfg');
net = train(net, x, t);
y = net(x);
```

Definitions

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k$$

where $\mathbf{A}_k^{-1}$ 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 `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, `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*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\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

Gill, Murray, & Wright, *Practical Optimization*, 1981

See Also

`cascadeforwardnet` | `feedforwardnet` | `traincgb` | `traincgf` | `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

```
[net,TR,Y,E,Pf,Af,flag_stop] = trainbfgc(net,P,T,Pi,Ai,epochs,TS,Q)
info = trainbfgc(code)
```

Description

`trainbfgc` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method. This function is called from `nnmodref`, a GUI for the model reference adaptive control Simulink block.

`[net,TR,Y,E,Pf,Af,flag_stop] = trainbfgc(net,P,T,Pi,Ai,epochs,TS,Q)` takes these inputs,

`net`	Neural network
`P`	Delayed input vectors
`T`	Layer target vectors
`Pi`	Initial input delay conditions
`Ai`	Initial layer delay conditions
`epochs`	Number of iterations for training
`TS`	Time steps
`Q`	Batch size

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch:

	`TR.epoch` Epoch number
	`TR.perf` Training performance
	`TR.vperf` Validation performance
	`TR.tperf` Test performance
`Y`	Network output for last epoch
`E`	Layer errors for last epoch
`Pf`	Final input delay conditions
`Af`	Collective layer outputs for last epoch
`flag_stop`	Indicates if the user stopped the training

Training occurs according to `trainbfgc`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchbacx'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to 0.1 (see `srch_cha`)

net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size

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

'pnames'	Names of training parameters
'pdefaults'	Default training parameters

Algorithms

`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 `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 according to the following formula:

```
dX = -H\gX;
```

where `gX` is the gradient and `H` is an 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`.
- Precision problems have occurred in the matrix inversion.

References

Gill, Murray, and Wright, *Practical Optimization*, 1981

Introduced in R2006a

trainbr

Bayesian regularization backpropagation

Syntax

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.mu`	`0.005`	Marquardt adjustment parameter
`net.trainParam.mu_dec`	`0.1`	Decrease factor for `mu`
`net.trainParam.mu_inc`	`10`	Increase factor for `mu`
`net.trainParam.mu_max`	`1e10`	Maximum value for `mu`
`net.trainParam.max_fail`	`0`	Maximum validation failures
`net.trainParam.min_grad`	`1e-7`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)

net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showWindow	true	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

Validation stops are disabled by default (max_fail = 0) 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*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) \ 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 `min_grad`.

- `mu` exceeds `mu_max`.

References

[1] MacKay, David J. C. "Bayesian interpolation." *Neural computation.* Vol. 4, No. 3, 1992, pp. 415–447.

[2] Foresee, F. Dan, and Martin T. Hagan. "Gauss-Newton approximation to Bayesian learning." *Proceedings of the International Joint Conference on Neural Networks*, June, 1997.

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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to determine tolerance for linear search.
net.trainParam.alpha	0.001	Scale factor that determines sufficient reduction in perf
net.trainParam.beta	0.1	Scale factor that determines sufficiently large step size
net.trainParam.delta	0.01	Initial step size in interval location step
net.trainParam.gama	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size

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.

Here a neural network is trained to predict body fat percentages.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgb');
net = train(net, x, t);
y = net(x);
```

Definitions

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:

$$\left|\mathbf{g}_{k-1}^{T}\mathbf{g}_{k}\right| \geq 0.2\left\|\mathbf{g}_{k}\right\|^{2}$$

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 `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Powell, M.J.D., “Restart procedures for the conjugate gradient method,” *Mathematical Programming*, Vol. 12, 1977, pp. 241–254

See Also

`trainbfg` | `traincgf` | `traincgp` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainscg`

Introduced before R2006a

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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to determine tolerance for linear search.
net.trainParam.alpha	0.001	Scale factor that determines sufficient reduction in perf
net.trainParam.beta	0.1	Scale factor that determines sufficiently large step size
net.trainParam.delta	0.01	Initial step size in interval location step
net.trainParam.gama	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size

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.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgf');
net = train(net, x, t);
y = net(x);
```

Definitions

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.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k \alpha_k \mathbf{p}_k$$

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:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of the conjugate gradient algorithm are distinguished by the manner in which the constant β_k is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

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

`traincgf` 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. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

```
Z = normnew_sqr/norm_sqr;
```

where `norm_sqr` is the norm square of the previous gradient and `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_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

See Also

`trainbfg` | `traincgb` | `traincgp` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainscg`

Introduced before R2006a

traincgp

Conjugate gradient backpropagation with Polak-Ribiére updates

Syntax

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to determine tolerance for linear search.
net.trainParam.alpha	0.001	Scale factor that determines sufficient reduction in perf
net.trainParam.beta	0.1	Scale factor that determines sufficiently large step size
net.trainParam.delta	0.01	Initial step size in interval location step
net.trainParam.gama	0.1	Parameter to avoid small reductions in performance, usually set to 0.1 (see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size

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.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'traincgp');
net = train(net, x, t);
y = net(x);
```

Definitions

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, `traincgf`, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant β_k is computed by

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

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 `traincgp` routine has performance similar to `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

`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*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. For the Polak-Ribiére variation of conjugate gradient, it is computed according to

```
Z = ((gX - gX_old)'*gX)/norm_sqr;
```

where `norm_sqr` is the norm square of the previous gradient, and `gX_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 `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Scales, L.E., *Introduction to Non-Linear Optimization*, New York, Springer-Verlag, 1985

See Also

`trainbfg` | `traincgb` | `traincgf` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainrp` | `trainscg`

Introduced before R2006a

traingd

Gradient descent backpropagation

Syntax

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.min_grad`	`1e-5`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.time`	`inf`	Maximum time to train in seconds

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.

Definitions

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;
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.lr_inc`	`1.05`	Ratio to increase learning rate
`net.trainParam.lr_dec`	`0.7`	Ratio to decrease learning rate
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.max_perf_inc`	`1.04`	Maximum performance increase
`net.trainParam.min_grad`	`1e-5`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI

net.trainParam.time	inf	Maximum time to train in seconds

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.

Definitions

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_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_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:

```
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*dperf/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

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.mc`	`0.9`	Momentum constant
`net.trainParam.min_grad`	`1e-5`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between showing progress
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `traingdm` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `traingdm`,

1. Set `net.trainFcn` to `'traingdm'`. This sets `net.trainParam` to `traingdm`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingdm`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

Definitions

Gradient Descent with Momentum

In addition to `traingd`, there are three other variations of gradient descent.

Gradient descent with momentum, implemented by `traingdm`, 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.)

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = feedforwardnet(3,'traingdm');
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*dXprev + lr*(1-mc)*dperf/dX
```

where `dXprev` 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

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.lr_inc`	`1.05`	Ratio to increase learning rate
`net.trainParam.lr_dec`	`0.7`	Ratio to decrease learning rate
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.max_perf_inc`	`1.04`	Maximum performance increase
`net.trainParam.mc`	`0.9`	Momentum constant
`net.trainParam.min_grad`	`1e-5`	Minimum performance gradient
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output

net.trainParam.showWindow	true	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

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.

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

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.min_grad`	`1e-7`	Minimum performance gradient
`net.trainParam.mu`	`0.001`	Initial `mu`
`net.trainParam.mu_dec`	`0.1`	`mu` decrease factor
`net.trainParam.mu_inc`	`10`	`mu` increase factor
`net.trainParam.mu_max`	`1e10`	Maximum `mu`

net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showWindow	true	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

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.

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.

```
[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.

Definitions

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

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}$$

where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and $\mathbf{e}$ 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:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton's method, using the approximate Hessian matrix. When μ 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, μ 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 `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) \ je
```

where `E` is all errors and `I` is the identity matrix.

The adaptive value `mu` is increased by `mu_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_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 `min_grad`.
- `mu` exceeds `mu_max`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

Introduced before R2006a

trainoss

One-step secant backpropagation

Syntax

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.min_grad`	`1e-10`	Minimum performance gradient
`net.trainParam.searchFcn`	`'srchbac'`	Name of line search routine to use
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length
`net.trainParam.bmax`	`26`	Maximum step size

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.

```
[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainoss');
net = train(net, x, t);
y = net(x);
```

Definitions

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*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_step + Bc*dgX;
```

where `gX` is the gradient, `X_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 (*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 `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

References

Battiti, R., “First and second order methods for learning: Between steepest descent and Newton’s method,” *Neural Computation*, Vol. 4, No. 2, 1992, pp. 141–166

See Also

`trainbfg` | `traincgb` | `traincgf` | `traincgp` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainrp` | `trainscg`

Introduced before R2006a

trainr

Random order incremental training with learning functions

Syntax

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

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

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-5`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.delt_inc`	`1.2`	Increment to weight change
`net.trainParam.delt_dec`	`0.5`	Decrement to weight change

net.trainParam.delta0	0.07	Initial weight change
net.trainParam.deltamax	50.0	Maximum weight change

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.

```
net = feedforwardnet(2,'trainrp');
```

Here the network is trained and retested.

```
net.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.

Definitions

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:

```
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:

```
dX = deltaX.*sign(gX);
```

where the elements of `deltaX` are all initialized to `delta0`, 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 `delta_dec`. If an element of `gX` maintains the same sign from one iteration to the next, then the corresponding element of `deltaX` is increased by `delta_inc`. 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

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.

See Also

`trainbfg` | `traincgb` | `traincgf` | `traincgp` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainscg`

Introduced before R2006a

trainru

Unsupervised random order weight/bias training

Syntax

```
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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`Inf`	Maximum time to train in seconds

Network Use

To prepare a custom network to be trained with `trainru`,

1. Set `net.trainFcn` to `'trainru'`. This sets `net.trainParam` to `trainru`'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`.

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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.time`	`Inf`	Maximum time to train in seconds

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:

`net.trainParam.epochs`	`1000`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (NaN for no displays)
`net.trainParam.showCommandLine`	`false`	Generate command-line output
`net.trainParam.showWindow`	`true`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`6`	Maximum validation failures
`net.trainParam.sigma`	`5.0e-5`	Determine change in weight for second derivative approximation

`net.trainParam.lambda`	`5.0e-7`	Parameter for regulating the indefiniteness of the Hessian

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.

```
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.

```
net = feedforwardnet(2,'trainscg');
```

Here the network is trained and retested.

```
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

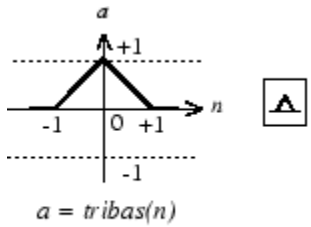
`trainbfg` | `traincgb` | `traincgf` | `traincgp` | `traingda` | `traingdm` | `traingdx` | `trainlm` | `trainoss` | `trainrp`

Introduced before R2006a

tribas

Triangular basis transfer function

Graph and Symbol



Triangular Basis Function

Syntax

```
A = tribas(N,FP)
```

Description

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

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

`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 triangular basis function applied to each element of `N`.

`info = tribas('code')` can take the following forms to return specific information:

`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 `dA_dN` is `S`-by-`S`-by-`Q` or `S`-by-`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.

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

Assign this transfer function to layer `i` of a network.

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

Algorithms

```
a = tribas(n) = 1 - abs(n), if -1 <= n <= 1
              = 0, otherwise
```

See Also

`radbas` | `sim`

Introduced before R2006a

tritop

Triangle layer topology function

Syntax

```
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:

`dimensions`	Row vector of dimension sizes

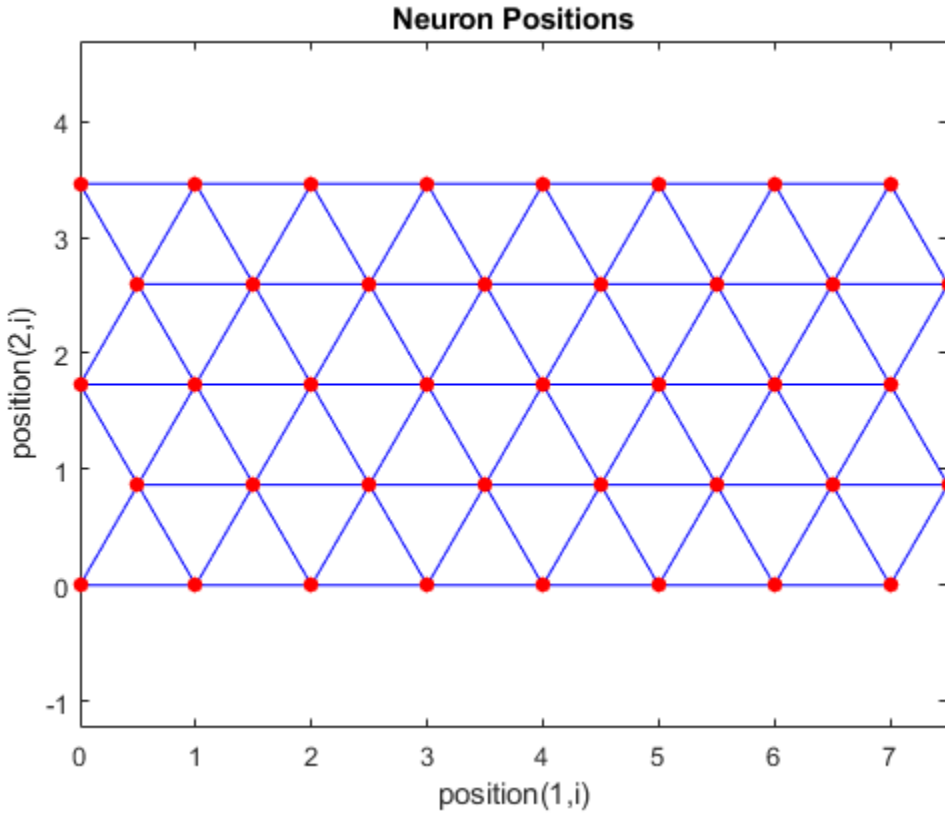
and returns an `N`-by-`S` matrix of `N` coordinate vectors, where `N` is the number of dimensions and `S` is the product of `dimensions`.

Examples

Display Layer with Triangular Pattern

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

```
pos = tritop([8 5]);
plotsom(pos)
```



See Also

`gridtop` | `hextop` | `randtop`

Introduced in R2010b

unconfigure

Unconfigure network inputs and outputs

Syntax

```
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.

```
[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,

`vec`	Matrix of vectors, each containing a single 1

and returns

`ind`	The indices of the 1s
`n`	The number of rows in `vec`

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

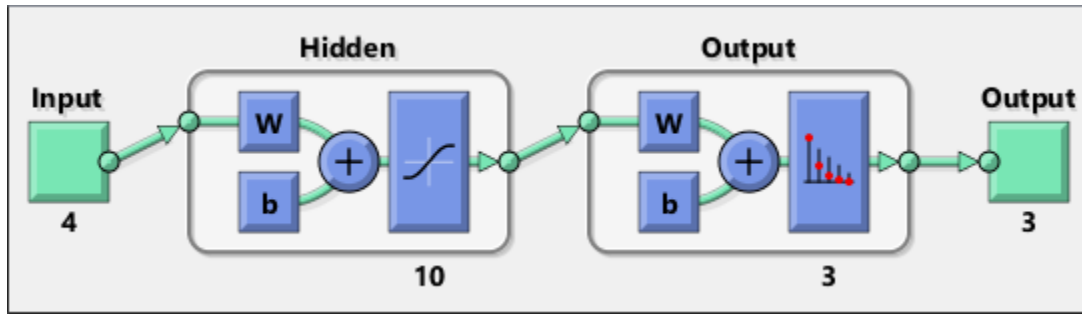
`view(net)` opens a window that shows your neural network (specified in `net`) as a graphical diagram.

Example

View Neural Network

This example shows how to view the diagram of a pattern recognition network.

```
[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
- Neural Network 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
- Neural Network 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
- Neural Network 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
- Neural Network 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

`MiniBatchable` is an abstract mixin class that adds support for mini-batches to your custom datastore for use with Neural Network Toolbox. A mini-batch datastore contains training and test data sets for use in network training, prediction, and classification.

To use this mixin class, you must inherit from the `MiniBatchable` class in addition to inheriting from the `Datastore` base class. Type the following syntax as the first line of your class definition file.

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.MiniBatchable
    ...
end
```

To add support for mini-batches to your datastore:

- Inherit from an additional class `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 mini-batch size defined in `trainingOptions`.

Attributes:

`Abstract`	`true`
`Access`	`Public`

`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:

`Abstract`	`true`
`SetAccess`	`Protected`
`ReadAccess`	`Public`

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).

Alternative Functionality

You can use built-in mini-batch datastores to perform specific image preprocessing operations on each batch of data. For more information, see “Advanced Image Preprocessing”.

See Also

`BackgroundDispatchable` | `Datastore` | `Partitionable` | `PartitionableByIndex` | `Shuffleable` | `matlab.io.datastore.MiniBatchable.read`

Topics

“Develop Custom Mini-Batch Datastore”
“Preprocess Images for Deep Learning”
“Deep Learning in MATLAB”

Introduced in R2018a

read

Class: matlab.io.datastore.MiniBatchable
Package: matlab.io.datastore

Read data from mini-batch datastore

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

custom `MiniBatchable` datastore | `augmentedImageDatastore` | ...

Mini-batch datastore, specified as a built-in or custom mini-batch datastore. For more information, see “Advanced Image Preprocessing”.

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).

See Also

`Datastore` | `MiniBatchable` | `matlab.io.Datastore.read`

Topics

“Develop Custom Mini-Batch Datastore”

Introduced in R2018a

matlab.io.datastore.BackgroundDispatchable class

Package: matlab.io.datastore

Add prefetch reading support to datastore

Description

`BackgroundDispatchable` is an abstract mixin class that adds support for prefetch reading to your custom datastore for use with Neural Network Toolbox.

To use this mixin class, you must inherit from the `BackgroundDispatchable` class in addition to inheriting from the `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 `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”.

Properties

`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.

Attributes:

`Public`	`true`

Methods

readByIndex Return observations from a datastore specified by index

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).

See Also

`Datastore` | `MiniBatchable` | `Partitionable` | `PartitionableByIndex` | `Shuffleable`

Introduced in R2018a

readByIndex

Class: matlab.io.datastore.BackgroundDispatchable
Package: matlab.io.datastore

Return observations from a datastore specified by index

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 mini-batch datastore, then `data` is a

table with the same format as returned by the `matlab.io.datastore.MiniBatchable.read` 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.

Field Name	Description
`Filename`	`Filename` is a fully resolved path containing the path string, name of the file, and file extension.
`FileSize`	Total file size, in bytes. For MAT-files, `FileSize` is the total number of key-value pairs in the file.

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 `BackgroundDispatchable` class. For more information, see “Develop Custom Mini-Batch Datastore”.

See Also

`BackgroundDispatchable` | `Datastore` | `matlab.io.Datastore.read` | `matlab.io.datastore.MiniBatchable.read` | `readall`

Introduced in R2018a

matlab.io.datastore.Shuffleable class

Package: matlab.io.datastore

Add shuffling support to datastore

Description

`Shuffleable` is an abstract mixin class that adds support for shuffling samples in a datastore in random order.

To use this mixin class, you must inherit from the `Shuffleable` class in addition to inheriting from the `Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.Shuffleable
    ...
end
```

To add support for shuffling to your custom datastore, you must:

- Inherit from the additional class `Shuffleable`
- Define this additional method: `shuffle`.

For more details and steps to create your custom datastore, see “Develop Custom Mini-Batch Datastore”.

Methods

shuffle Return a shuffled version of a datastore

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).

See Also

`BackgroundDispatchable` | `Datastore` | `HadoopFileBased` | `MiniBatchable` | `Partitionable` | `PartitionableByIndex`

Introduced in R2018a

shuffle

Class: matlab.io.datastore.Shuffleable
Package: matlab.io.datastore

Return a shuffled version of a datastore

Syntax

```
ds2 = shuffle(ds)
```

Description

`ds2 = shuffle(ds)` shuffles samples of the datastore `ds` in random order, and returns the shuffled datastore, `ds2`.

Input Arguments

ds — Input datastore

`Datastore` object

Input datastore, specified as a `Datastore` object.

Output Arguments

ds2 — Shuffled datastore

`Datastore` object

Shuffled datastore, returned as a `Datastore` object.

Attributes

`Abstract`	`true`
`Access`	`Public`

To learn about attributes of methods, see Method Attributes (MATLAB).

Tips

- You must implement the `shuffle` method by deriving a subclass from the `Shuffleable` class. For more information, see “Develop Custom Mini-Batch Datastore”.

See Also

`Datastore` | `Shuffleable`

Introduced in R2018a

matlab.io.datastore.PartitionableByIndex class

Package: matlab.io.datastore

Add parallelization support to datastore

Description

`PartitionableByIndex` is an abstract mixin class that adds parallelization support to your custom datastore for use with Neural Network Toolbox. This class requires Parallel Computing Toolbox.

To use this mixin class, you must inherit from the `PartitionableByIndex` class in addition to inheriting from the `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 `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 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).

See Also

`BackgroundDispatchable` | `Datastore` | `HadoopFileBased` | `MiniBatchable` | `Partitionable` | `Shuffleable`

Introduced in R2018a

partitionByIndex

Partition a datastore according to indices

Syntax

```
ds2 = partitionByIndex(ds,ind)
```

Description

`ds2 = partitionByIndex(ds,ind)` partitions a subset of observations in a datastore, `ds`, into a new datastore, `ds2`. 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

`ds2` — Partitioned datastore
`Datastore` object

Partitioned datastore, returned as a `Datastore` object.

Attributes

`Abstract`	`true`
`Access`	`Public`

To learn about attributes of methods, see Method Attributes (MATLAB).

Tips

- You must implement the `partitionByIndex` method by deriving a subclass from the `PartitionableByIndex` class. For more information, see “Develop Custom Mini-Batch Datastore”.

See Also

`Datastore` | `PartitionableByIndex`

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.

```
autoenc = trainAutoencoder(X);
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed = predict(autoenc,X);
```

Compute the mean squared reconstruction error.

```
mseError = mse(X-XReconstructed)
```

```
mseError = 0.0167
```

Train Autoencoder with Specified Options

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 hidden size 4, 400 maximum epochs, and linear transfer function for the decoder.

```
autoenc = trainAutoencoder(X,4,'MaxEpochs',400,...
'DecoderTransferFunction','purelin');
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed = predict(autoenc,X);
```

Compute the mean squared reconstruction error.

```
mseError = mse(X-XReconstructed)
```

```
mseError = 0.0046
```

Reconstruct Observations Using Sparse Autoencoder

Generate the training data.

```
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.

```
hiddenSize = 25;
autoenc = trainAutoencoder(x',hiddenSize,...
        'EncoderTransferFunction','satlin',...
        'DecoderTransferFunction','purelin',...
        'L2WeightRegularization',0.01,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.10);
```

Generate the test data.

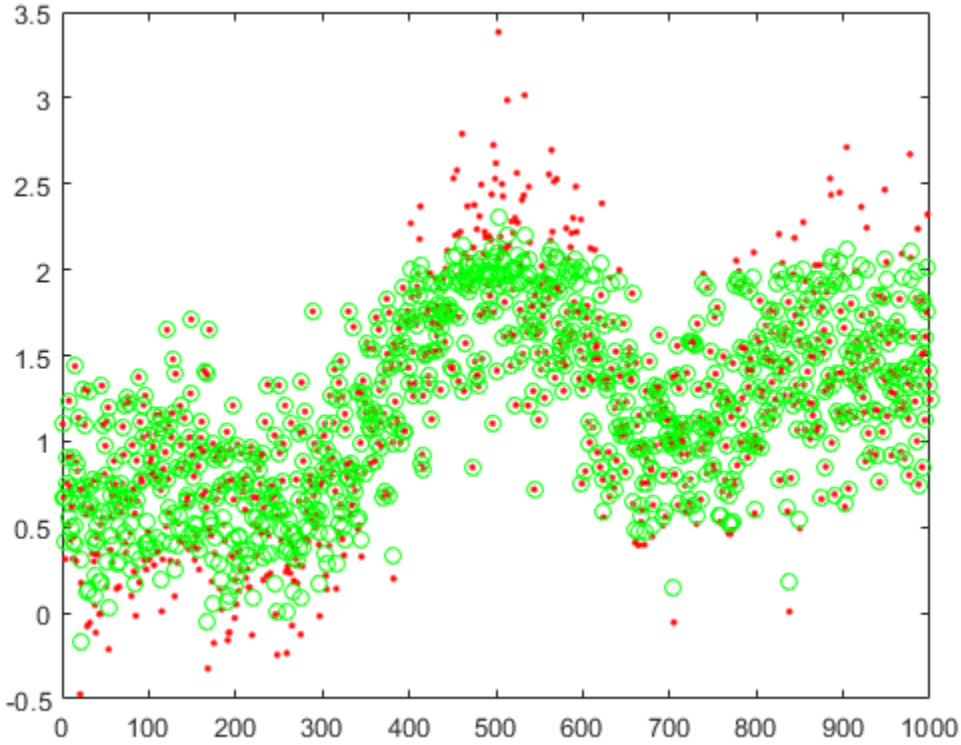
```
n = 1000;
r = sort(-10 + 20*rand(n,1));
xtest = 1 + r*5e-2 + sin(r)./r + 0.4*randn(n,1);
```

Predict the test data using the trained autoencoder, `autoenc` .

```
xReconstructed = predict(autoenc,xtest');
```

Plot the actual test data and the predictions.

```
figure;
plot(xtest,'r.');
hold on
plot(xReconstructed,'go');
```



Reconstruct Handwritten Digit Images Using Sparse Autoencoder

Load the training data.

```
XTrain = 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;
```

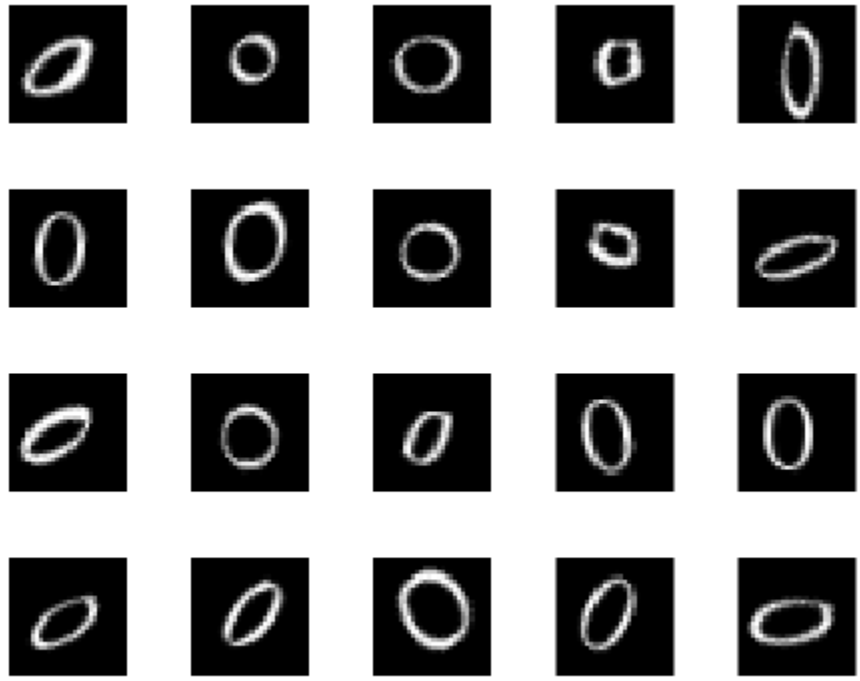
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, `autoenc`.

```
xReconstructed = predict(autoenc,XTest);
```

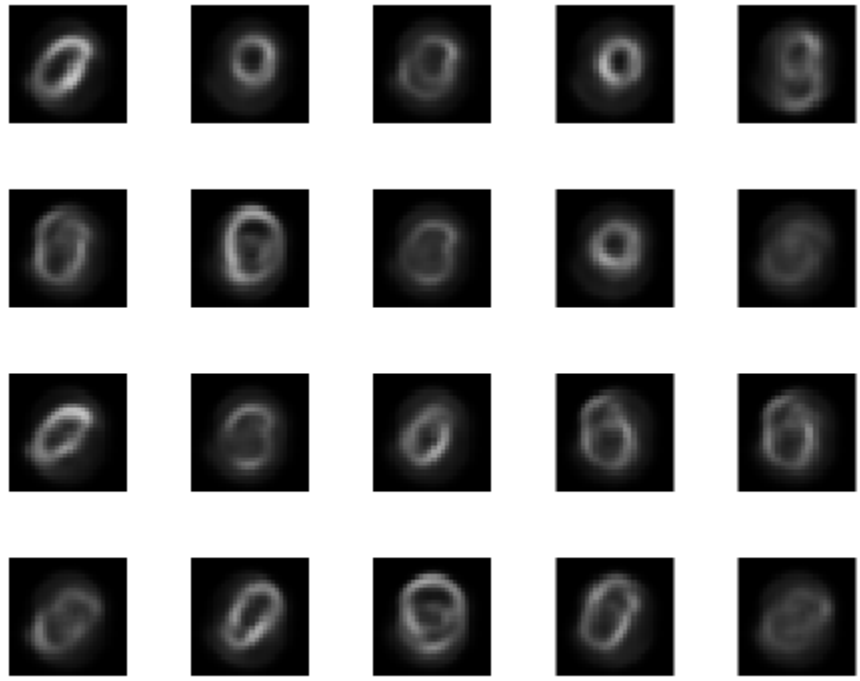
View the actual test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(XTest{i});
end
```



View the reconstructed test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(xReconstructed{i});
end
```



- “Construct Deep Network Using Autoencoders”

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 single 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.

Transfer Function Option	Definition
`'logsig'`	Logistic sigmoid function $$f(z)=\frac{1}{1+e^{-z}}$$

Transfer Function Option	Definition
`'satlin'`	Positive saturating linear transfer function $$f(z) = \begin{cases} 0, & \text{if } z \le 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \ge 1 \end{cases}$$

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.

Transfer Function Option	Definition
`'logsig'`	Logistic sigmoid function $$f(z) = \frac{1}{1+e^{-z}}$$
`'satlin'`	Positive saturating linear transfer function $$f(z) = \begin{cases} 0, & \text{if } z \le 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \ge 1 \end{cases}$$
`'purelin'`	Linear transfer function $$f(z) = z$$

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 1-602 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 1-601 as follows:

$$E = \underbrace{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(x_{kn}-\hat{x}_{kn}\right)^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{weights}}_{\substack{L_2\\ \text{regularization}}} + \beta * \underbrace{\Omega_{sparsity}}_{\substack{\text{sparsity}\\ \text{regularization}}},$$

where λ is the coefficient for the L_2 regularization term on page 1-602 and β is the coefficient for the sparsity regularization term on page 1-601. You can specify the values of λ and β 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 1-601.

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 1-601 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.

Definitions

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 $\boldsymbol{x} \in \mathbb{R}^{D_x}$, then the encoder maps the vector x to another vector $\boldsymbol{z} \in \mathbb{R}^{D^{(1)}}$ as follows:

$$\boldsymbol{z} = h^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

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:

$$\hat{\boldsymbol{x}} = h^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{z} + \boldsymbol{b}^{(2)}\right),$$

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.

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:

$$\hat{\rho}_i = \frac{1}{n}\sum_{j=1}^{n} z_i^{(1)}\left(x_j\right) = \frac{1}{n}\sum_{j=1}^{n} h\left(w_i^{(1)T} x_j + b_i^{(1)}\right),$$

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 $\boldsymbol{W}^{(1)}$, and $b_i^{(1)}$ is the *i*th entry of the bias vector, $\boldsymbol{b}^{(1)}$. 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, ρ, are not close in value [2]. One such sparsity regularization term can be the Kullback-Leibler divergence.

$$\Omega_{sparsity} = \sum_{i=1}^{D^{(1)}} KL(\rho \parallel \hat{\rho}_i) = \sum_{i=1}^{D^{(1)}} \rho \log\left(\frac{\rho}{\hat{\rho}_i}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_i}\right)$$

Kullback-Leibler divergence is a function for measuring how different two distributions are. In this case, it takes the value zero when ρ 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 ρ 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 $w^{(l)}$ 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:

$$\Omega_{weights} = \frac{1}{2}\sum_{l}^{L}\sum_{j}^{n}\sum_{i}^{k}\left(w_{ji}^{(l)}\right)^2,$$

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 1-601 is an adjusted mean squared error function as follows:

$$E = \underbrace{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}(x_{kn} - \hat{x}_{kn})^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{weights}}_{\substack{L_2 \\ \text{regularization}}} + \beta * \underbrace{\Omega_{sparsity}}_{\substack{\text{sparsity} \\ \text{regularization}}},$$

where λ is the coefficient for the L_2 regularization term on page 1-602 and β is the coefficient for the sparsity regularization term on page 1-601. You can specify the values

of λ and β by using the `L2WeightRegularization` and `SparsityRegularization` name-value pair arguments, respectively, while training an autoencoder.

References

[1] Moller, M. F. “A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning”, *Neural Networks*, Vol. 6, 1993, pp. 525–533.

[2] Olshausen, B. A. and D. J. Field. “Sparse Coding with an Overcomplete Basis Set: A Strategy Employed by V1.” *Vision Research*, Vol.37, 1997, pp.3311–3325.

See Also

`Autoencoder` | `encode` | `stack` | `trainSoftmaxLayer`

Topics

“Construct Deep Network Using Autoencoders”

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.

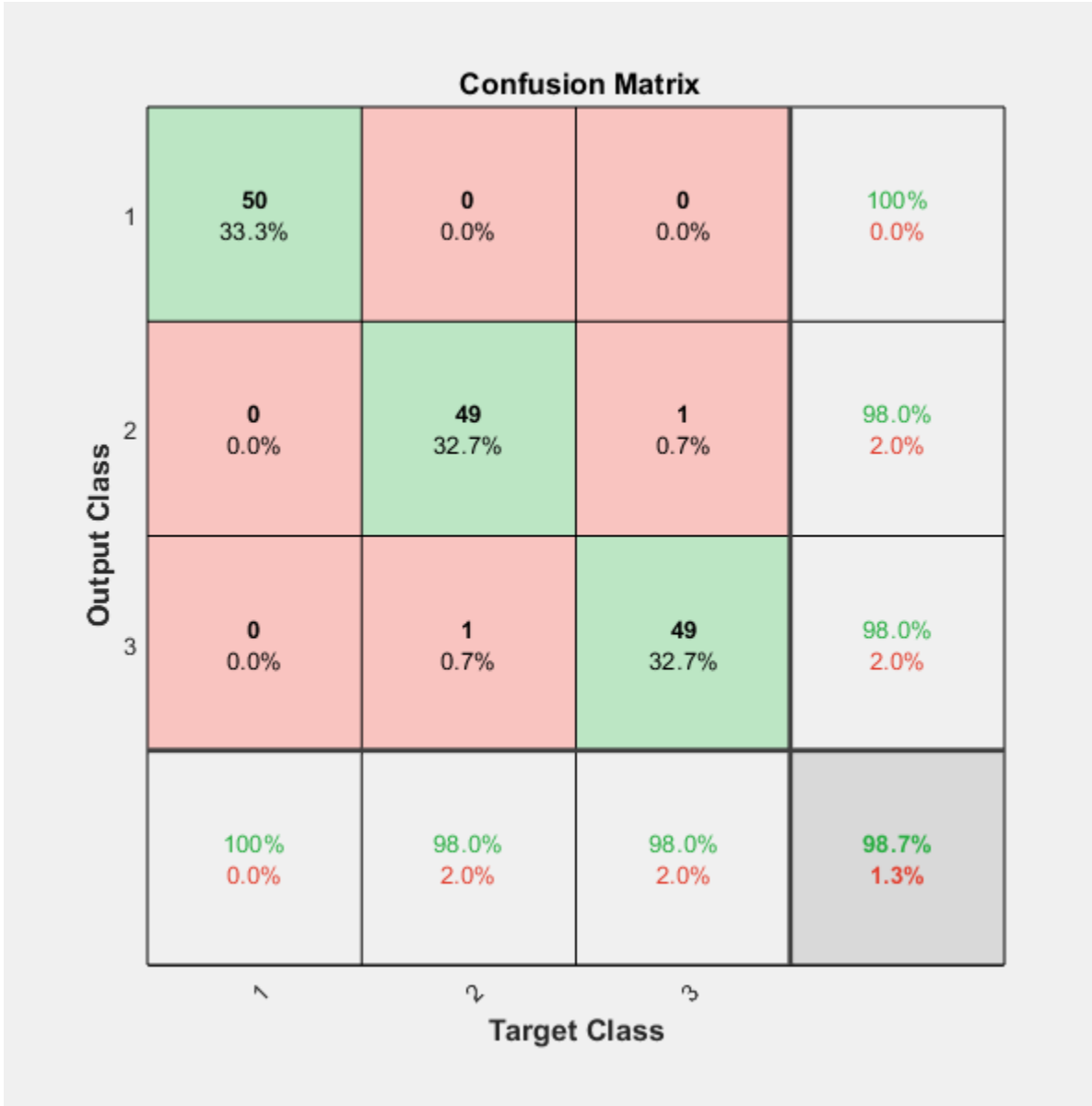
```
net = trainSoftmaxLayer(X,T);
```

Classify the observations into one of the three classes using the trained softmax layer.

```
Y = net(X);
```

Plot the confusion matrix using the targets and the classifications obtained from the softmax layer.

```
plotconfusion(T,Y);
```


Confusion Matrix
Output Class
Target Class
1
2
3
50
33.3%
0
0.0%
0
0.0%
100%
0.0%
0
0.0%
49
32.7%
1
0.7%
98.0%
2.0%
0
0.0%
1
0.7%
49
32.7%
98.0%
2.0%
100%
0.0%
98.0%
2.0%
98.0%
2.0%
98.7%
1.3%

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 single 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:

$$E = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k}\left(t_{ij} - y_{ij}\right)^2,$$

where n is the number of training examples, and k is the number of classes. t_{ij} is the ijth entry of the target matrix, `T`, and y_{ij} is the ith output from the autoencoder when the input vector is $\boldsymbol{x_j}$.

The cross entropy function is given by:

$$E = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k} t_{ij}\ln y_{ij} + \left(1 - t_{ij}\right)\ln\left(1 - y_{ij}\right).$$

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 1-616 `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.

```
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.

```
hiddenSize = 15;
autoenc = trainAutoencoder(X,hiddenSize);
```

Extract the encoded data for new images using the autoencoder.

```
Xnew = digitTestCellArrayData;
features = encode(autoenc,Xnew);
```

Decode the encoded data from the autoencoder.

```
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 $\boldsymbol{x} \in \mathbb{R}^{D_x}$, then the encoder maps the vector x to another vector $\boldsymbol{z} \in \mathbb{R}^{D^{(1)}}$ as follows:

$$\boldsymbol{z} = h^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

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:

$$\hat{\boldsymbol{x}} = h^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{z} + \boldsymbol{b}^{(2)}\right),$$

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 = encode(autoenc,Xnew)
```

Description

`Z = encode(autoenc,Xnew)` returns the encoded data on page 1-618, `Z`, for the input data `Xnew`, using the autoencoder, `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 `autoenc` was trained on a matrix, where each column represents a single sample, then `Xnew` must be a matrix, where each column represents a single sample.

If the autoencoder `autoenc` was trained on a cell array of images, then `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 $\boldsymbol{x} \in \mathbb{R}^{D_x}$, then the encoder maps the vector x to another vector $\boldsymbol{z} \in \mathbb{R}^{D^{(1)}}$ as follows:

$$\boldsymbol{z} = h^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

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.

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 single 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.

```
X = iris_dataset;
```

Train an autoencoder with 4 neurons in the hidden layer.

```
autoenc = trainAutoencoder(X,4);
```

Generate the code for running the autoencoder. Show the links to the MATLAB function.

```
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.

```
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;
```

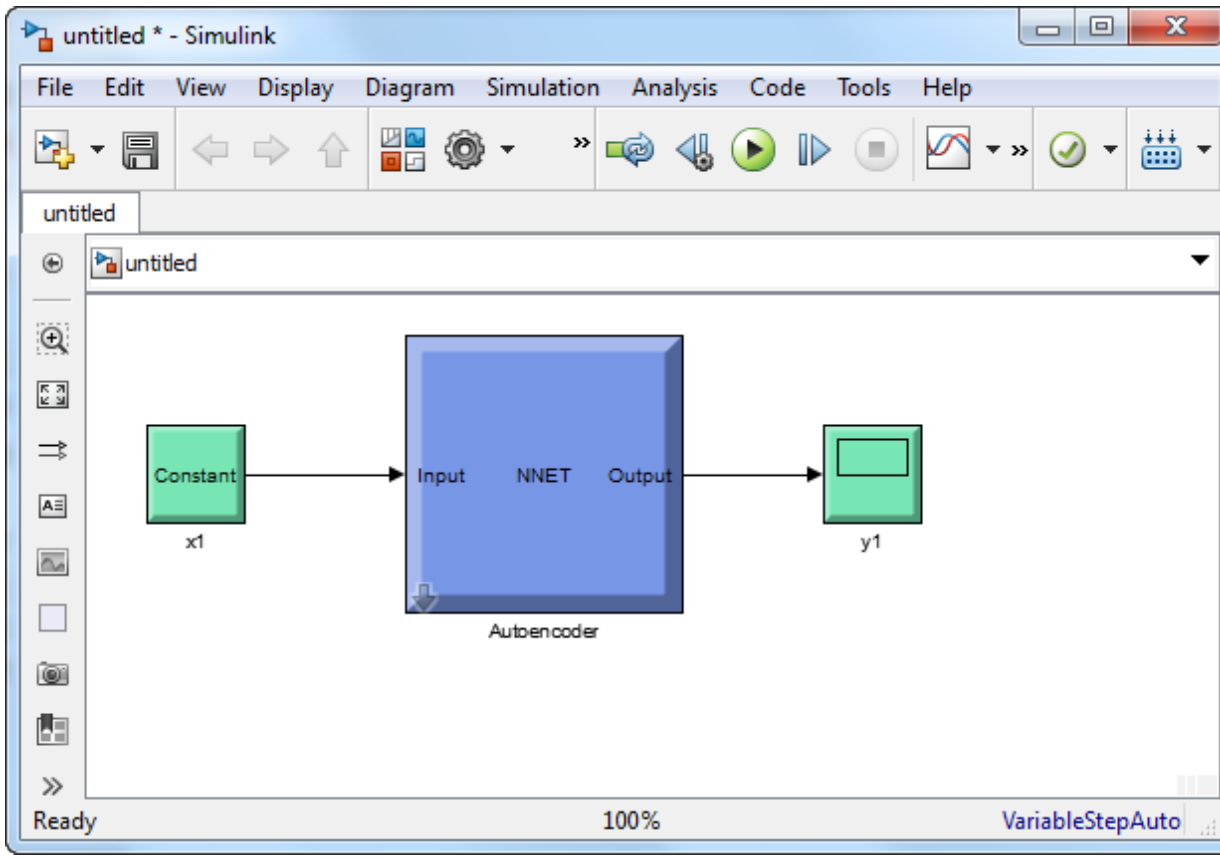
The training data is a 1-by-500 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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)
```



See Also

`trainAutoencoder`

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 = bodyfat_dataset;
```


```
X = bodyfat_dataset;
```


X is a 13-by-252 matrix defining thirteen attributes of 252 different neighborhoods. For more information on the data, type `help house_dataset` in the command line.

Train an autoencoder on the attribute data.

```
autoenc = trainAutoencoder(X);
```

Create a network object from the autoencoder, `autoenc` .

```
net = network(autoenc);
```

Predict the attributes using the network, `net` .

```
Xpred = 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] = regression(X, Xpred, 'one')
```

```
C = 0.9996
```

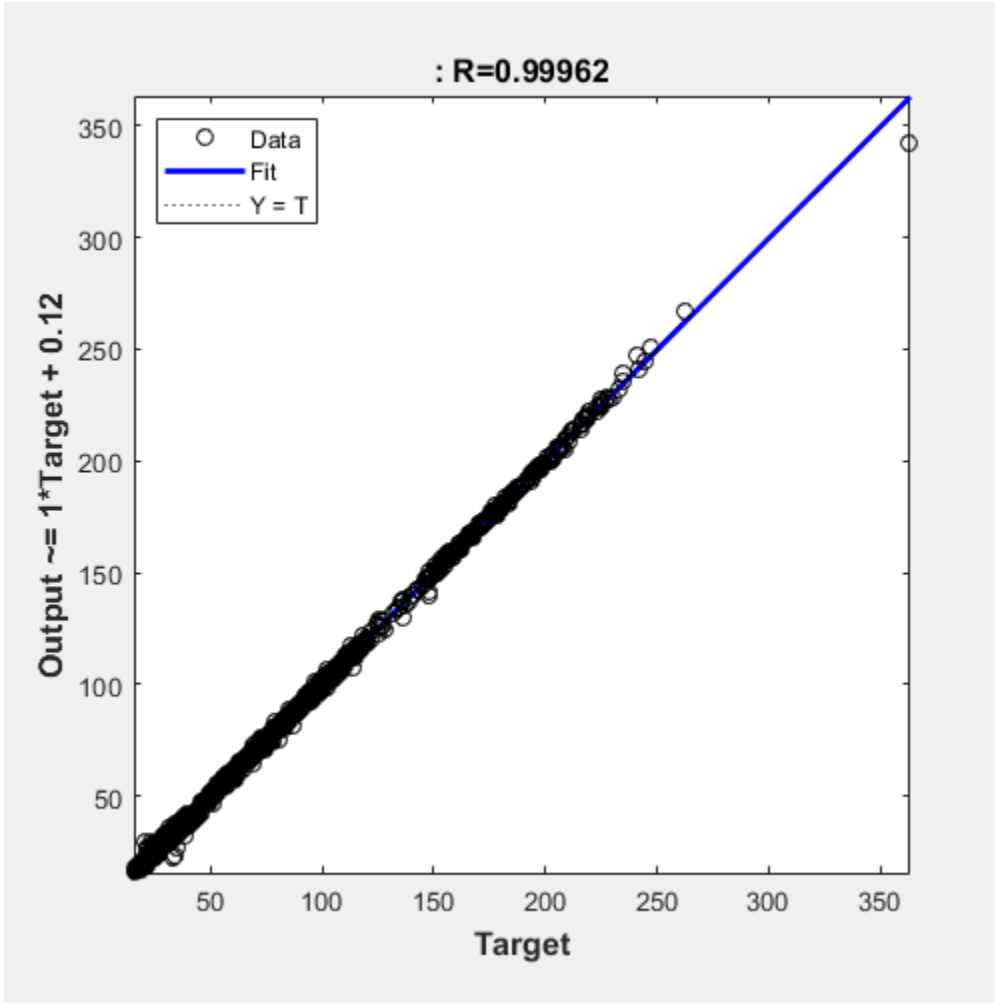
```
S = 0.9982
```

```
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.

```
plotregression(X, Xpred);
```



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 = digitTrainCellArrayData;
```

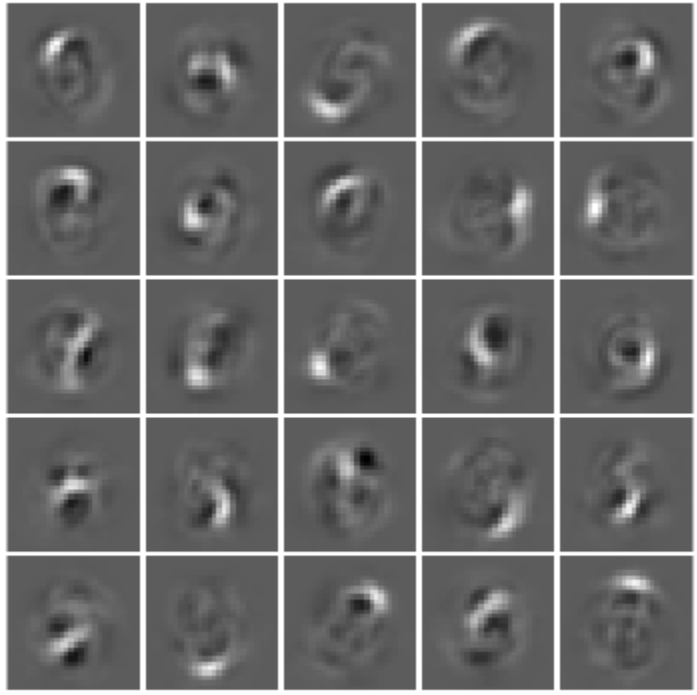
The training data is a 1-by-5000 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden layer of 25 neurons.

```
hiddenSize = 25;
autoenc = trainAutoencoder(X,hiddenSize, ...
  'L2WeightRegularization',0.004, ...
  'SparsityRegularization',4, ...
  'SparsityProportion',0.2);
```

Visualize the learned features.

```
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 = predict(autoenc,X)
```

Description

`Y = predict(autoenc,X)` returns the predictions `Y` for the input data `X`, using the autoencoder `autoenc`. The result `Y` is a reconstruction of `X`.

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 `autoenc` was trained on a matrix, where each column represents a single sample, then `Xnew` must be a matrix, where each column represents a single sample.

If the autoencoder `autoenc` was trained on a cell array of images, then `Xnew` must either be a cell array of image data or an array of single image data.

Data Types: `single` | `double` | `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');
```

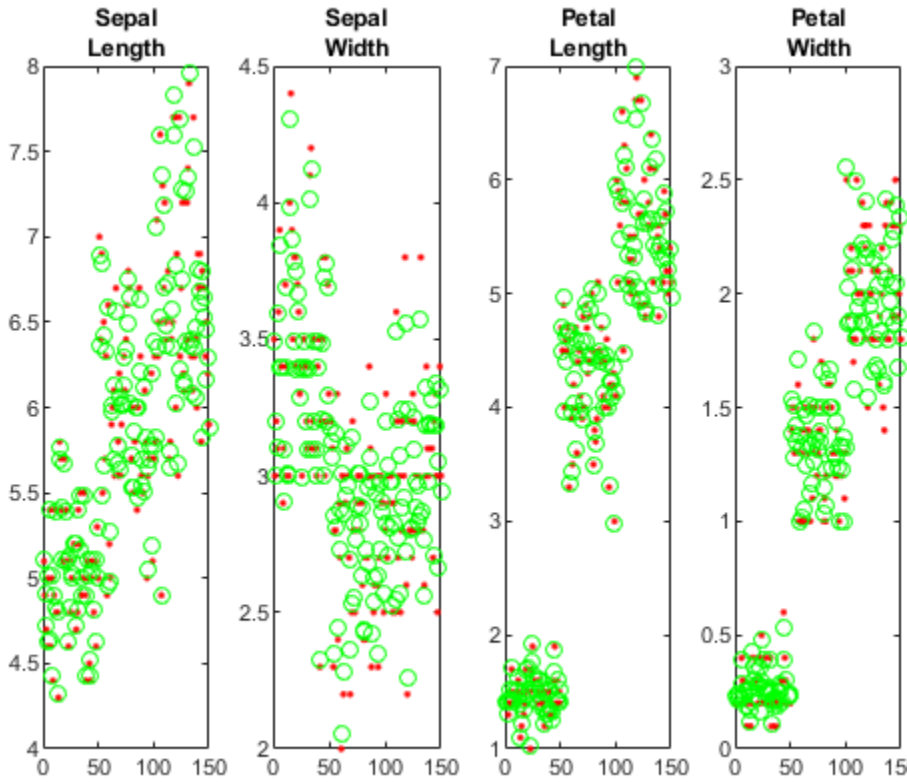
Reconstruct the measurements using the trained network, `autoenc`.

```
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);
```

```
plot(X(i,:),'r.');
hold on
plot(xReconstructed(i,:),'go');
hold off;
end
title(h(1),{'Sepal';'Length'});
title(h(2),{'Sepal';'Width'});
title(h(3),{'Petal';'Length'});
title(h(4),{'Petal';'Width'});
```



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.

```
XTrain = 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;
```

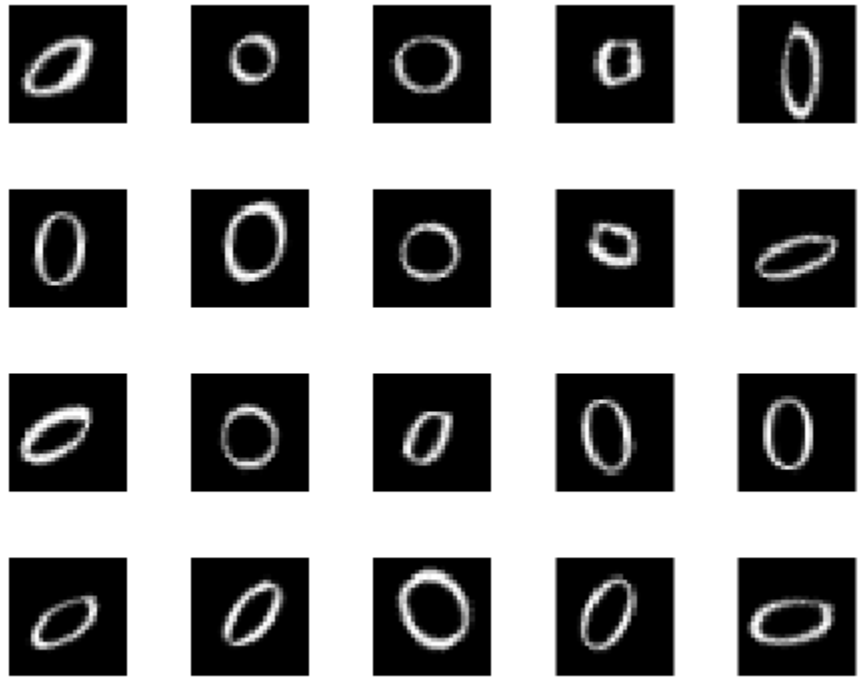
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, `autoenc`.

```
xReconstructed = predict(autoenc,XTest);
```

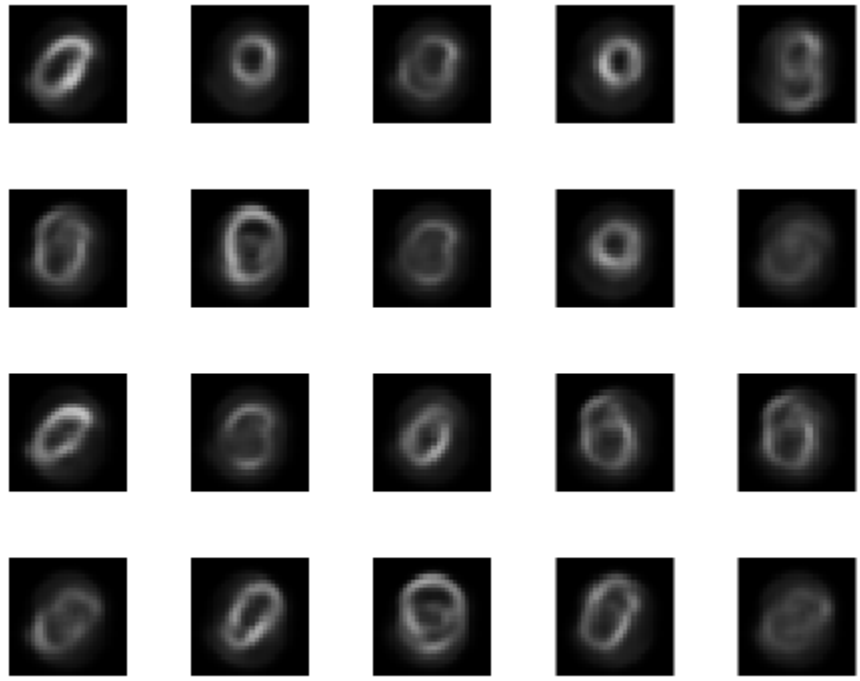
View the actual test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(XTest{i});
end
```



View the reconstructed test data.

```
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, ...
    'DecoderTransferFunction','purelin');
```

Extract the features in the hidden layer.

```
features = encode(autoenc,X);
```

Train a softmax layer for classification using the `features` .

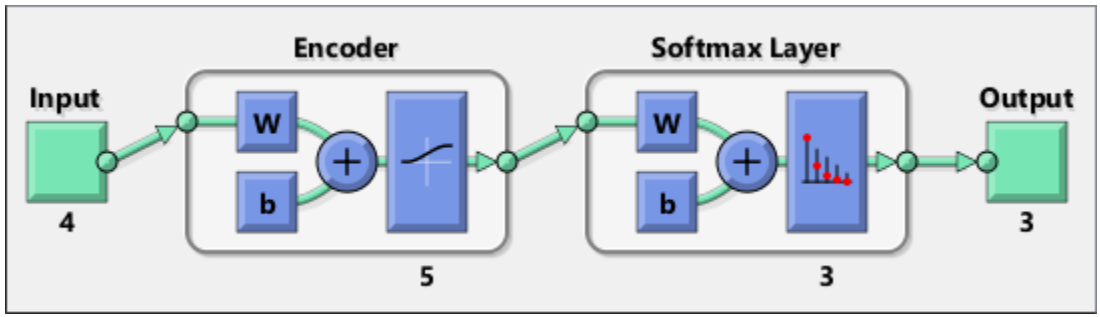
```
softnet = trainSoftmaxLayer(features,T);
```

Stack the encoder and the softmax layer to form a deep network.

```
stackednet = stack(autoenc,softnet);
```

View the stacked network.

```
view(stackednet);
```



- “Construct Deep Network Using Autoencoders”

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

“Construct Deep Network Using Autoencoders”

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;
```

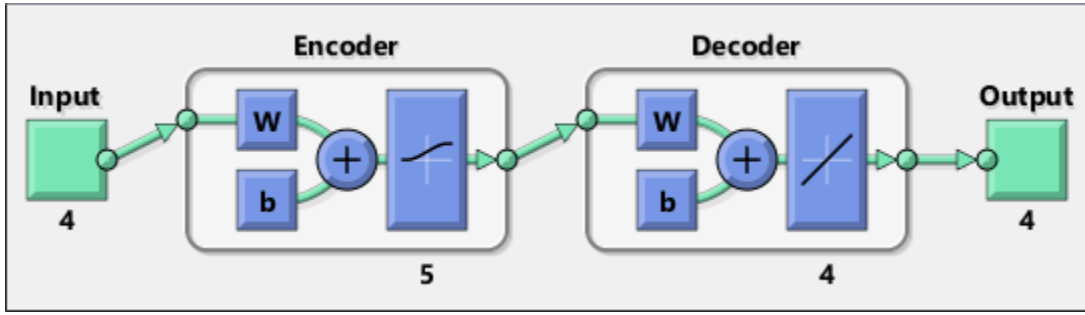
Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
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

Create classification output layer

Syntax

```
coutputlayer = classificationLayer
coutputlayer = classificationLayer('Name',Name)
```

Description

`coutputlayer = classificationLayer` returns a classification output layer for a neural network. The classification output layer holds the name of the loss function on page 1-647 that the software uses for training the network for multi-class classification, the size of the output, and the class labels.

To specify the number of classes K of the network, specify the previous layer to have output size K. For example, include a fully connected layer with output size K and a softmax layer before the classification output layer.

`coutputlayer = classificationLayer('Name',Name)` returns a classification layer with name specified by `name`.

Examples

Create Classification Output Layer

Create a classification output layer with the name `'coutput'`.

```
layer = classificationLayer('Name','coutput')
```

```
layer =
  ClassificationOutputLayer with properties:

            Name: 'coutput'
```

```
      ClassNames: {1x0 cell}
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

The default loss function for classification is cross entropy for k mutually exclusive classes.

Include a classification output 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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Input Arguments

Name — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Output Arguments

`coutputlayer` — Classification output layer
`ClassificationOutputLayer` object

Classification output layer, returned as a `ClassificationOutputLayer` object.

For information on concatenating layers to construct convolutional neural network architecture, see `Layer`.

Definitions

Cross Entropy Function for *k* Mutually Exclusive Classes

For multiclass classification problems, the software assigns each input to one of the *k* mutually exclusive classes. The loss (error) function for this case is the cross entropy function for a 1-of-*k* coding scheme:

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\sum_{j=1}^{k} t_{ij} \ln y_j(\boldsymbol{x}_i, \boldsymbol{\theta}),$$

where $\boldsymbol{\theta}$ is the parameter vector, t_{ij} is the indicator that the *i*th sample belongs to the *j*th class, and $y_j(\boldsymbol{x}_i, \boldsymbol{\theta})$ is the output for sample *i*. The output $y_j(\boldsymbol{x}_i, \boldsymbol{\theta})$ can be interpreted as the probability that the network associates *i*th input with class *j*, that is, $P\left(t_j = 1 \middle| \boldsymbol{x}_i\right)$.

The output unit activation function is the softmax function:

$$y_r(\boldsymbol{x}) = \frac{\exp\left(a_r(\boldsymbol{x})\right)}{\sum_{j=1}^{k} \exp\left(a_j(\boldsymbol{x})\right)},$$

where $0 \le y_r \le 1$ and $\sum_{j=1}^{k} y_j = 1$.

References

[1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

See Also

`ClassificationOutputLayer` | `regressionLayer` | `softmaxLayer`

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;
```

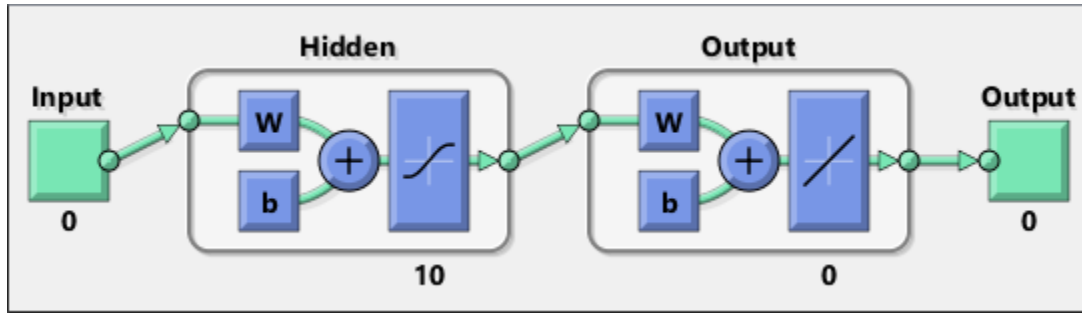
The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

Construct a function fitting neural network with one hidden layer of size 10.

```
net = fitnet(10);
```

View the network.

```
view(net)
```



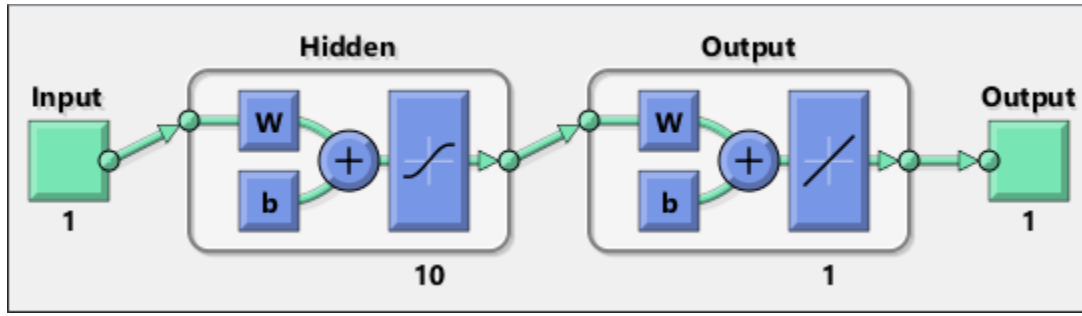
The sizes of the input and output are zero. The software adjusts the sizes of these during training according to the training data.

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.

```
net = fitnet(10,'trainbr');
net = train(net,x,t);
y = net(x);
perf = perform(net,y,t)
```

```
perf =

   3.3529e-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.

Training Function	Algorithm
`'trainlm'`	Levenberg-Marquardt
`'trainbr'`	Bayesian Regularization
`'trainbfg'`	BFGS Quasi-Newton
`'trainrp'`	Resilient Backpropagation
`'trainscg'`	Scaled Conjugate Gradient
`'traincgb'`	Conjugate Gradient with Powell/Beale Restarts
`'traincgf'`	Fletcher-Powell Conjugate Gradient
`'traincgp'`	Polak-Ribiére Conjugate Gradient
`'trainoss'`	One Step Secant
`'traingdx'`	Variable Learning Rate Gradient Descent
`'traingdm'`	Gradient Descent with Momentum
`'traingd'`	Gradient Descent

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 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')
```

```
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: 5
                        Shuffle: 'once'
                 CheckpointPath: ''
           ExecutionEnvironment: 'auto'
                     WorkerLoad: []
                      OutputFcn: []
                          Plots: 'training-progress'
                 SequenceLength: 'longest'
           SequencePaddingValue: 0
```

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”.

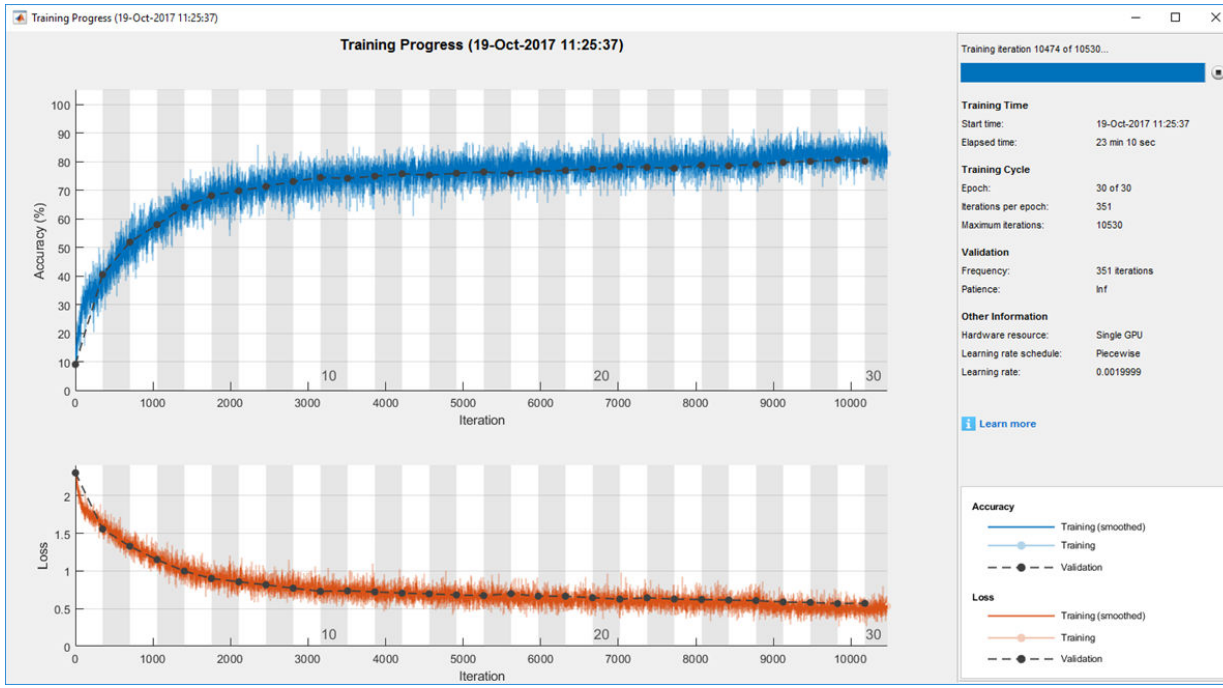
For regression networks, the figure plots the root mean square error (RMSE) instead of the accuracy.

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 labelled **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

This example shows how to 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.

```
[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.

```
layers = [
    imageInputLayer([28 28 1])
```

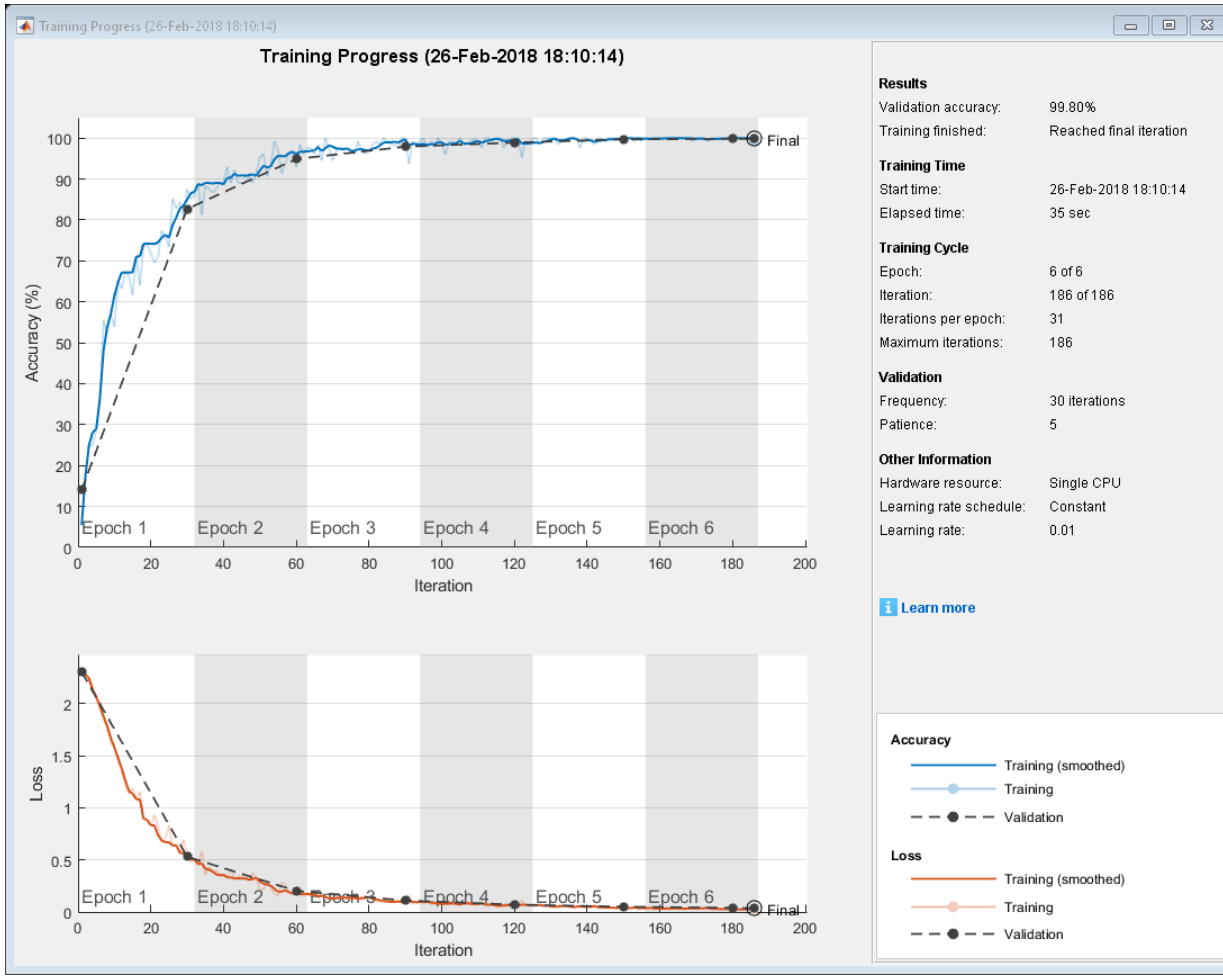
```
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    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 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.

```
options = trainingOptions('sgdm',...
    'MaxEpochs',6, ...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',30,...
    'Verbose',false,...
    'Plots','training-progress');
```

Train the network.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



- “Create Simple Deep Learning Network for Classification”
- “Transfer Learning Using AlexNet”
- “Resume Training from a Checkpoint Network”
- “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”

Input Arguments

`solverName` — Solver for training network
`'sgdm'` | `'rmsprop'` | `'adam'`

Solver to use for training the 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 1-672.

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 single 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 theL_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 1-655.

Example: `'Plots','training-progress'`

`Verbose` — Indicator to display training progress information
1 (default) | `0`

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 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). Use the `'ValidationData'` name-value pair to specify validation data.

Example: `'Verbose',0`

`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: `'VerboseFrequency',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 1-672.

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
`ImageDatastore` | mini-batch datastore | `table` | cell array

Data to use for validation during training, specified as the comma-separated pair consisting of `'ValidationData'` and one of the following:

- `ImageDatastore` with `categorical` labels for image classification problems.

- mini-batch datastore with defined responses. For more information on types of mini-batch datastores, see details on page 1-0 on the `trainNetwork` page.
- `table`, where the first column contains either image paths or images, and the subsequent columns contain the responses. For an image classification problem, the response must be a `categorical` variable in the second table column. For a regression problem, 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.
- cell array `{X,Y}`, where `X` is a numeric array of images and `Y` contains the responses. The first three dimensions of `X` are the height, width, and channels, and the last dimension is the image index. For an image classification problem, `Y` must be a `categorical` vector. For a regression problem, `Y` must be a numeric array. For more information on the allowed shape of `Y`, see details on page 1-0 on the `trainNetwork` page.

During training, `trainNetwork` predicts the labels of the validation data and calculates the validation accuracy and validation loss. To specify the validation frequency, use the `'ValidationFrequency'` name-value pair argument. By default, if the validation loss is larger than or equal to the previously smallest loss five times in a row, then network training stops. To change the number of times that the validation loss is allowed to not decrease before training stops, 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.

You cannot specify validation data when training long short-term memory (LSTM) networks.

Example: `'ValidationData',imds`

`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
5 (default) | positive integer | Inf

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 off automatic validation stopping, specify Inf as the 'ValidationPatience' value. To specify validation data, use the 'ValidationData' name-value pair argument.

Example: 'ValidationPatience',4

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_2 regularization
0.0001 (default) | nonnegative scalar

Factor for L_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 1-675.

You can specify a multiplier for the L_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 1-672.

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 β_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 1-672.

Example: `'GradientDecayFactor',0.95`

Data Types: `single` | `double`

`SquaredGradientDecayFactor` — Decay rate of squared gradient moving average
0.999 (default) | scalar from 0 to 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 scalar from 0 to 1. The squared gradient decay rate is denoted by β_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 1-672.

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 1-672.

Example: `'Epsilon',1e-6`

Data Types: `single` | `double`

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 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 1-675.

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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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`

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. If no pool is open, then the software opens one based on your default parallel settings.
- `'parallel'` — Use a local parallel pool or compute cluster. If no pool is open, then the software opens 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 the training takes place on all cluster CPUs.

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 increasing the `MiniBatchSize` training option to offset the communication overhead.

To train long short-term memory (LSTM) networks, the hardware resource must be `'auto'`, `'cpu'`, or `'gpu'`.

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)/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.

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 directory does not already 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:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`

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.

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 1-803 and “Specify Initial Weights and Biases in Fully Connected Layer” on page 1-829.

Stochastic Gradient Descent

The gradient descent algorithm updates the network parameters (weights and biases) to minimize the loss function by taking small steps in the direction of the negative gradient of the loss,

$$\boldsymbol{\theta}_{\ell+1} = \boldsymbol{\theta}_{\ell} - \alpha \nabla E(\boldsymbol{\theta}_{\ell}),$$

where ℓ stands for the iteration number, $\alpha > 0$ is the learning rate, $\boldsymbol{\theta}$ is the parameter vector, and $E(\boldsymbol{\theta})$ is the loss function. The gradient of the loss function, $\nabla E(\boldsymbol{\theta})$, is evaluated using the entire training set, and the standard gradient descent algorithm uses the entire data set at once.

The stochastic gradient descent algorithm evaluates the gradient and updates the parameters using a subset of the training set. This subset is called a mini-batch. Each evaluation of the gradient using the mini-batch is an iteration. At each iteration, the algorithm takes one step towards minimizing the loss function. The full pass of the training algorithm over the entire training set using mini-batches is an epoch. You can specify the mini-batch size and the maximum number of epochs using the `'MiniBatchSize'` and `'MaxEpochs'` name-value pair arguments, respectively.

The stochastic gradient descent algorithm might 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 [2]. The stochastic gradient descent with momentum update is

$$\theta_{\ell+1} = \theta_{\ell} - \alpha \nabla E(\theta_{\ell}) + \gamma(\theta_{\ell} - \theta_{\ell-1}),$$

where γ 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 use stochastic gradient descent with momentum to train a neural network, specify `solverName` as `'sgdm'`. To specify the initial value of the learning rate α, 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”.

Stochastic gradient descent with momentum uses a single learning rate for all the parameters. Other optimization algorithms try to improve network training by using learning rates that are different for different parameters and that 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,

$$v_{\ell} = \beta_2 v_{\ell-1} + (1-\beta_2)[\nabla E(\theta_{\ell})]^2$$

β_2 is the decay rate of the moving average. Typical values of the decay rate are 0.9, 0.99, and 0.999. The corresponding averaging lengths of the squared gradients are given by $1/(1-\beta_2)$, that is, 10, 100, and 1000 parameter updates, respectively. You can specify β_2 by using the `'SquaredGradientDecayFactor'` name-value pair argument. The RMSProp algorithm uses this moving average to normalize the updates of each parameter individually,

$$\theta_{\ell+1} = \theta_{\ell} - \frac{\alpha \nabla E(\theta_{\ell})}{\sqrt{v_l} +}$$

where the division is performed element-wise. With RMSProp, parameters with large gradients have their learning rates effectively reduced, and parameters with small gradients have their learning rates effectively increased. ε is a small constant added to avoid division by zero. You can specify ε 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 (derived from *adaptive moment estimation*) [4] uses a parameter update that is similar to RMSProp with momentum. It keeps an element-wise moving average of both the parameter gradients and their squared values,

$$m_{\ell} = \beta_1 m_{\ell-1} + (1-\beta_1)\nabla E(\theta_{\ell})$$

$$v_{\ell} = \beta_2 v_{\ell-1} + (1-\beta_2)[\nabla E(\theta_{\ell})]^2$$

You can specify the β_1 and β_2 decay rates using the `'GradientDecayFactor'` and `'SquaredGradientDecayFactor'` name-value pair arguments, respectively. Adam uses these averages to update the network parameters as

$$\theta_{\ell+1} = \theta_{\ell} - \frac{\alpha m_l}{\sqrt{v_l} +}$$

If gradients over many iterations are similar, then using a moving average of the gradient lets the parameter updates pick up momentum in a certain direction. If the gradients contain mostly noise, then the moving average of the gradient becomes smaller, and so also the parameter updates become smaller. You can specify ε 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 α 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 different learning rates for different layers and parameters. 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.

L_2 Regularization

Adding a regularization term for the weights to the loss function $E(\boldsymbol{\theta})$ 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

$$E_R(\boldsymbol{\theta}) = E(\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{w}),$$

where $\boldsymbol{w}$ is the weight vector, λ is the regularization factor (coefficient), and the regularization function $\Omega(\boldsymbol{w})$ is

$$\Omega(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}.$$

Note that the biases are not regularized [2]. You can specify the regularization factor λ 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”.

References

[1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

[2] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, Cambridge, Massachusetts, 2012.

[3] Pascanu, R., T. Mikolov, and Y. Bengio. "On the difficulty of training recurrent neural networks". *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28(3), 2013, pp. 1310–1318.

[4] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

See Also

`trainNetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from a Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Learn About Convolutional Neural Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

Introduced in R2016a

trainNetwork

Train neural network for deep learning

Use `trainNetwork` to train a convolutional neural network (ConvNet, CNN) or long short-term memory networks (LSTM or BiLSTM networks) 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`.

Syntax

```
trainedNet = trainNetwork(imds,layers,options)
trainedNet = trainNetwork(mbds,layers,options)

trainedNet = trainNetwork(X,Y,layers,options)
trainedNet = trainNetwork(C,Y,layers,options)

trainedNet = trainNetwork(tbl,layers,options)
trainedNet = trainNetwork(tbl,responseName,layers,options)

[trainedNet,traininfo] = trainNetwork( ___ )
```

Description

`trainedNet = trainNetwork(imds,layers,options)` trains a network for image classification problems. `imds` stores the input image data, `layers` defines the network architecture, and `options` defines the training options.

`trainedNet = trainNetwork(mbds,layers,options)` trains a network for image classification and regression problems. `mdbs` is a type of mini-batch datastore, such as an augmented image datastore, that preprocesses images for deep learning.

`trainedNet = trainNetwork(X,Y,layers,options)` trains a network for image classification and regression problems. `X` contains the predictor variables and `Y` contains the categorical labels or numeric responses.

`trainedNet = trainNetwork(C,Y,layers,options)` trains an LSTM or BiLSTM network for classification and regression problems. `C` is a cell array containing sequence or time series predictors and `Y` contains the responses. For classification problems, `Y` is a categorical vector or cell array of categorical sequences. For regression problems, `Y` is a matrix of targets, or a cell array of numeric sequences.

`trainedNet = trainNetwork(tbl,layers,options)` trains a network for classification and regression problems. `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 the “tbl” on page 1-0 argument description.

`trainedNet = 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 the table `tbl`.

`[trainedNet,traininfo] = trainNetwork( ___ )` also returns information on the training for any of the input arguments.

Examples

Train Network for Image Classification

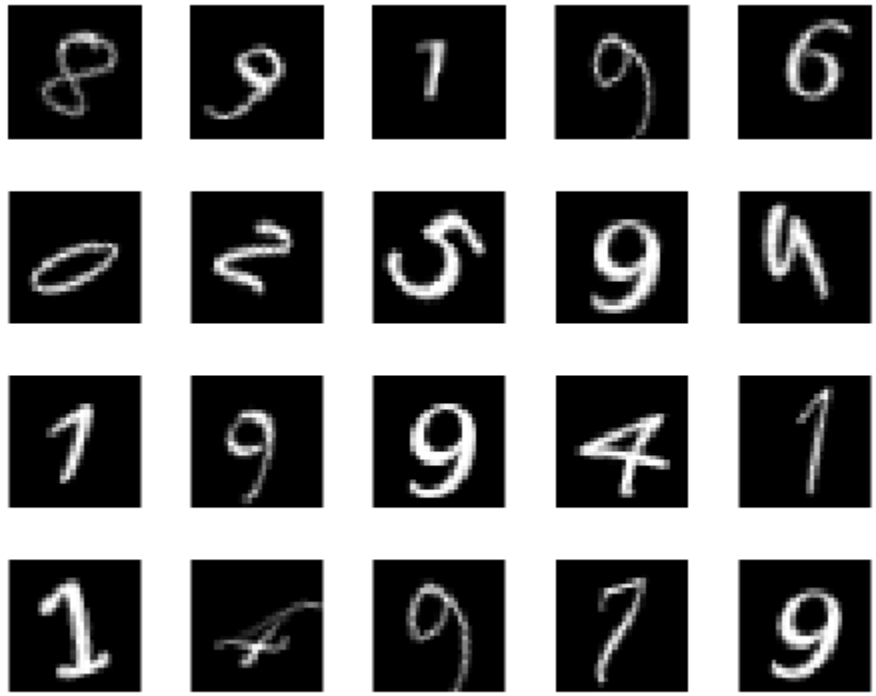
Load the data as an `ImageDatastore` object.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

The datastore contains 10,000 synthetic images of digits 0-9. 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.

```
figure
numImages = 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.

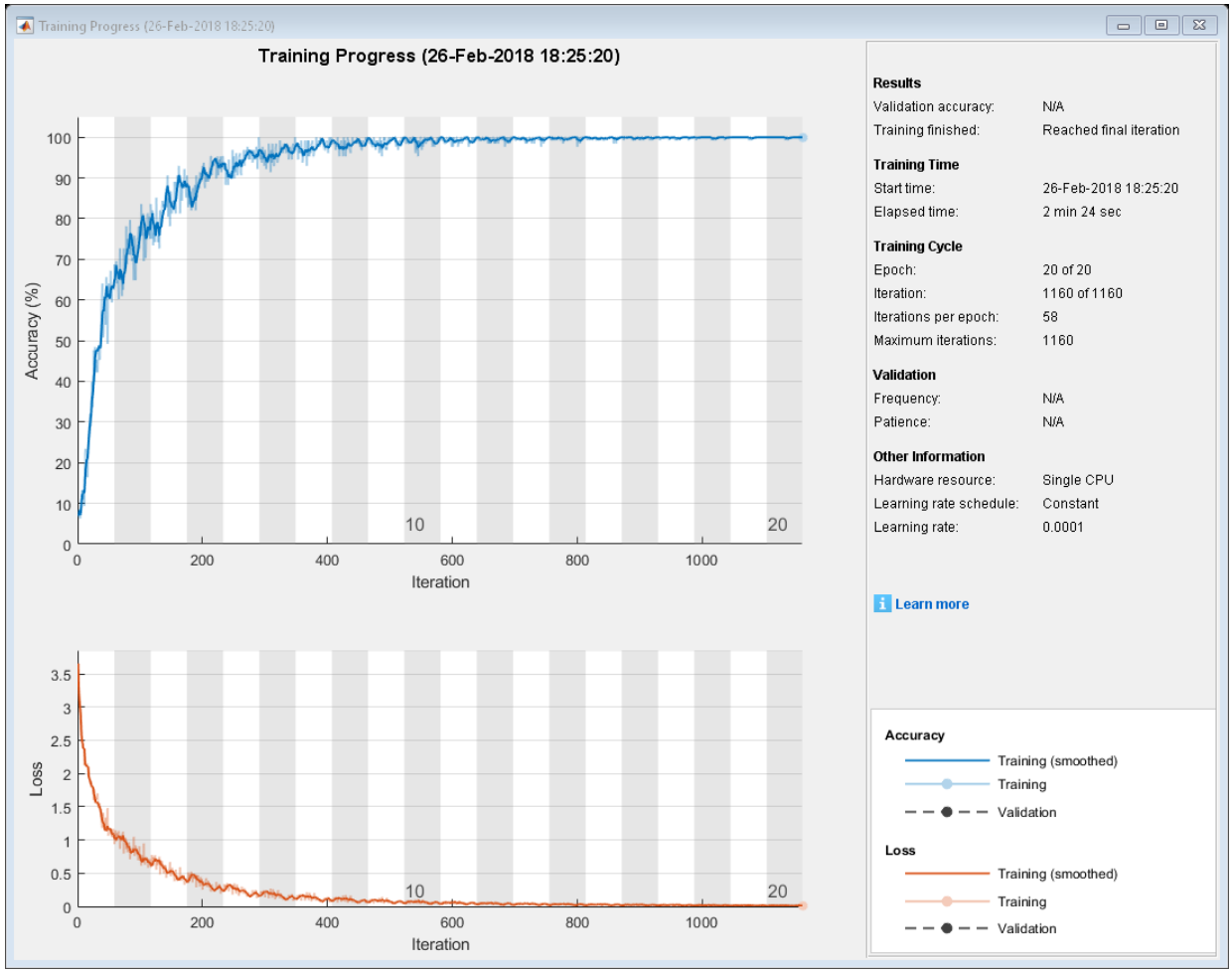
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

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',0, ...
    '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).

```
YPred = classify(net,imdsTest);
YTest = imdsTest.Labels;
```

Calculate the accuracy. 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 = 0.9896
```

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])
```

```
imageAugmenter =
  imageDataAugmenter with properties:
```

```
           FillValue: 0
      RandXReflection: 0
      RandYReflection: 0
         RandRotation: [-20 20]
           RandXScale: [1 1]
           RandYScale: [1 1]
           RandXShear: [0 0]
           RandYShear: [0 0]
     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 'on the fly' and does not save any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.

```
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugme
```

Specify the convolutional neural network architecture.

```
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)
```

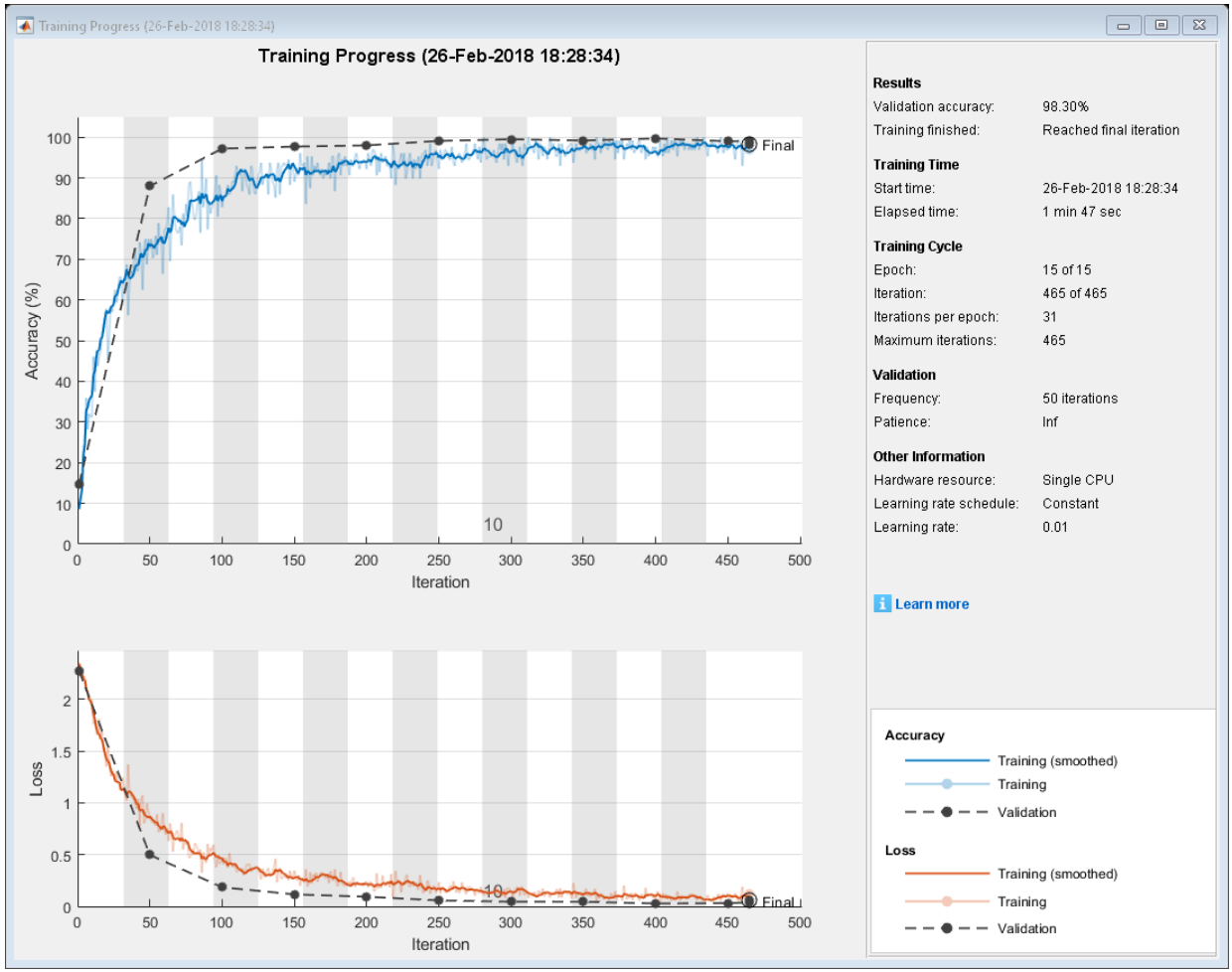
```
    softmaxLayer
    classificationLayer];
```

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},...
    'ValidationPatience',Inf);
```

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:

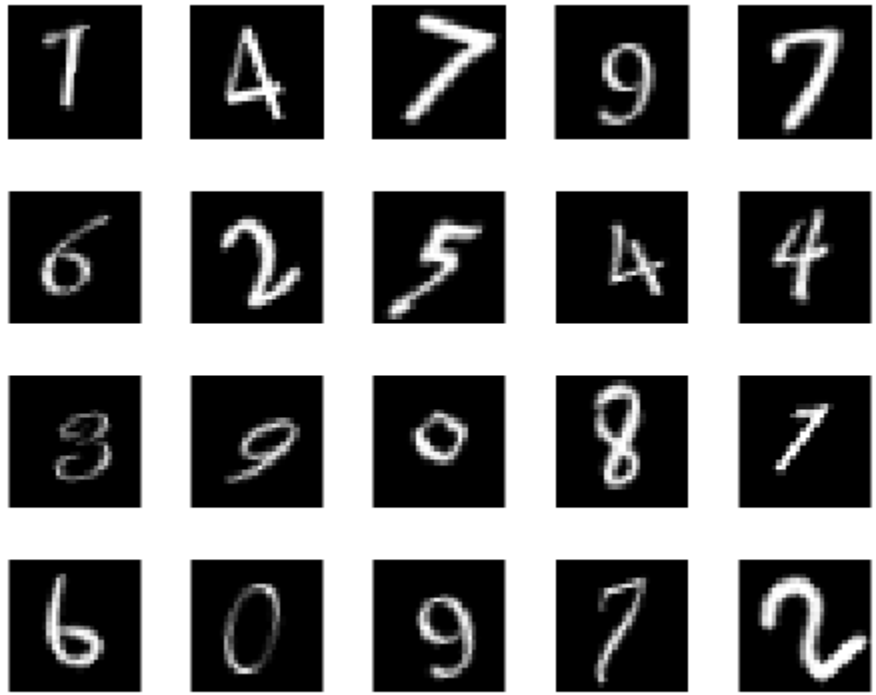
- 28 is the height and width of the images.
- 1 is the number of channels.
- 5000 is the number of synthetic images of handwritten digits.

`YTrain` contains the rotation angles in degrees.

```
[XTrain,~,YTrain] = digitTrain4DArrayData;
```

Display 20 random training images using `imshow`.

```
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.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(12,25)
    reluLayer
    fullyConnectedLayer(1)
    regressionLayer];
```

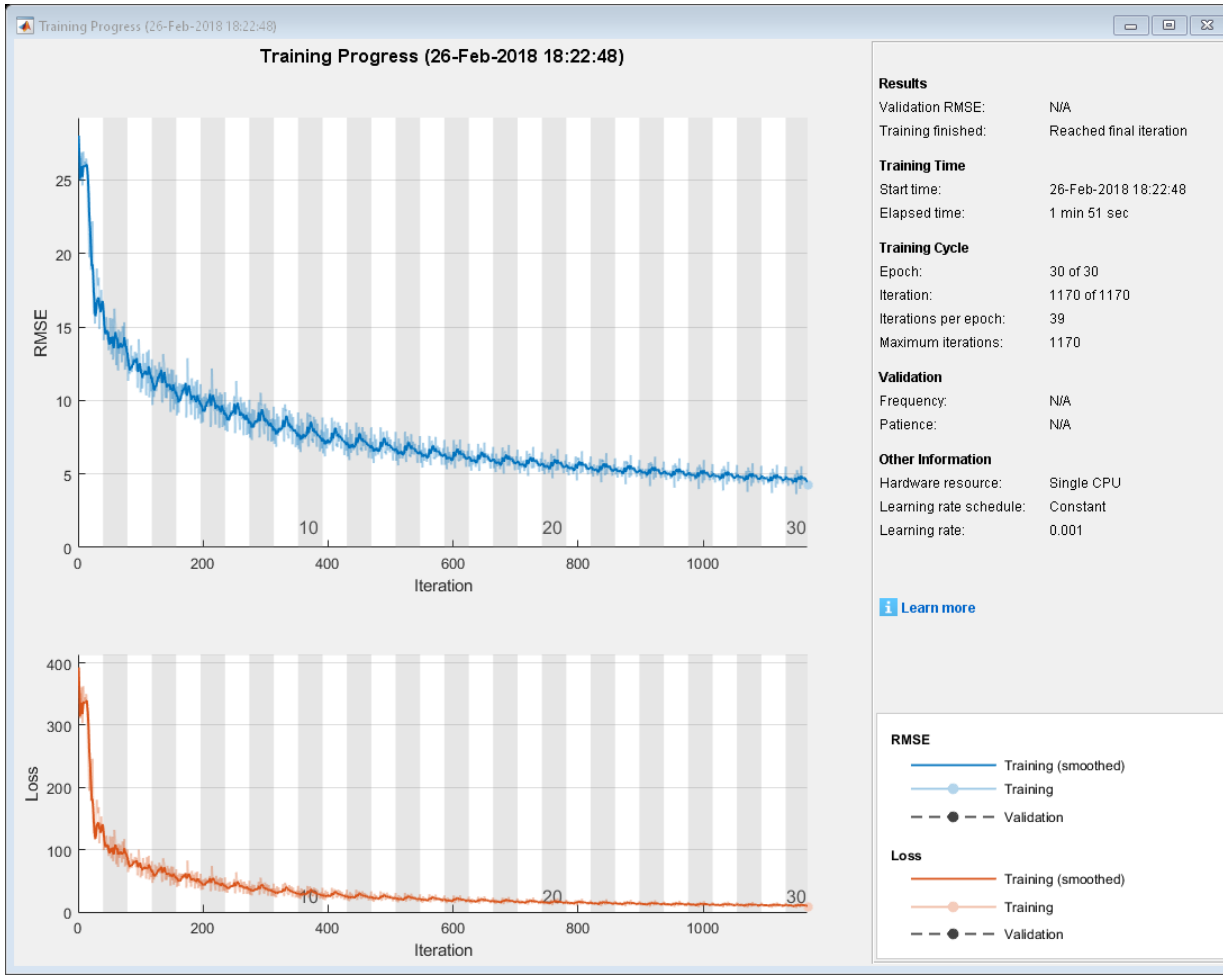
Specify the network training options. Set the initial learn rate to 0.001.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.001, ...
```

```
    'Verbose',false, ...
    'Plots','training-progress');
```

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.

```
rmse = sqrt(mean((YTest - YPred).^2))
```

```
rmse = single
    6.4026
```

Train Network for Sequence Classification

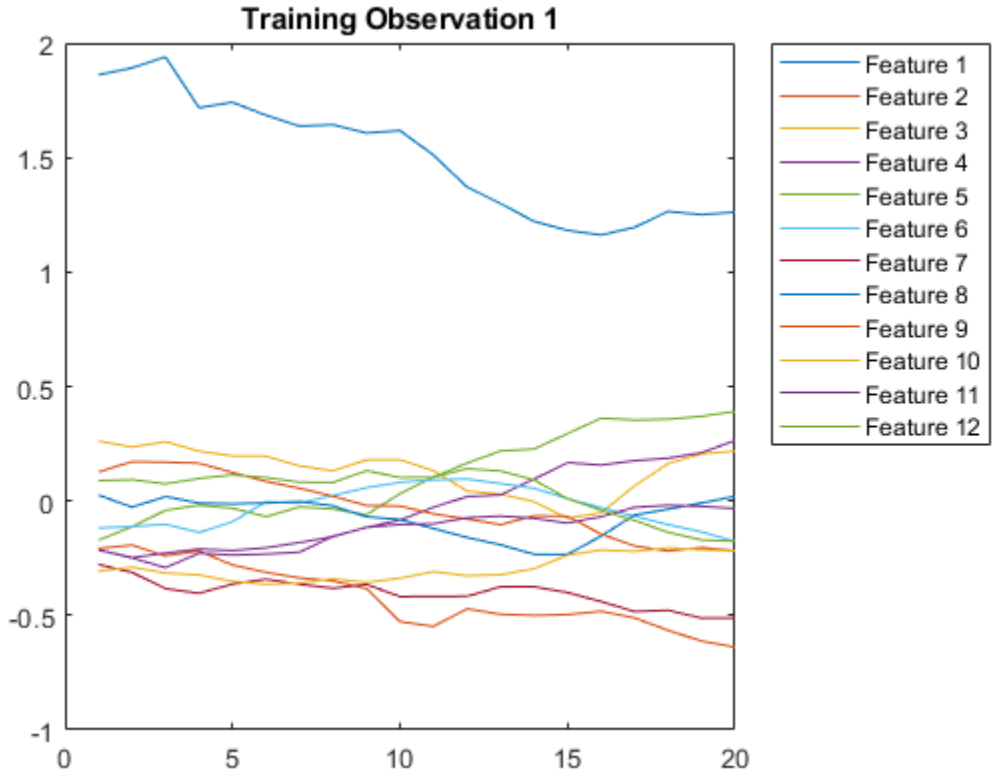
Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with feature dimension 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.

```
figure
plot(XTrain{1}')
title("Training Observation 1")
legend("Feature " + string(1:12),'Location','northeastoutside')
```



Define the LSTM network architecture. Specify the input size 12 (the dimension of the input data). Specify an LSTM layer to have 100 hidden units and output the last element of the sequence. Finally, specify 9 classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```
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
```

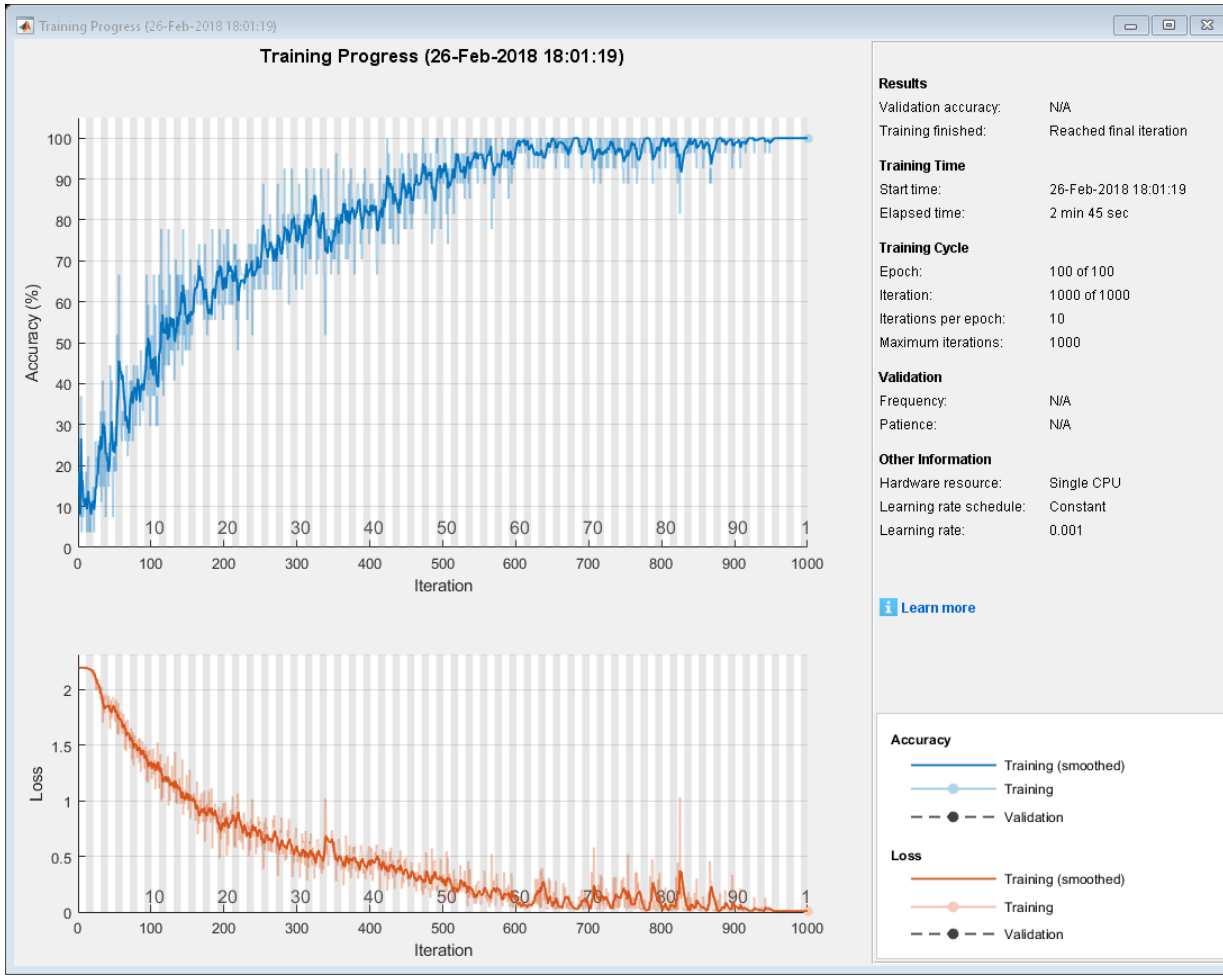
Specify the training options. Specify the solver to be `'adam'` and `'GradientThreshold'` to be 1. Set the mini-batch size to 27, and set the maximum number of epochs to 100.

Because the mini-batches are small with short sequences, training is better suited for the CPU. Specify `'ExecutionEnvironment'` to be `'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',0, ...
    '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. Set the mini-batch size to 27.

```
miniBatchSize = 27;
YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9270
```

- “Create Simple Deep Learning Network for Classification”
- “Transfer Learning Using AlexNet”
- “Train Convolutional Neural Network for Regression”
- “Sequence Classification Using Deep Learning”

Input Arguments

`imds` — Images with labels

`ImageDatastore` object

Images with labels, specified as an `ImageDatastore` object with `categorical` labels. You can store data in `ImageDatastore` for image classification networks only.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, prefetching does not happen.

`mbds` — Mini-batch datastore

custom `MiniBatchable` datastore | `augmentedImageDatastore` | ...

Mini-batch datastore for image preprocessing, specified as one of the following:

- An `augmentedImageDatastore`, which preprocesses images for deep learning. For example, an augmented image datastore can resize, rotate, and reflect input images.
- A `denoisingImageDatastore`, which preprocesses images for use in training denoising networks. For example, a denoising image datastore can add Gaussian noise to input images.
- A `pixelLabelImageDatastore`, which specifies inputs and responses when you train semantic segmentation networks.
- A custom implementation of a `MiniBatchable` datastore. For more information, see “Develop Custom Mini-Batch Datastore”.

`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 `NaN`s, 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`

`C` — Sequences or time series data

cell array of matrices | matrix

Sequences or time series data, specified as a cell array of matrices, or a matrix. For cell array input, `C` is an *N*-by-1 cell array where *N* is the number of observations. Each entry of `C` is a time series represented by a matrix with rows corresponding to data points, and columns corresponding to time steps.

For sequence-to-sequence problems with one observation, `C` can be a *D*-by-*S* matrix, where *D* is the number of features, and *S* is the number of time steps. If `C` is a matrix, then `Y` must be a categorical sequence of labels or a matrix of responses.

For sequence classification and regression problems, `layers` must begin with a sequence input layer.

Data Types: `cell` | `single` | `double`

`Y` — Responses

categorical vector of labels | matrix | 4-D numeric array | cell array of categorical row vectors | cell array of numeric sequences

Responses, specified as a categorical vector of labels, matrix, 4-D numeric array, cell array of categorical row vectors, or cell array of numeric sequences. The format of `Y` depends on the type of problem.

The following table describes the format for classification problems.

Task	Format
Image classification	*N*-by-1 categorical vector of labels, where *N* is the number of observations.
Sequence-to-label classification	

Task	Format
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.

For sequence-to-sequence classification problems with one observation, `C` can be a matrix. In this case, `Y` must be a categorical sequence of labels.

The following table describes the format for regression problems.

Task	Format
Image Regression	• *N*-by-*r* matrix, where *N* is the number of observations and *r* is the number of responses. • *h*-by-*w*-by-*c*-by-*N* numeric array, where *N* is the number of observations and *h*-by-*w*-by-*c* is the image size of a single response.
Sequence-to-one regression	*N*-by-*r* matrix, where *N* is the number of observations 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 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.

For sequence-to-sequence regression problems with one observation, `C` 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”.

Responses cannot contain `NaN`s.

Data Types: `cell` | `categorical` | `double`

`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. The following table describes the predictors and responses.

Classification

Task	Predictors	Responses
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 classification	Absolute or relative file path to a MAT file containing sequence or time series data. The MAT file must contain a time series represented by a matrix with rows corresponding to data points, and columns corresponding to time steps.	Categorical label
Sequence-to-sequence classification		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.

For classification problems, if you do not specify `responseName`, then the function, by default, uses the responses in the second column of `tbl`.

Regression

Task	Predictors	Responses
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. The MAT file must contain a time series represented by a matrix with rows corresponding to data points, and columns corresponding to time steps.	• One or more columns of scalar values • Numeric vector
Sequence-to-sequence regression		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”.

For sequence classification and regression problems, `layers` must begin with a sequence input layer.

Responses cannot contain `NaN`s. If the predictor data contains `NaN`s, then they are propagated through the training. However, the training usually fails to converge.

Data Types: `table`

`responseName` — Names of response variables in the input table
character vector | cell array of character vectors

Names of the response variables in the input table, specified as a character vector or cell array of character vectors. For problems with one response, `responseName` is the corresponding variable name in `tbl`. For regression problems with multiple response variables, `responseName` is a cell array of the corresponding variable names in `tbl`.

Data Types: `char` | `cell`

`layers` — Network layers
`Layer` array | `LayerGraph` object

Network layers, specified as a `Layer` array or a `LayerGraph` object.

To train a network with all layers connected sequentially, you can use a `Layer` array as the input argument. In this case, the returned trained network is a `SeriesNetwork` object.

A directed acyclic graph (DAG) network has a more complex structure in which layers can have multiple inputs and outputs. To train a DAG network, specify the network architecture as a `LayerGraph` object and then use that layer graph as the input argument to `trainNetwork`.

`options` — Training options
`TrainingOptionsSGDM` | `TrainingOptionsRMSProp` | `TrainingOptionsADAM`

Training options, specified as a `TrainingOptionsSGDM`, `TrainingOptionsRMSProp`, or `TrainingOptionsADAM` object returned by the `trainingOptions` function. Use `trainingOptions` to specify solver and other training options to use for network training.

Output Arguments

`trainedNet` — 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 `trainedNet` is a `SeriesNetwork` object. If you train the network using a `LayerGraph` object as the input argument, then `trainedNet` is a `DAGNetwork` object.

`traininfo` — Training information
structure

Training information for each iteration, returned as a structure with a combination of the following fields:

- `TrainingLoss` — Loss function value at each iteration
- `TrainingAccuracy` — Training accuracy at each iteration
- `TrainingRMSE` — Training RMSE at each iteration
- `ValidationLoss` — Loss function value for validation data
- `ValidationAccuracy` — Validation accuracy
- `ValidationRMSE` — Validation RMSE
- `BaseLearnRate` — Learning rate at each iteration

`trainNetwork` returns accuracy values for classification networks, RMSE values for regression networks, and validation metrics when you validate the network during training. Each field is a numeric vector with one element per training iteration. Values that have not been calculated at a specific iteration are represented by `NaN`.

Definitions

Save Checkpoint Networks and Resume Training

Neural Network 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, for example, `convnet_checkpoint__351__2016_11_09__12_04_23.mat`. In this example, 351 is the iteration number, 2016_11_09 is the date, and 12_04_23 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 convnet_checkpoint__351__2016_11_09__12_04_23.mat
```

You can then resume training by using the layers of the network as an input argument to `trainNetwork`. For example:

```
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 a Checkpoint Network”.

Floating-Point Arithmetic

All functions for deep learning training, prediction, and validation in Neural Network 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

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`DAGNetwork` | `LayerGraph` | `SeriesNetwork` | `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”

Introduced in R2016a

deepDreamImage

Visualize network features using deep dream

Syntax

```
I = deepDreamImage(net,layer,channels)
I = deepDreamImage(net,layer,channels,Name,Value)
```

Description

`I = deepDreamImage(net,layer,channels)` returns an array of images that strongly activate the channels `channels` within the network `net` of the layer with numeric index or name given by `layer`. These images highlight the features learned by a network.

`I = deepDreamImage(net,layer,channels,Name,Value)` returns an image with additional options specified by one or more `Name,Value` pair arguments.

Examples

Visualize Convolutional Neural Network Features

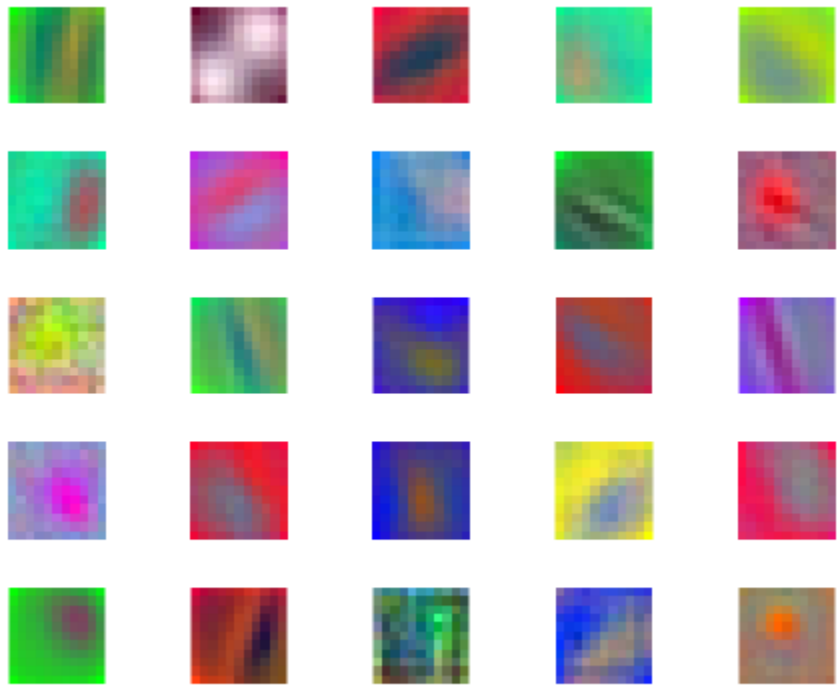
Load a pretrained AlexNet network.

```
net = alexnet;
```

Visualize the first 25 features learned by the first convolutional layer (`'conv1'`) using `deepDreamImage`. Set `'PyramidLevels'` to 1 so that the images are not scaled.

```
layer = 'conv1';
channels = 1:25;

I = deepDreamImage(net,layer,channels, ...
    'PyramidLevels',1, ...
```

```
    'Verbose',0);

figure
for i = 1:25
    subplot(5,5,i)
    imshow(I(:,:,:,i))
end
```



- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”
- “Deep Dream Images Using AlexNet”
- “Visualize Features of a Convolutional Neural Network”

- “Visualize Activations of a Convolutional Neural Network”

Input Arguments

`net` — Trained network

`SeriesNetwork` object

Trained network, specified as a `SeriesNetwork` 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 Convolutional Neural Networks”.

`deepDreamImage` only supports networks with an image input layer.

`layer` — Layer index or name

numeric index | character vector

Layer to visualize, specified as a positive integer scalar or character vector. 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 single

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:

Value	Description
`'linear'`	Scale output pixel values in the interval [0,1]. The output image corresponding to each layer channel, `I(:,:,:,channel)`, is scaled independently.
`'none'`	Disable output scaling.

Example: `'OutputScaling','linear'`

Data Types: `char`

`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 Neural Network 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

[1] *DeepDreaming with TensorFlow*. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/deepdream/deepdream.ipynb

See Also

`activations` | `alexnet` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Deep Dream Images Using AlexNet”
“Visualize Features of a Convolutional Neural Network”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2017a

regressionLayer

Create a regression output layer

Syntax

```
routputlayer = regressionLayer
routputlayer = regressionLayer('Name',Name)
```

Description

`routputlayer = 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”.

`routputlayer = regressionLayer('Name',Name)` returns a regression layer with the name specified by `Name`.

Examples

Create Regression Output Layer

Create a regression output layer with the name `'routput'`.

```
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
```

- “Deep Learning in MATLAB”
- “Train Convolutional Neural Network for Regression”

Input Arguments

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as the comma-separated pair consisting of `'Name'` and a character vector. If you do not specify a name, then the software initially specifies the default value `''`, and automatically assigns the name `'regressionoutputlayer'` at training time.

Example: `'Name','routput'`

Data Types: `char`

Output Arguments

routputlayer — Regression output layer
RegressionOutputLayer object

Regression output layer, returned as a RegressionOutputLayer object.

See Also

FullyConnectedLayer | RegressionOutputLayer | classificationLayer

Topics

“Deep Learning in MATLAB”
“Train Convolutional Neural Network for Regression”

Introduced in R2017a

alexnet

Pretrained AlexNet convolutional neural network

You can download and install pretrained networks to use for your problems. Use functions such as `alexnet` to get links to download pretrained networks from the Add-On Explorer. For more pretrained networks in MATLAB, see “Pretrained Convolutional Neural Networks”.

For a free hands-on introduction to practical deep learning methods, see Deep Learning Onramp.

Syntax

```
net = alexnet
```

Description

`net = alexnet` returns a pretrained AlexNet model. This model is trained on a subset of the ImageNet database [1], which is used in ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [2]. The model 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.

This function requires Neural Network Toolbox Model *for AlexNet Network* support package. If this support package is not installed, the function provides a download link. Alternatively, see Neural Network Toolbox Model *for AlexNet Network*.

For more pretrained networks in MATLAB, see “Pretrained Convolutional Neural Networks”.

Examples

Download AlexNet Support Package

Download and install Neural Network Toolbox Model *for AlexNet Network* support package.

Type `alexnet` at the command line.

```
alexnet
```

If Neural Network 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.

```
alexnet
```

```
ans =

  SeriesNetwork with properties:

    Layers: [25×1 nnet.cnn.layer.Layer]
```

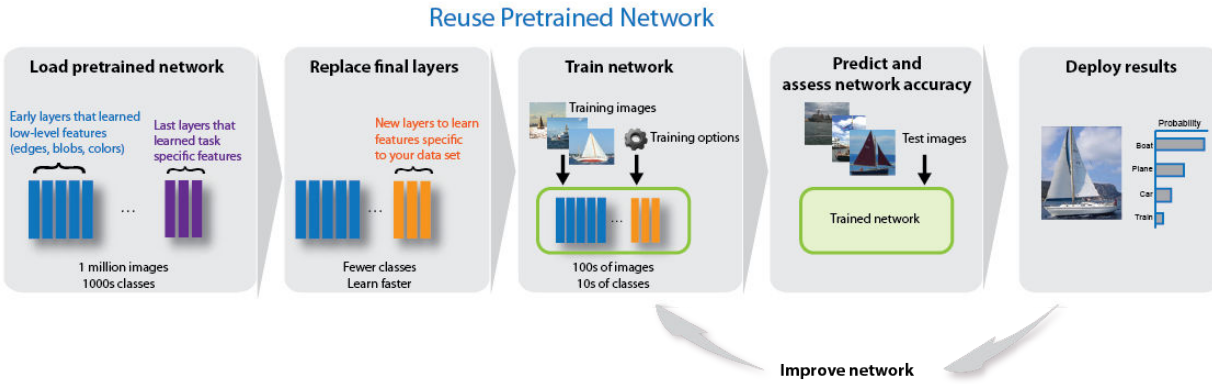
If the required support package is installed, then the function returns a `SeriesNetwork` object.

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.

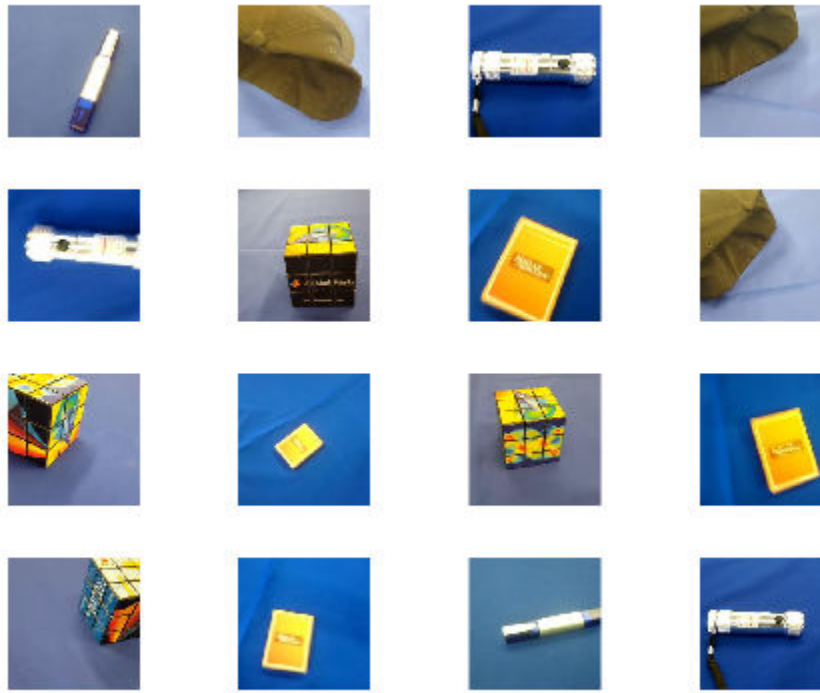
```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

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.

```
[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.

```
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 Neural Network 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;
```

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                   227x227x3 images with 'zerocenter' no
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 ch
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] an
     6   'conv2'    Convolution                   256 5x5x48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 ch
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] an
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Convolution                   384 3x3x192 convolutions with stride
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Convolution                   256 3x3x192 convolutions with stride
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] an
    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
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench' and 999 o
```

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.

```
inputSize = net.Layers(1).InputSize
```

```
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.

```
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.

```
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.

```
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.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

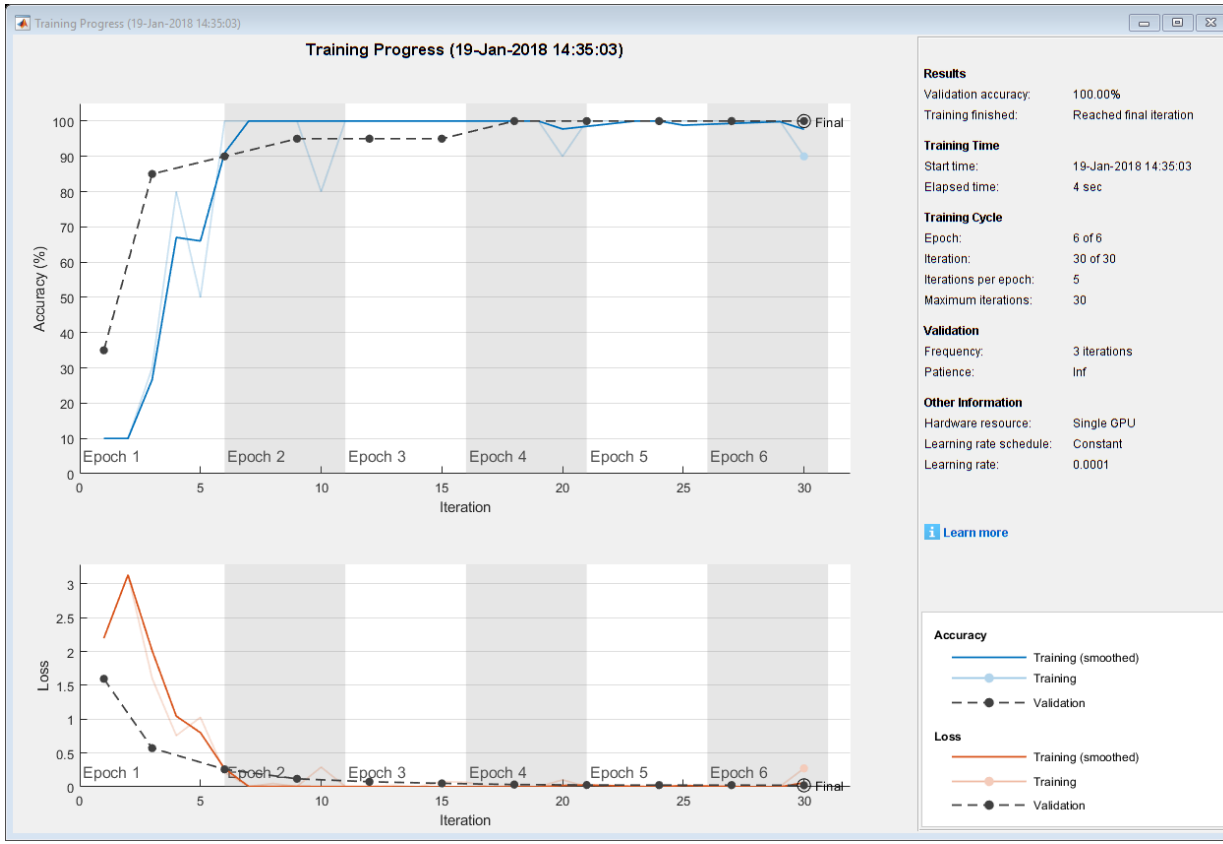
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the

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.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',1e-4, ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'ValidationPatience',Inf, ...
    '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`.

```
netTransfer = trainNetwork(augimdsTrain,layers,options);
```



Classify Validation Images

Classify the validation images using the fine-tuned network.

```
[YPred,scores] = classify(netTransfer,augimdsValidation);
```

Display four sample validation images with their predicted labels.

```
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));
end
```

Calculate the classification accuracy on the validation set. Accuracy is the fraction of labels that the network predicts correctly.

```
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 1
```

This trained network has high accuracy. If the accuracy is not high enough using transfer learning, then try feature extraction instead.

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 =

   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.

```
figure
imshow(I)
title(char(label))
```

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.

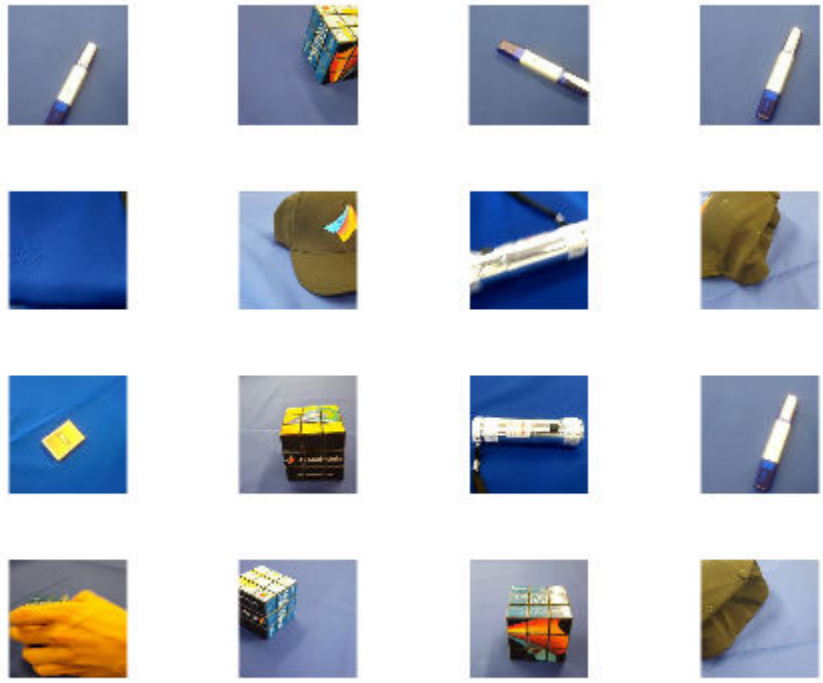
Load Data

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.

```
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 a pretrained AlexNet network. If Neural Network Toolbox Model *for AlexNet Network* 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                   227x227x3 images with 'zerocenter' no
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 ch
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] an
     6   'conv2'    Convolution                   256 5x5x48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 ch
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] an
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Convolution                   384 3x3x192 convolutions with stride
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Convolution                   256 3x3x192 convolutions with stride
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] an
    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
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench' and 999 o
```

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.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize =

   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`.

```
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.

```
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).

```
classifier = fitcecoc(featuresTrain,YTrain);
```

Classify Test Images

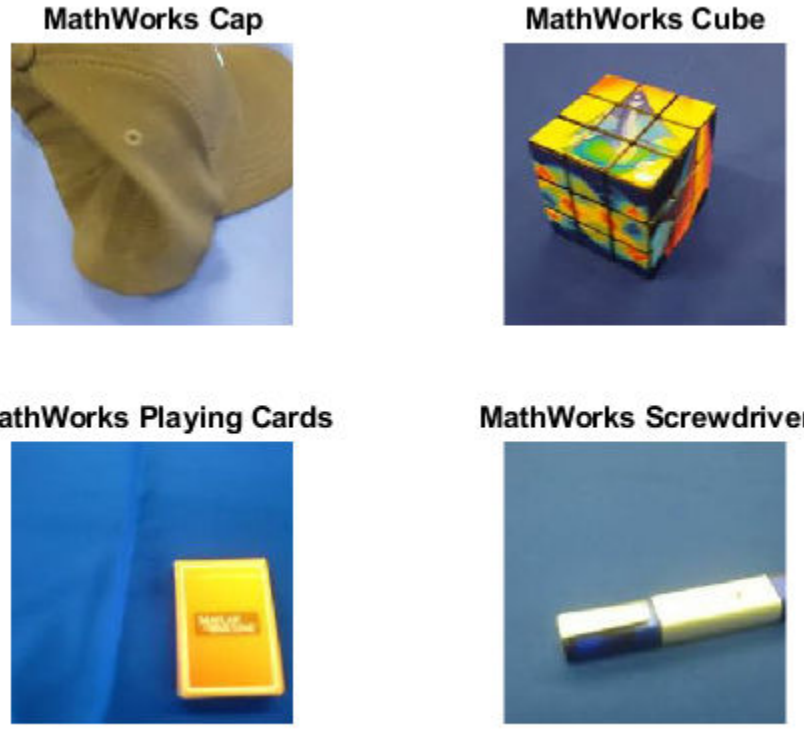
Classify the test images using the trained SVM model the features extracted from the test images.

```
YPred = predict(classifier,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(char(label))
end
```



Calculate the classification accuracy on the test set. Accuracy is the fraction of labels that the network predicts correctly.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 1
```

This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

- “Deep Learning in MATLAB”

- “Classify Webcam Images Using Deep Learning”
- “Pretrained Convolutional Neural Networks”
- “Transfer Learning Using GoogLeNet”
- “Deep Dream Images Using AlexNet”
- “Visualize Features of a Convolutional Neural Network”
- “Visualize Activations of a Convolutional Neural Network”

Output Arguments

net — Pretrained AlexNet convolutional neural network
`SeriesNetwork` object

Pretrained AlexNet convolutional neural network returned as a `SeriesNetwork` object.

Definitions

Filter Groups

In the original implementation of AlexNet, to split the network across two limited-memory GPUs for training, some convolutional layers use *filter groups*.

In these layers, the filters are split into two groups. The layer splits the input into two sections along the channel dimension, and then applies each filter group to a different section. The layer then concatenates the two resulting sections together to produce the output.

For example, in the second convolutional layer in AlexNet, the layer splits the weights into two groups of 128 filters. Each filter has 48 channels. The input to the layer has 96 channels, and is split into two sections with 48 channels. The layer applies each group of filters to a different section, and produces two outputs with 128 channels. The layer then concatenates these two outputs to give a final output with 256 channels.

Networks trained on newer hardware, for example, `googlenet` and `resnet50`, do not require filter groups for training.

Tips

- For a free hands-on introduction to practical deep learning methods, see Deep Learning Onramp.

References

[1] *ImageNet*. http://www.image-net.org

[2] Russakovsky, O., Deng, J., Su, H., et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision (IJCV)*. Vol 115, Issue 3, 2015, pp. 211–252

[3] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in neural information processing systems*. 2012.

[4] *BVLC AlexNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

See Also

`deepDreamImage` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `resnet50` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Classify Webcam Images Using Deep Learning”
“Pretrained Convolutional Neural Networks”
“Transfer Learning Using GoogLeNet”
“Deep Dream Images Using AlexNet”
“Visualize Features of a Convolutional Neural Network”
“Visualize Activations of a Convolutional Neural Network”

Introduced in R2017a

vgg16

Pretrained VGG-16 convolutional neural network

Syntax

```
net = vgg16
```

Description

`net = vgg16` returns a pretrained VGG-16 model. This model is trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [2]. VGG-16 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.

This function requires Neural Network 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 Neural Network Toolbox Model *for VGG-16 Network* support package.

Type `vgg16` at the command line.

```
vgg16
```

If Neural Network 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: [41×1 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: [41×1 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' normali
     2   'conv1_1'   Convolution             64 3x3x3 convolutions with stride [1  1] a
     3   'relu1_1'   ReLU                    ReLU
```

```
 4   'conv1_2'   Convolution             64 3x3x64 convolutions with stride [1  1]
 5   'relu1_2'   ReLU                    ReLU
 6   'pool1'     Max Pooling             2x2 max pooling with stride [2  2] and pad
 7   'conv2_1'   Convolution             128 3x3x64 convolutions with stride [1  1]
 8   'relu2_1'   ReLU                    ReLU
 9   'conv2_2'   Convolution             128 3x3x128 convolutions with stride [1  1
10   'relu2_2'   ReLU                    ReLU
11   'pool2'     Max Pooling             2x2 max pooling with stride [2  2] and pad
12   'conv3_1'   Convolution             256 3x3x128 convolutions with stride [1  1
13   'relu3_1'   ReLU                    ReLU
14   'conv3_2'   Convolution             256 3x3x256 convolutions with stride [1  1
15   'relu3_2'   ReLU                    ReLU
16   'conv3_3'   Convolution             256 3x3x256 convolutions with stride [1  1
17   'relu3_3'   ReLU                    ReLU
18   'pool3'     Max Pooling             2x2 max pooling with stride [2  2] and pad
19   'conv4_1'   Convolution             512 3x3x256 convolutions with stride [1  1
20   'relu4_1'   ReLU                    ReLU
21   'conv4_2'   Convolution             512 3x3x512 convolutions with stride [1  1
22   'relu4_2'   ReLU                    ReLU
23   'conv4_3'   Convolution             512 3x3x512 convolutions with stride [1  1
24   'relu4_3'   ReLU                    ReLU
25   'pool4'     Max Pooling             2x2 max pooling with stride [2  2] and pad
26   'conv5_1'   Convolution             512 3x3x512 convolutions with stride [1  1
27   'relu5_1'   ReLU                    ReLU
28   'conv5_2'   Convolution             512 3x3x512 convolutions with stride [1  1
29   'relu5_2'   ReLU                    ReLU
30   'conv5_3'   Convolution             512 3x3x512 convolutions with stride [1  1
31   'relu5_3'   ReLU                    ReLU
32   'pool5'     Max Pooling             2x2 max pooling with stride [2  2] and pad
33   'fc6'       Fully Connected         4096 fully connected layer
34   'relu6'     ReLU                    ReLU
35   'drop6'     Dropout                 50% dropout
36   'fc7'       Fully Connected         4096 fully connected layer
37   'relu7'     ReLU                    ReLU
38   'drop7'     Dropout                 50% dropout
39   'fc8'       Fully Connected         1000 fully connected layer
40   'prob'      Softmax                 softmax
41   'output'    Classification Output   crossentropyex with 'tench', 'goldfish', a
```

To view the names of the classes learned by the network, you can view the `ClassNames` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

```
net.Layers(end).ClassNames(1:10)
```

```
ans =

  10×1 cell array

    'tench'
    'goldfish'
    'great white shark'
    'tiger shark'
    'hammerhead'
    'electric ray'
    'stingray'
    'cock'
    'hen'
    'ostrich'
```

- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”
- “Transfer Learning Using AlexNet”
- “Visualize Activations of a Convolutional Neural Network”

Output Arguments

net — Pretrained VGG-16 convolutional neural network
`SeriesNetwork` object

Pretrained VGG-16 convolutional neural network returned as a `SeriesNetwork` object.

References

[1] *ImageNet*. http://www.image-net.org

[2] Russakovsky, O., Deng, J., Su, H., et al. “ImageNet Large Scale Visual Recognition Challenge.” *International Journal of Computer Vision (IJCV)*. Vol 115, Issue 3, 2015, pp. 211–252

[3] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[4] *Very Deep Convolutional Networks for Large-Scale Visual Recognition* http://www.robots.ox.ac.uk/~vgg/research/very_deep/

See Also

`alexnet` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `resnet50` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“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

`net = vgg19` returns a pretrained VGG-19 model. This model is trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [2]. VGG-19 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.

This function requires Neural Network 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 Neural Network Toolbox Model *for VGG-19 Network* support package.

Type `vgg19` at the command line.

```
vgg19
```

If Neural Network 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
```

```
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.

```
net = vgg19
```

```
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.

```
net.Layers
```

```
ans =

  47x1 Layer array with layers:

     1   'input'     Image Input             224x224x3 images with 'zerocenter' normali
     2   'conv1_1'   Convolution             64 3x3x3 convolutions with stride [1  1] a
     3   'relu1_1'   ReLU                    ReLU
```

```
 4   'conv1_2'   Convolution             64 3x3x64 convolutions with stride [1  1]
 5   'relu1_2'   ReLU                    ReLU
 6   'pool1'     Max Pooling             2x2 max pooling with stride [2  2] and pad
 7   'conv2_1'   Convolution             128 3x3x64 convolutions with stride [1  1]
 8   'relu2_1'   ReLU                    ReLU
 9   'conv2_2'   Convolution             128 3x3x128 convolutions with stride [1  1
10   'relu2_2'   ReLU                    ReLU
11   'pool2'     Max Pooling             2x2 max pooling with stride [2  2] and pad
12   'conv3_1'   Convolution             256 3x3x128 convolutions with stride [1  1
13   'relu3_1'   ReLU                    ReLU
14   'conv3_2'   Convolution             256 3x3x256 convolutions with stride [1  1
15   'relu3_2'   ReLU                    ReLU
16   'conv3_3'   Convolution             256 3x3x256 convolutions with stride [1  1
17   'relu3_3'   ReLU                    ReLU
18   'conv3_4'   Convolution             256 3x3x256 convolutions with stride [1  1
19   'relu3_4'   ReLU                    ReLU
20   'pool3'     Max Pooling             2x2 max pooling with stride [2  2] and pad
21   'conv4_1'   Convolution             512 3x3x256 convolutions with stride [1  1
22   'relu4_1'   ReLU                    ReLU
23   'conv4_2'   Convolution             512 3x3x512 convolutions with stride [1  1
24   'relu4_2'   ReLU                    ReLU
25   'conv4_3'   Convolution             512 3x3x512 convolutions with stride [1  1
26   'relu4_3'   ReLU                    ReLU
27   'conv4_4'   Convolution             512 3x3x512 convolutions with stride [1  1
28   'relu4_4'   ReLU                    ReLU
29   'pool4'     Max Pooling             2x2 max pooling with stride [2  2] and pad
30   'conv5_1'   Convolution             512 3x3x512 convolutions with stride [1  1
31   'relu5_1'   ReLU                    ReLU
32   'conv5_2'   Convolution             512 3x3x512 convolutions with stride [1  1
33   'relu5_2'   ReLU                    ReLU
34   'conv5_3'   Convolution             512 3x3x512 convolutions with stride [1  1
35   'relu5_3'   ReLU                    ReLU
36   'conv5_4'   Convolution             512 3x3x512 convolutions with stride [1  1
37   'relu5_4'   ReLU                    ReLU
38   'pool5'     Max Pooling             2x2 max pooling with stride [2  2] and pad
39   'fc6'       Fully Connected         4096 fully connected layer
40   'relu6'     ReLU                    ReLU
41   'drop6'     Dropout                 50% dropout
42   'fc7'       Fully Connected         4096 fully connected layer
43   'relu7'     ReLU                    ReLU
44   'drop7'     Dropout                 50% dropout
45   'fc8'       Fully Connected         1000 fully connected layer
46   'prob'      Softmax                 softmax
47   'output'    Classification Output   crossentropyex with 'tench', 'goldfish', a
```

To view the names of the classes learned by the network, you can view the `ClassNames` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

```
net.Layers(end).ClassNames(1:10)
```

```
ans =

  10×1 cell array

    'tench'
    'goldfish'
    'great white shark'
    'tiger shark'
    'hammerhead'
    'electric ray'
    'stingray'
    'cock'
    'hen'
    'ostrich'
```

- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”
- “Transfer Learning Using AlexNet”
- “Visualize Activations of a Convolutional Neural Network”

Output Arguments

net — Pretrained VGG-19 convolutional neural network
`SeriesNetwork` object

Pretrained VGG-19 convolutional neural network returned as a `SeriesNetwork` object.

References

[1] *ImageNet*. http://www.image-net.org

[2] Russakovsky, O., Deng, J., Su, H., et al. “ImageNet Large Scale Visual Recognition Challenge.” *International Journal of Computer Vision (IJCV)*. Vol 115, Issue 3, 2015, pp. 211–252

[3] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[4] *Very Deep Convolutional Networks for Large-Scale Visual Recognition* http://www.robots.ox.ac.uk/~vgg/research/very_deep/

See Also

`alexnet` | `deepDreamImage` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `resnet50` | `vgg16`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“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 network from Caffe [1] as a `Layer` array. The function returns the layers defined in the `.prototxt` given by the file name `protofile`.

This function requires Neural Network 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 Neural Network Toolbox Importer *for Caffe Models* support package.

Download the required support package by typing `importCaffeLayers` at the command line.

```
importCaffeLayers
```

If Neural Network 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.

```
protofile = 'digitsnet.prototxt';
```

Import the network layers.

```
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]
     3   'relu1'      ReLU                   ReLU
     4   'pool1'      Max Pooling            2x2 max pooling with stride [2  2] and pa
     5   'ip1'        Fully Connected        10 fully connected layer
     6   'loss'       Softmax                softmax
     7   'output'     Classification Output  crossentropyex with 'class1', 'class2', a
```

- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”

Input Arguments

`protofile` — File name

character vector

File name of the `.prototxt` file containing the network architecture, specified as a character vector. `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.

`importCaffeLayers` can import only the layers of a series network with the following Caffe layer types:

- Input
- Data
- Convolution
- ReLU
- Local Response Normalization (LRN)
- Pooling
- Inner Product
- Dropout
- Softmax With Loss
- Euclidean Loss

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.

Example: `'digitsnet.prototxt'`

Data Types: `char`

`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]`

Data Types: `double`

Output Arguments

`layers` — Network architecture
layer array

Network architecture, returned as a `Layer` array. Caffe networks that classify truecolor images expect a BGR image input. When imported into MATLAB, the image input layer expects an RGB image input

References

[1] *Caffe*. http://caffe.berkeleyvision.org/

[2] *Caffe Model Zoo*. http://caffe.berkeleyvision.org/model_zoo.html

See Also

`alexnet` | `googlenet` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”

Introduced in R2017a

importCaffeNetwork

Import pretrained convolutional neural network models from Caffe

Syntax

```
net = importCaffeNetwork(protofile,datafile)
net = importCaffeNetwork(protofile,datafile,'InputSize',sz)
net = importCaffeNetwork( ___ ,Name,Value)
```

Description

`net = importCaffeNetwork(protofile,datafile)` imports a pretrained network from Caffe [1] as a `SeriesNetwork` object. 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 Neural Network 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(protofile,datafile,'InputSize',sz)` returns a pretrained network and specifies the size of the input data. If the `.prototxt` file does not specify the size of the input data, you must specify the input size.

`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 Neural Network Toolbox Importer *for Caffe Models* support package.

To download the required support package, type `importCaffeNetwork` at the command line.

```
importCaffeNetwork
```

If Neural Network 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.

```
protofile = 'digitsnet.prototxt';
datafile = 'digits_iter_10000.caffemodel';
```

Import network.

```
net = importCaffeNetwork(protofile,datafile)
```

```
net =

  SeriesNetwork with properties:

    Layers: [7×1 nnet.cnn.layer.Layer]
```

- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”

Input Arguments

`protofile` — File name
character vector

File name of the `.prototxt` file containing the network architecture, specified as a character vector. `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.

`importCaffeNetwork` can import only a series network with the following Caffe layer types:

- Input
- Data
- Convolution
- ReLU
- Local Response Normalization (LRN)
- Pooling
- Inner Product
- Dropout
- Softmax With Loss
- Euclidean Loss

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.

The software cannot import a solver definition.

Example: `'digitsnet.prototxt'`

Data Types: `char`

`datafile` — File name
character vector

File name of the `.caffemodel` file containing the network weights, specified as a character vector. `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, see `importCaffeLayers`.

Example: `'digits_iter_10000.caffemodel'`

Data Types: `char`

`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]`

Data Types: `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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importCaffeNetwork(protofile,datafile,'AverageImage',I,'ClassNames',{'A','B','C'})` imports a pretrained network using the average image `I` for zero-center normalization and specifies the class names to be `'A'`, `'B'`, and `'C'`.

`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'`.

Data Types: `single`

`ClassNames` — Class names
cell array of character vectors

Class names associated with the output layer of the network, specified by a cell array of character vectors.

Example: `{'0','1','2','3','4','5','6','7','8','9'}`

Data Types: `cell`

Output Arguments

net — Imported pretrained Caffe network
series network object

Imported pretrained Caffe network, returned as a `SeriesNetwork` object. Caffe networks that classify truecolor images expect a BGR image input. When imported into MATLAB, the image input layer expects an RGB image input.

References

[1] *Caffe*. http://caffe.berkeleyvision.org/

[2] *Caffe Model Zoo*. http://caffe.berkeleyvision.org/model_zoo.html

See Also

`alexnet` | `googlenet` | `importCaffeLayers` | `importKerasLayers` | `importKerasNetwork` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”

Introduced in R2017a

augmentedImageDatastore

Generate batches of augmented image data

Description

An augmented image datastore generates batches of new images, with optional preprocessing such as resizing, rotation, and reflection, based on the training images. Augmenting image data helps prevent the network from overfitting and memorizing the exact details of the training images. It also increases the effective size of the training data set.

You can configure options for image preprocessing using an `imageDataAugmenter`. To train a network using augmented images, supply the `augmentedImageDatastore` to `trainNetwork`. For more information, see “Preprocess Images for Deep Learning”.

Creation

Syntax

```
auimds = augmentedImageDatastore(outputSize,imds)
auimds = augmentedImageDatastore(outputSize,X,Y)
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,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` — Images with labels

`ImageDatastore` object

Images with labels, specified as an `ImageDatastore` object with `categorical` labels. You can store data in `ImageDatastore` for image classification networks only.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, prefetching does not happen.

`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 `NaN`s, 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 `NaN`s.

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 `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 `NaN`s. If there are `NaN`s 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

Names of the response variables in the input table, specified as a character vector or cell array of character vectors. For problems with one response, `responseName` is the corresponding variable name in `tbl`. For regression problems with multiple response variables, `responseName` is a cell array of the corresponding variable names in `tbl`.

Data Types: `char` | `cell`

Properties

`ColorPreprocessing` — Preprocessing color operations

`'none'` (default) | `'gray2rgb'` | `'rgb2gray'`

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.

`MiniBatchSize` — Number of observations in each batch
positive integer

This property is read-only.

Number of observations that are returned in each batch. 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.

`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 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`

Object Functions

hasdata	Determine if data is available to read
partitionByIndex	Partition augmentedImageDatastore according to indices
preview	Subset of data in datastore
read	Read data from augmentedImageDatastore
readall	Read all data in datastore
readByIndex	Read data specified by index from augmentedImageDatastore
reset	Reset datastore to initial state
shuffle	Shuffle data in augmentedImageDatastore

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.

```
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])
```

```
imageAugmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [-20 20]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    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 'on the fly' and does not save any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.

```
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugme
```

Specify the convolutional neural network architecture.

```
layers = [
    imageInputLayer(imageSize)

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
```

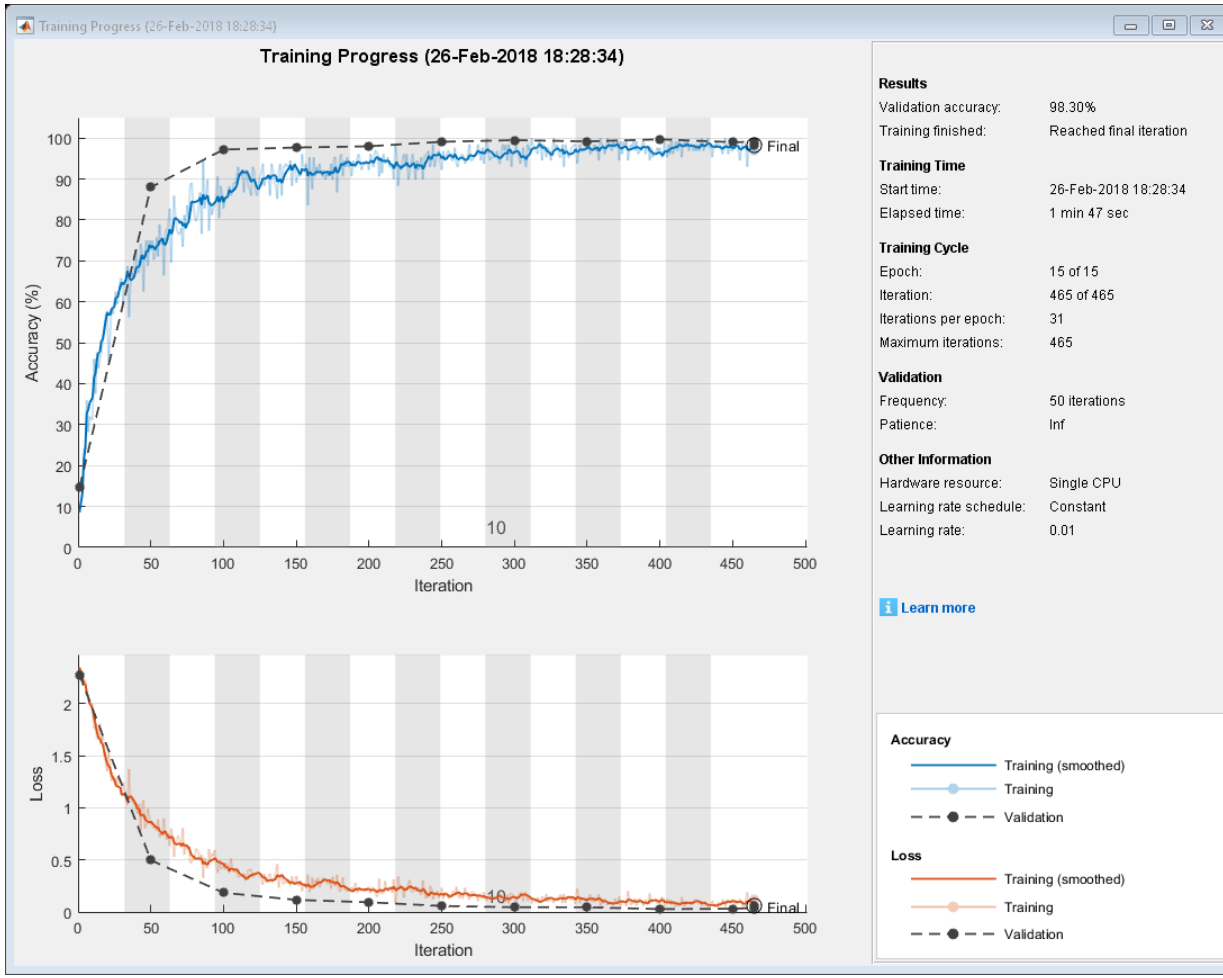
```
    reluLayer

    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},...
    'ValidationPatience',Inf);
```

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

```
net = trainNetwork(augimds,layers,opts);
```



See Also

`imageDataAugmenter` | `imageInputLayer` | `trainNetwork`

Topics

“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”

Introduced in R2017b

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.

Field Name	Description
`Filename`	`Filename` is a fully resolved path containing the path string, name of the file, and file extension.
`FileSize`	Total file size, in bytes. For MAT-files, `FileSize` is the total number of key-value pairs in the file.

See Also

`matlab.io.datastore.read` | `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 `length(ind)` number of rows.

`info` — Information about read data
structure array

Information about read data, returned as a structure array. The structure array can contain the following fields.

Field Name	Description
`Filename`	`Filename` is a fully resolved path containing the path string, name of the file, and file extension.
`FileSize`	Total file size, in bytes. For MAT-files, `FileSize` is the total number of key-value pairs in the file.

See Also

`partitionByIndex` | `read` | `readall`

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.

Syntax

```
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 Augmented Image Source

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. See the example "Train Network with Augmented Images" on page 1-754 to see the recommended workflow, which uses the `augmentedImageDatastore` function to create an augmented image datastore object.

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]
```

```
          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.

```
imageSize = [28 28 1];
auimds = augmentedImageSource(imageSize,XTrain,YTrain,'DataAugmentation',imageAugmenter
```

```
auimds =
  augmentedImageDatastore with properties:

           MiniBatchSize: 128
         NumObservations: 5000
       DataAugmentation: [1x1 imageDataAugmenter]
     ColorPreprocessing: 'none'
             OutputSize: [28 28]
         OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

Specify the convolutional neural network architecture.

```
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.
|=========================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate
|=========================================================================================
|       1 |           1 |       00:00:00 |       10.94% |       2.4515 |          0.001
|       2 |          50 |       00:00:13 |       51.56% |       1.5589 |          0.001
|       3 |         100 |       00:00:27 |       63.28% |       1.1184 |          0.001
|       4 |         150 |       00:00:41 |       75.00% |       0.8843 |          0.001
|       6 |         200 |       00:00:54 |       79.69% |       0.7065 |          0.001
|       7 |         250 |       00:01:08 |       85.16% |       0.5724 |          0.001
|       8 |         300 |       00:01:21 |       91.41% |       0.4801 |          0.001
|       9 |         350 |       00:01:35 |       85.94% |       0.4681 |          0.001
|      10 |         390 |       00:01:46 |       92.19% |       0.3722 |          0.001
|=========================================================================================
```

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 1-0` property of the returned augmented image datastore, `auimds`.

`imds` — Images with labels
`ImageDatastore` object

Images with labels, specified as an `ImageDatastore` object with `categorical` labels. You can store data in `ImageDatastore` for image classification networks only.

`ImageDatastore` allows batch reading of JPG or PNG image files using prefetching. If you use a custom function for reading the images, prefetching does not happen.

`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 `NaN`s, 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 `NaN`s.

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 `NaN`s. If there are `NaN`s 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

Names of the response variables in the input table, specified as a character vector or cell array of character vectors. For problems with one response, `responseName` is the corresponding variable name in `tbl`. For regression problems with multiple response variables, `responseName` is a cell array of the corresponding variable names in `tbl`.

Data Types: `char` | `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 single 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`.

Data Types: `char` | `string`

`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.

Note You cannot augment images in parallel and train a network in parallel at the same time. If you set `'BackgroundExecution'` to `true`, then you must set the `'ExecutionEnvironment'` of `trainingOptions` to `'auto'`, `'cpu'`, or `'gpu'`.

Output Arguments

`auimds` — Augmented image datastore
`augmentedImageDatastore` object

Augmented image datastore, returned as an `augmentedImageDatastore` object.

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”.

Creation

Syntax

```
aug = imageDataAugmenter
aug = imageDataAugmenter(Name,Value)
```

Description

`aug = imageDataAugmenter` creates an `imageDataAugmenter` object with default property values consistent with the identity transformation.

`aug = imageDataAugmenter(Name,Value)` configures a set of image augmentation options using name-value pairs to set properties on page 1-775. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`FillValue` — Fill value
`0` (default) | numeric scalar or vector

Fill value used to define out-of-bounds points when resampling, specified as a numeric scalar or vector.

- If the augmented images are single channel, then `FillValue` must be 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.

Example: `128`

`RandRotation` — Range of rotation
`[0 0]` (default) | 2-element numeric vector

Range of rotation applied to the input image, specified as a 2-element numeric vector. The second element must be larger than or equal to the first element.

Each image is rotated by an amount, in degrees, picked randomly from within the range. By default, augmented images are not rotated.

Example: `[0 360]`

`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.

`RandXScale` — Range of horizontal scale
`[1 1]` (default) | 2-element vector of positive numbers

Range of horizontal scale applied to the input image, specified as a 2-element vector of positive numbers. The second element must be larger than or equal to the first element.

Each image is scaled by a factor picked randomly from a uniform distribution within the range. By default, augmented images are not scaled in the horizontal direction.

Example: `[0.5 4]`

`RandYScale` — Range of vertical scale
`[1 1]` (default) | 2-element vector of positive numbers

Range of vertical scale applied to the input image, specified as a 2-element vector of positive numbers. The second element must be larger than or equal to the first element.

Each image is scaled by a factor picked randomly from a uniform distribution within the range. By default, augmented images are not scaled in the vertical direction.

Example: `[0.5 4]`

`RandXShear` — Range of horizontal shear
`[0 0]` (default) | 2-element numeric vector

Range of horizontal shear applied to the input image, specified as a 2-element numeric vector with values in the range (–90, 90). The second element must be larger than or equal to the first element. The elements in the vector represent shear angles, measured in units of degrees.

Each augmented image is sheared by a factor picked randomly from a uniform distribution within the range. 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

Range of vertical shear applied to the input image, specified as a 2-element numeric vector with values in the range (–90, 90). The second element must be larger than or equal to the first element. The elements in the vector represent shear angles, measured in units of degrees.

Each augmented image is sheared by a factor picked randomly from a uniform distribution within the range. 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

Range of horizontal translation applied to the input image, specified as a 2-element numeric vector. The second element must be larger than or equal to the first element.

Each augmented image is translated by a distance, in pixels, picked randomly from a uniform distribution within the range. By default, augmented images are not translated in the horizontal direction.

Example: `[0 10]`

`RandYTranslation` — Range of vertical translation
`[0 0]` (default) | 2-element numeric vector

Range of vertical translation applied to the input image, specified as a 2-element numeric vector. The second element must be larger than or equal to the first element.

Each augmented image is translated by a distance, in pixels, picked randomly from a uniform distribution within the range. By default, augmented images are not translated in the vertical direction.

Example: `[0 10]`

Examples

Create Image Data Augmenter to Resize and Reflect Images

Create an image augmenter that preprocesses images before training. This augmenter resizes and reflects images at random.

```
augmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXScale',[1 2], ...
    'RandYReflection',true, ...
    'RandYScale',[1 2])
```

```
augmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 1
     RandYReflection: 1
        RandRotation: [0 0]
```

```
          RandXScale: [1 2]
          RandYScale: [1 2]
          RandXShear: [0 0]
          RandYShear: [0 0]
    RandXTranslation: [0 0]
    RandYTranslation: [0 0]
```

Include the image augmenter in an augmented image datastore. The augmented image datastore also requires sample data, classifications, and an output image size.

```
[XTrain,YTrain] = digitTrain4DArrayData;
imageSize = [28 28 1];
datastore = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',augmente
```

```
datastore =
  augmentedImageDatastore with properties:

           MiniBatchSize: 128
         NumObservations: 5000
       DataAugmentation: [1x1 imageDataAugmenter]
     ColorPreprocessing: 'none'
             OutputSize: [28 28]
         OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

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])
```

```
imageAugmenter =
  imageDataAugmenter with properties:

           FillValue: 0
     RandXReflection: 0
     RandYReflection: 0
        RandRotation: [-20 20]
          RandXScale: [1 1]
          RandYScale: [1 1]
          RandXShear: [0 0]
          RandYShear: [0 0]
    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 'on the fly' and does not save any images to memory. `trainNetwork` updates the network parameters and then discards the augmented images.

```
imageSize = [28 28 1];
augimds = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugme
```

Specify the convolutional neural network architecture.

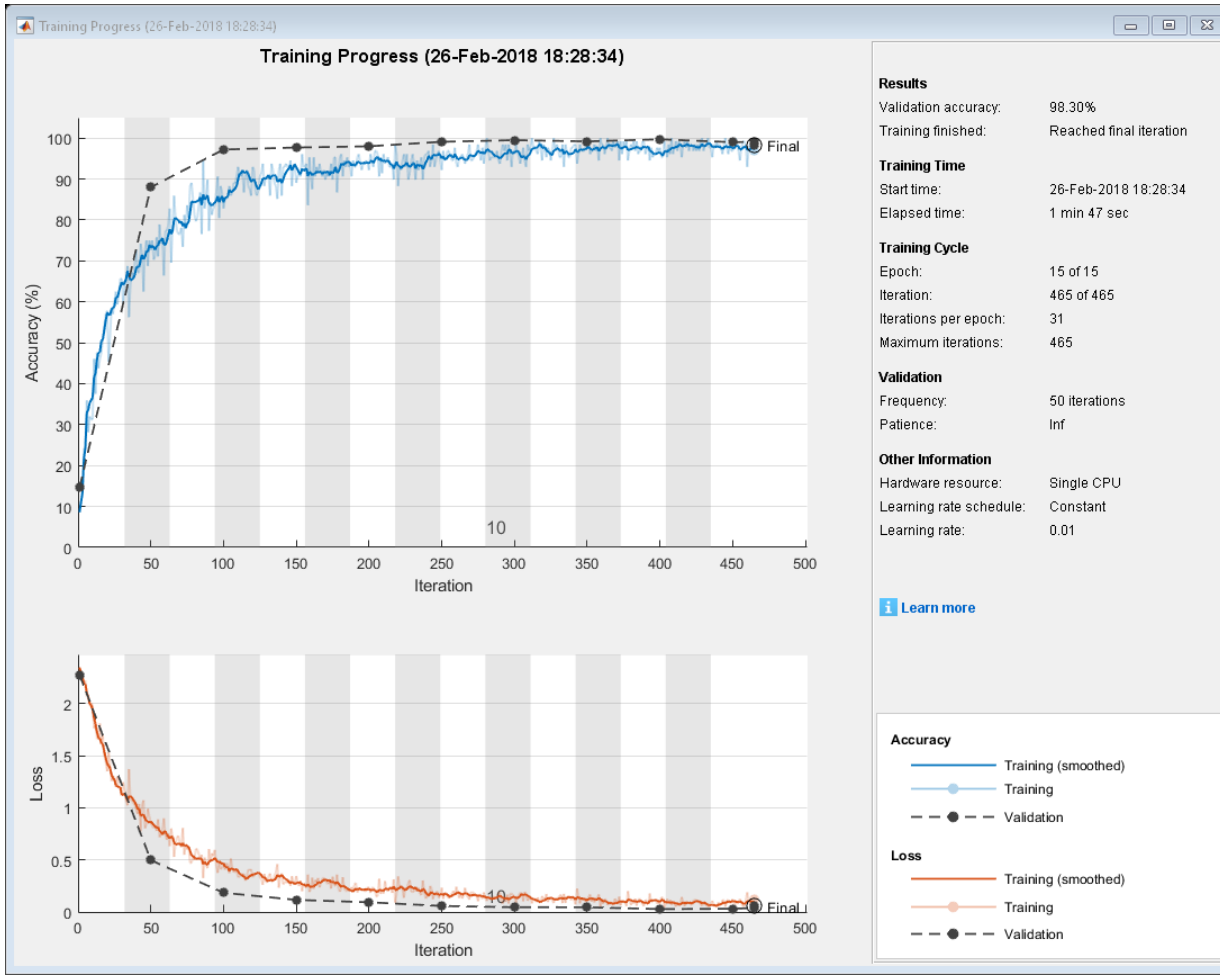
```
layers = [
    imageInputLayer(imageSize)

    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},...
    'ValidationPatience',Inf);
```

Train the network. Because the validation images are not augmented, the validation accuracy is higher than the training accuracy.

```
net = trainNetwork(augimds,layers,opts);
```



See Also

`augmentedImageDatastore` | `imageInputLayer` | `trainNetwork`

Topics

“Deep Learning in MATLAB”
“Preprocess Images for Deep Learning”

Introduced in R2017b

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

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` — Padding to add to input edges
`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Padding to add to input edges, specified as the comma-separated pair consisting of `'Padding'` and one of the following:

- `'same'` — The software 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 an odd amount of padding must be added vertically, then the software adds padding to the bottom. If an odd amount of padding must be added horizontally, 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

`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 of four nonnegative integers `[t b l r]`, 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 padding.

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'`.

If you specify a scalar or vector of nonnegative integers as the `'Padding'` value when creating a layer, then `PaddingMode` equals `'manual'`.

If you specify `'same'` as the `'Padding'` value when creating a layer, then `PaddingMode` equals `'same'`. The software 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 an odd amount of padding must be added vertically, then the software adds padding to the bottom. If an odd amount of padding must be added horizontally, 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 you create a layer, use the `'Padding'` name-value pair argument to specify padding.

Size of padding to apply to input borders vertically and horizontally, specified as a vector of two nonnegative integers `[a b]`, 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.

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Average Pooling Layer

Create an average pooling layer with the name `'avg1'`.

```
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 in a `Layer` array.

```
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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Average Pooling         2x2 average pooling with stride [1  1] and paddi
     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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Average Pooling         2x2 average pooling with stride [2  2] and paddi
     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: [3 2]
         Stride: [2 2]
    PaddingMode: 'manual'
    PaddingSize: [0 0 0 0]
```

This layer creates pooling regions of size [3 2] 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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Average Pooling         3x2 average pooling with stride [2  2] and paddi
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Average Pooling Layer

An average pooling layer outputs the average values of rectangular regions of its input. The `PoolSize` property determines the size of the rectangular regions. For example, if `PoolSize` is `[2,3]`, then the layer outputs the average value of regions of height 2 and width 3. The layer scans through the input horizontally and vertically in step sizes specified by the `Stride` property. If the pooling region dimensions `PoolSize` are less than or equal to the stride dimensions `Stride`, then the pooling regions do not overlap.

Pooling layers perform down-sampling operations. The output size of a pooling layer with input size *InputSize* has dimensions (*InputSize* – *PoolSize* + 2**PaddingSize*)/*Stride* + 1. This value must be an integer for the whole image to be fully covered. If the combination of these parameters 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.

See Also

`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”

Introduced in R2016a

ClassificationOutputLayer

Classification output layer

Description

A classification output layer holds the name of the loss function the software uses for training the network for multiclass classification, the size of the output, and the class labels.

Creation

Create a classification output layer using `classificationLayer`.

Properties

`OutputSize` — Size of the output
`'auto'` (default) | positive integer

This property is read-only.

Size of the output, specified as a positive integer. The software determines the size of the output during training. This value is the number of labels in the data. Before the training, the output size is set to `'auto'`. After training, you can view the output size by indexing into the `Layers` property of the `SeriesNetwork` object.

`LossFunction` — Loss function for training
`'crossentropyex'`

This property is read-only.

Loss function for training, specified as `'crossentropyex'`, which stands for "Cross Entropy Function for k Mutually Exclusive Classes" on page 1-794.

Data Types: `char`

`ClassNames` — Names of the classes
`{}` (default) | cell array

This property is read-only.

Names of the classes, specified as a cell array of class names determined during training. Before training, this property is an empty cell array.

Data Types: `cell`

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Classification Output Layer

Create a classification output layer with the name `'coutput'`.

```
layer = classificationLayer('Name','coutput')
```

```
layer =
  ClassificationOutputLayer with properties:

            Name: 'coutput'
      ClassNames: {1x0 cell}
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

The default loss function for classification is cross entropy for k mutually exclusive classes.

Include a classification output 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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Cross Entropy Function for *k* Mutually Exclusive Classes

For multiclass classification problems, the software assigns each input to one of the k mutually exclusive classes. The loss (error) function for this case is the cross entropy function for a 1-of-k coding scheme:

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\sum_{j=1}^{k} t_{ij} \ln y_j(\boldsymbol{x}_i, \boldsymbol{\theta}),$$

where $\boldsymbol{\theta}$ is the parameter vector, t_{ij} is the indicator that the ith sample belongs to the jth class, and $y_j(\boldsymbol{x}_i, \boldsymbol{\theta})$ is the output for sample i. The output $y_j(\boldsymbol{x}_i, \boldsymbol{\theta})$ can be interpreted as the probability that the network associates ith input with class j, that is, $P\left(t_j = 1 \middle| \boldsymbol{x}_i\right)$.

The output unit activation function is the softmax function:

$$y_r(\boldsymbol{x}) = \frac{\exp\left(a_r(\boldsymbol{x})\right)}{\sum_{j=1}^{k}\exp\left(a_j(\boldsymbol{x})\right)},$$

where $0 \le y_r \le 1$ and $\sum_{j=1}^{k} y_j = 1$.

References

[1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

See Also

`RegressionOutputLayer` | `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”

Introduced in R2016a

convolution2dLayer

2-D convolutional layer

Description

A 2-D convolutional layer applies sliding 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, NumChanels, WeightLearnRateFactor, BiasLearnRateFactor, WeightL2Factor, BiasL2Factor, 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

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`, `NumChannels`, `WeightLearnRateFactor`, `BiasLearnRateFactor`, `WeightL2Factor`, `BiasL2Factor`, 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 the size of the zero padding so that the layer output has the same size as the input.

`Padding` — Padding to add to input edges

`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Padding to add to input edges, specified as the comma-separated pair consisting of `'Padding'` and one of the following:

- `'same'` — The software 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 an odd amount of padding must be added vertically, then the software adds padding to the bottom. If an odd amount of padding must be added horizontally, 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

`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 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.

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 of four nonnegative integers `[t b l r]`, 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 padding.

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'`.

If you specify a scalar or vector of nonnegative integers as the `'Padding'` value when creating a layer, then `PaddingMode` equals `'manual'`.

If you specify `'same'` as the `'Padding'` value when creating a layer, then `PaddingMode` equals `'same'`. The software 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 an odd amount of padding must be added vertically, then the software adds padding to the bottom. If an odd amount of padding must be added horizontally, 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 you create a layer, use the `'Padding'` name-value pair argument to specify padding.

Size of padding to apply to input borders vertically and horizontally, specified as a vector of two nonnegative integers `[a b]`, 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.

`Weights` — Layer weights
4-D array

Layer weights for the convolutional layer, specified as a `FilterSize(1)`-by-`FilterSize(2)`-by-`NumChannels`-by-`NumFilters` array.

You cannot set this property using a name-value pair.

Data Types: `single` | `double`

`Bias` — Layer biases
3-D array

Layer biases for the convolutional layer, specified as a 1-by-1-by-`NumFilters` array.

You cannot set this property using a name-value pair.

Data Types: `single` | `double`

`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 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 is 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 is 16.

If `NumChannels` is `'auto'`, then the software infers the correct value for the number of channels during training time.

Example: 256

`WeightLearnRateFactor` — Learning rate factor for the 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 the layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights 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

`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 the 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) | scalar value

L2 regularization factor for the weights, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate 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.

Data Types: `double`

`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 learning rate for the biases in the layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in the layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

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:

           Name: ''

   Hyperparameters
     FilterSize: [11 11]
    NumChannels: 'auto'
     NumFilters: 96
         Stride: [4 4]
    PaddingMode: 'manual'
    PaddingSize: [0 0 0 0]

   Learnable Parameters
        Weights: []
           Bias: []

  Show all 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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Specify Initial Weights and Biases in Convolutional Layer

Create a convolutional layer with 32 filters, each with a height and width of 5. Pad the input image with 2 pixels along its border. Set the learning rate factor for the bias to 2. Manually initialize the weights from a Gaussian distribution with a standard deviation of 0.0001.

```
layer = convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2)
```

```
layer =
  Convolution2DLayer with properties:

           Name: ''

   Hyperparameters
     FilterSize: [5 5]
    NumChannels: 'auto'
     NumFilters: 32
         Stride: [1 1]
    PaddingMode: 'manual'
    PaddingSize: [2 2 2 2]

   Learnable Parameters
        Weights: []
           Bias: []

  Show all properties
```

Suppose the input has color images. Manually initialize the weights from a Gaussian distribution with standard deviation of 0.0001.

```
layer.Weights = randn([5 5 3 32]) * 0.0001;
```

The size of the local regions in the layer is 5-by-5. The number of color channels for each region is 3. The number of feature maps is 32 (the number of filters). Therefore, there are 5*5*3*32 weights in the layer.

`randn([5 5 3 32])` returns a 5-by-5-by-3-by-32 array of values from a Gaussian distribution with a mean of 0 and a standard deviation of 1. Multiplying the values by 0.0001 sets the standard deviation of the Gaussian distribution equal to 0.0001.

Similarly, initialize the biases from a Gaussian distribution with a mean of 1 and a standard deviation of 0.00001.

```
layer.Bias = randn([1 1 32])*0.00001 + 1;
```

There are 32 feature maps, and therefore 32 biases. `randn([1 1 32])` returns a 1-by-1-by-32 array of values from a Gaussian distribution with a mean of 0 and a standard deviation of 1. Multiplying the values by 0.00001 sets the standard deviation of values equal to 0.00001, and adding 1 sets the mean of the Gaussian distribution equal to 1.

Create Convolutional Layer That Fully Covers the Input

Suppose the size of the input is 28-by-28-1. Create a convolutional layer with 16 filters that have a height of 6 and a width of 4, that traverses the input with a stride of 4 both horizontally and vertically. 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 zero padding is required on the top and bottom of the image: (28 – 6+ 2*1)/4 + 1 = 7. For the vertical output dimension to be an integer, no zero padding is required: (28 – 4+ 2*0)/4 + 1 = 7. Construct the convolutional layer as follows:

```
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]
    PaddingMode: 'manual'
    PaddingSize: [1 1 0 0]

  Learnable Parameters
       Weights: []
          Bias: []

  Show all properties
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Convolutional Layer

A convolutional layer consists of neurons that connect to small regions of the input or the layer before it. These regions are called *filters*. You can specify the size of these regions using the `filterSize` input argument.

For each region, the software computes a dot product of the weights and the input, and then adds a bias term. The filter then moves along the input vertically and horizontally, repeating the same computation for each region, that is, convolving the input. The step size with which it moves is called the *stride*. You can specify this step size with the `Stride` property. These local regions that the neurons connect to might overlap depending on the filter size and the stride.

The number of weights used for a filter is $h*w*c$, where h is the height, and w is the width of the filter size, and c is the number of channels in the input (for example, if the input is a color image, then the number of channels is three). As a filter moves along the input, it uses the same set of weights and bias for the convolution, forming a feature map. The convolution layer usually has multiple feature maps, each with a different set of weights and a bias. The number of filters determines the number of feature maps.

The total number of parameters in a convolutional layer is $((h*w*c + 1)*\textit{Number of Filters})$, where 1 is for the bias.

The output height and width of the convolutional layer is (*Input Size* – *Filter Size* + 2**Padding*)/*Stride* + 1. This value must be an integer for the whole image to be fully covered. If the combination of these parameters does not lead the image to be fully covered, then the software by default ignores the remaining part of the image along the right and bottom edge in the convolution.

The total number of neurons in a feature map, say *Map Size*, is the product of the output height and width. The total number of neurons (output size) in a convolutional layer, then, is *Map Size*Number of Filters*.

For example, suppose that the input image is a 28-by-28-by-3 color image. For a convolutional layer with 16 filters, and a filter size of 8-by-8, the number of weights per filter is 8*8*3 = 192, and the total number of parameters in the layer is (192+1) * 16 = 3088. Assuming stride is 4 in each direction, the total number of neurons in each feature map is 6-by-6 ((28 – 8+0)/4 + 1 = 6). Then, the total number of neurons in the layer is 6*6*16 = 256. Usually, the results from these neurons pass through some form of nonlinearity, such as rectified linear units (ReLU).

References

[1] Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Handwritten Digit Recognition with a Back-Propagation Network." In *Advances in Neural Information Processing Systems*. 1990.

[2] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. ''Gradient-based Learning Applied to Document Recognition.'' *Proceedings of the IEEE*. Vol 86, pp. 2278–2324, 1998.

[3] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: The MIT Press, 2012.

See Also

`MaxPooling2dLayer` | `batchNormalizationLayer` | `fullyConnectedLayer` | `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”

Introduced in R2016a

transposedConv2dLayer

Create a transposed 2-D convolution layer

Syntax

```
tcoutputlayer = transposedConv2dLayer(filterSize,numFilters)
tcoutputlayer = transposedConv2dLayer(filterSize,numFilters,
Name,Value)
```

Description

`tcoutputlayer = transposedConv2dLayer(filterSize,numFilters)` returns a transposed 2-D convolution layer and sets the `filterSize` and `numFilters` properties.

`tcoutputlayer = transposedConv2dLayer(filterSize,numFilters, Name,Value)` returns a transposed 2-D convolutional layer and sets the optional `Stride`, `Cropping`, `NumChannels`, `WeightLearnRateFactor`, `BiasLearnRateFactor`, `WeightL2Factor`, `BiasL2Factor`, and `Name` properties.

Height and width of the output feature map:

- `stride`*(*input_size* - 1) + `filterSize` - 2 * *cropping*

Size of the layer's `Weights` matrix:

- [`filterSize` `numFilters` `numChannels`]

Size of the layer's `Bias` vector:

- [1 1 `numFilters`]

Examples

Create a Transposed Convolutional Layer With 96 Filters

Set the height and width of the layer to 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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Cropping',0`

`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.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`Cropping` — Output layer size reduction
`0` (default) | scalar | two-element vector

Output layer size reduction, specified as '`Cropping`' and a scalar or scalar vector. You can trim the edges of the full transposed convolution by the same amount or specify vertical and horizontal amounts. If you specify a vector, [*vertical*,*horizontal*], the vertical value trims the top and bottom, and the horizontal value trims the sides.

Data Types: `single` | `double`

`NumChannels` — Number of channels for each filter
`'auto'` (default) | integer

Number of channels for each filter, specified as '`NumChannels`' and a character vector `'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.

`WeightLearnRateFactor` — Learning rate factor for the 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 the layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights 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

`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 the 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) | scalar value

L2 regularization factor for the weights, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate 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.

Data Types: `double`

`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 learning rate for the biases in the layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in the layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Output Arguments

`tcoutputlayer` — Transposed 2-D convolution layer
`TransposedConvolution2DLayer` object

Transposed 2-D convolution layer, returned as a `TransposedConvolution2dLayer` object.

See Also

`SoftmaxLayer` | `TransposedConvolution2DLayer` | `averagePooling2dLayer` | `maxPooling2dLayer`

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

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

`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.

`NumChannels` — Number of channels for each filter

`'auto'` (default) | integer

Number of channels for each filter, specified as '`NumChannels`' and a character vector `'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.

`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 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.

Example: `[2 3]` specifies a vertical step size of 2 and a horizontal step size of 3.

`Cropping` — Output layer size reduction

`0` (default) | scalar | two-element vector

Output layer size reduction, specified as '`Cropping`' and a scalar or scalar vector. You can trim the edges of the full transposed convolution by the same amount or specify vertical and horizontal amounts. If you specify a vector, [*vertical*,*horizontal*], the vertical value trims the top and bottom, and the horizontal value trims the sides.

Data Types: `single` | `double`

`Weights` — Layer weights

4-D array

Layer weights for the convolutional layer, specified as a `FilterSize(1)`-by-`FilterSize(2)`-by-`NumChannels`-by-`NumFilters` array.

You cannot set this property using a name-value pair.

Data Types: `single` | `double`

`Bias` — Layer biases
3-D array

Layer biases for the convolutional layer, specified as a 1-by-1-by-`NumFilters` array.

You cannot set this property using a name-value pair.

Data Types: `single` | `double`

`WeightLearnRateFactor` — Learning rate factor for the 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 the layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights 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

`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 the 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) | scalar value

L2 regularization factor for the weights, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate 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.

Data Types: `double`

`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 learning rate for the biases in the layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in the layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

Examples

Create a Transposed Convolutional Layer With 96 Filters

Set the height and width of the layer to 11. Use a stride of 4 in the horizontal and vertical directions.

```
layer = transposedConv2dLayer(11,96,'Stride',4);
```

See Also

`averagePooling2dLayer` | `convolution2dLayer` | `maxPooling2dLayer` | `transposedConv2dLayer`

Introduced in R2017b

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

`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` — *α* hyperparameter in normalization
0.0001 (default) | numeric scalar

α hyperparameter in the normalization (the multiplier term), specified as a numeric scalar.

Example: `0.0002`

`Beta` — *β* hyperparameter in normalization
0.75 (default) | numeric scalar

β 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`

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

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.

```
layer = crossChannelNormalizationLayer(5,'K',1)
```

```
layer =
  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.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    crossChannelNormalizationLayer(3)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer]
```

```
layers =
  7x1 Layer array with layers:

     1   ''   Image Input                   28x28x1 images with 'zerocenter' normalizat
     2   ''   Convolution                   20 5x5 convolutions with stride [1  1] and
     3   ''   ReLU                          ReLU
     4   ''   Cross Channel Normalization   cross channel normalization with 3 channels
     5   ''   Fully Connected               10 fully connected layer
     6   ''   Softmax                       softmax
     7   ''   Classification Output         crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Local Response Normalization

For each element x in the input, the software computes a normalized value x' using

$$x' = \frac{x}{\left(K + \frac{\alpha * ss}{windowChannelSize}\right)^{\beta}},$$

where K, α, and β are the hyperparameters, and ss is the sum of squares of the elements in the normalization window. This formula is slightly different than the formula presented in [1]. You can obtain the equivalent formula by multiplying the `alpha` value by the `windowChannelSize`.

References

[1] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. Vol. 25, 2012.

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”

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

`Probability` — Probability to drop out input elements
0.5 (default) | numeric scalar in the range 0 to 1

Probability for dropping out input elements (neurons) during training time, specified as a numeric scalar in the range 0–1.

A higher number results in more neurons being dropped during training.

Example: `0.4`

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Dropout Layer

Create a dropout layer with name `'drop1'`.

```
layer = dropoutLayer('Name','drop1')
```

```
layer =
  DropoutLayer with properties:

           Name: 'drop1'

   Hyperparameters
    Probability: 0.5000
```

Include a dropout layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    dropoutLayer
    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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Dropout                 50% dropout
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Dropout Layer

A dropout layer randomly sets input elements to zero with a given probability.

This operation corresponds to temporarily dropping a randomly chosen unit and all its connections from the network during training. So, for each new input element, the software randomly selects a subset of neurons, forming a different network architecture. These architectures use common weights, but because the learning does not depend on specific neurons and connections, the dropout layer may help prevent overfitting [1], [2].

References

[1] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*. Vol. 15, pp. 1929-1958, 2014.

[2] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. Vol. 25, 2012.

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”

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 properties WeightLearnRateFactor, BiasLearnRateFactor, WeightL2Factor, BiasL2Factor, and Name 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

`WeightLearnRateFactor` — Learning rate factor for the 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 the layer. For example, if `WeightLearnRateFactor` is 2, then the learning rate for the weights 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

`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 the 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) | scalar value

L2 regularization factor for the weights, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate 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.

Data Types: `double`

`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 learning rate for the biases in the layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in the layer is twice the global L2 regularization factor. You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: 2

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

`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.

`OutputSize` — Output size
positive integer

Output size for the fully connected layer, specified as a positive integer.

Example: `10`

`Weights` — Layer weights
matrix

Layer weights, specified as an `OutputSize`-by-`InputSize` matrix.

Data Types: `single` | `double`

`Bias` — Layer biases
matrix

Layer biases, specified as an `OutputSize`-by-`1` matrix.

Data Types: `single` | `double`

Examples

Create Fully Connected Layer

Create a fully connected layer with an output size of 10 and the name `'fc1'`.

```
layer = fullyConnectedLayer(10,'Name','fc1')
```

```
layer =
  FullyConnectedLayer with properties:

         Name: 'fc1'

   Hyperparameters
     InputSize: 'auto'
    OutputSize: 10

   Learnable Parameters
       Weights: []
          Bias: []

  Show all properties
```

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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Specify Initial Weights and Biases in Fully Connected Layer

Create a fully connected layer with an output size of 10. Set the learning rate factor for the bias to 2. Manually initialize the weights from a Gaussian distribution with a standard deviation of 0.0001.

```
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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

To initialize the weights of the fully connected layer, you must know the layer's input size. The input size is equal to the output size of the preceding max pooling layer, which, in turn, depends on the output size of the convolutional layer.

For one direction in a channel (feature map) of the convolutional layer, the output is $((28-5+2\times 0)/1)+1$. The max pooling layer has nonoverlapping regions, so it down-samples by 2 in each direction, that is, $24/2=12$. For one channel of the convolutional layer, the output of the max pooling layer is $12\times 12=144$. There are 20 channels in the convolutional layer, so the output of the max pooling layer is $144\times 20=2880$. This is the size of the input to the fully connected layer.

The formula for overlapping regions gives the same result. For one direction of a channel, the output is $((24-2+0)/2)+1=12$. For one channel, the output is 144. For all 20 channels in the convolutional layer, the output of the max pooling layer is 2880.

Initialize the weights of the fully connected layer from a Gaussian distribution with a mean of 0 and a standard deviation of 0.0001.

```
layers(5).Weights = randn([10 2880]) * 0.0001;
```

`randn([10 2880])` returns a 10-by-2880 matrix of values from a Gaussian distribution with mean 0 and standard deviation 1. Multiplying the values by 0.0001 sets the standard deviation of the Gaussian distribution equal to 0.0001.

Initialize the biases from a Gaussian distribution with a mean of 1 and a standard deviation of 0.0001.

```
layers(5).Bias = randn([10 1])*0.0001 + 1;
```

The size of the bias vector is equal to the output size of the fully connected layer, which is 10. `randn([10 1])` returns a 10-by-1 vector of values from a Gaussian distribution with a mean of 0 and a standard deviation of 1. Multiplying the values by 0.00001 sets the standard deviation of values equal to 0.00001, and adding 1 sets the mean of the Gaussian distribution equal to 1.

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Fully Connected Layer

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 `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.

Algorithms

The default for the initial weights is a Gaussian distribution with mean 0 and standard deviation 0.01. The default for the initial bias is 0. For an example showing how to change the initialization for the weights and bias manually, see “Specify Initial Weights and Biases in Fully Connected Layer” on page 1-829.

See Also

`BatchNormalizationLayer` | `Convolution2dLayer` | `ReLULayer` | `RegressionOutputLayer` | `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”

Introduced in R2016a

imageInputLayer

Image input layer

Description

An image input layer inputs images to a network and applies data normalization.

Creation

Syntax

```
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 1-833 using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

`InputSize` — Size of the input
row vector of three integer values

Size of the input data, specified as a row vector of three integer values `[h w c]`, where `h` is the height, `w` is the width, and `c` is the number of channels. Set `c` to `1` for grayscale images, `3` for RGB images, or the number of channels in multispectral or hyperspectral images. If you set `InputSize` using the function `imageInputLayer`, you can specify `inputSize` as a row vector of two values `[h w]` to set the number of channels to 1.

Note If `c` is greater than `1`, then the input data is treated as a stack of 2-dimensional planes of data, not as a 3-dimensional volume. Each plane is processed independently.

Example: `[128 128 3]`

`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: `char` | `cell`

`Normalization` — Data transformation
`'zerocenter'` (default) | `'none'`

Data transformation to apply every time data is forward propagated through the input layer, specified as one of the following.

- `'zerocenter'` — The layer subtracts the mean image of the training set.
- `'none'` — No transformation.

Data Types: `char`

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

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.

```
inputlayer = imageInputLayer([28 28 3],'Name','input')
```

```
inputlayer =
  ImageInputLayer with properties:

                Name: 'input'
           InputSize: [28 28 3]

   Hyperparameters
    DataAugmentation: 'none'
       Normalization: 'zerocenter'
```

Include an image input 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 paddi
```

```
3   ''   ReLU                    ReLU
4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
5   ''   Fully Connected         10 fully connected layer
6   ''   Softmax                 softmax
7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

References

[1] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". *Advances in Neural Information Processing Systems*. Vol 25, 2012.

[2] Cireşan, D., U. Meier, J. Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

See Also

`Convolution2dLayer` | `FullyConnectedLayer` | `MaxPooling2dLayer` | `augmentedImageDatastore` | `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”

Introduced in R2016a

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',2)` creates a max 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

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',2)` creates a max pooling layer with pool size `[2 2]` and stride `[2 2]`.

`Padding` — Padding to add to input edges
`[0 0 0 0]` (default) | vector of nonnegative integers | `'same'`

Padding to add to input edges, specified as the comma-separated pair consisting of `'Padding'` and one of the following:

- `'same'` — The software 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 an odd amount of padding must be added vertically, then the software adds padding to the bottom. If an odd amount of padding must be added horizontally, 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

`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 of four nonnegative integers `[t b l r]`, 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 padding.

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'`.

If you specify a scalar or vector of nonnegative integers as the `'Padding'` value when creating a layer, then `PaddingMode` equals `'manual'`.

If you specify `'same'` as the `'Padding'` value when creating a layer, then `PaddingMode` equals `'same'`. The software 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 an odd amount of padding must be added vertically, then the software adds padding to the bottom. If an odd amount of padding must be added horizontally, 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 you create a layer, use the `'Padding'` name-value pair argument to specify padding.

Size of padding to apply to input borders vertically and horizontally, specified as a vector of two nonnegative integers `[a b]`, 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`.

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Max Pooling Layer with Nonoverlapping Pooling Regions

Create a max pooling layer with nonoverlapping pooling regions.

```
layer = maxPooling2dLayer(2,'Stride',2)
```

```
layer =
  MaxPooling2DLayer with properties:

                   Name: ''
    HasUnpoolingOutputs: 0

   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.

```
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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

Create Max Pooling Layer with Overlapping Pooling Regions

Create a max pooling layer with overlapping pooling regions.

```
layer = maxPooling2dLayer([3 2],'Stride',2)

layer =
  MaxPooling2DLayer with properties:

                  Name: ''
    HasUnpoolingOutputs: 0

   Hyperparameters
              PoolSize: [3 2]
                Stride: [2 2]
           PaddingMode: 'manual'
           PaddingSize: [0 0 0 0]
```

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.

```
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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             3x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Max Pooling Layer

A max pooling layer outputs the maximum values of rectangular regions of its input. The `PoolSize` property determines the size of the rectangular regions. For example, if `PoolSize` is `[2,3]`, then the layer outputs the maximum value of regions of height 2 and width 3. The layer scans through the input horizontally and vertically in step sizes specified by the `Stride` property. If the pooling region dimensions `PoolSize` are less than or equal to the stride dimensions `Stride`, then the pooling regions do not overlap.

Pooling layers perform down-sampling operations. The output size of a pooling layer with input size *InputSize* has dimensions (*InputSize* – *PoolSize* + 2**PaddingSize*)/*Stride* + 1. This value must be an integer for the whole image to be fully covered. If the combination of these parameters 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.

See Also

`AveragePooling2dLayer` | `Convolution2dLayer` | `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”

Introduced in R2016a

maxUnpooling2dLayer

Max unpooling layer

Description

A max unpooling layer unpools the output of a max pooling 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 unpooled 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`.

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

Layer name, specified as a character vector.

Data Types: char

Examples

Create Max Unpooling Layer

Create a max unpooling layer that unpools the output of a max pooling layer.

```
layer = maxUnpooling2dLayer
```

```
layer =
  MaxUnpooling2DLayer with properties:

    Name: ''
```

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
```

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.

```
lgraph = layerGraph(layers)
```

```
lgraph =
  LayerGraph with properties:

         Layers: [2×1 nnet.cnn.layer.Layer]
    Connections: [1×2 table]
```

Unpool the output of the max pooling layer, by connecting the max pooling layer outputs to the max unpooling layer inputs.

```
lgraph = connectLayers(lgraph,'mpool/indices','unpool/indices');
lgraph = connectLayers(lgraph,'mpool/size','unpool/size');
```

See Also

`MaxPooling2dLayer` | `connectLayers` | `disconnectLayers` | `layerGraph` | `trainNetwork`

Topics

“Deep Learning in MATLAB”
“Specify Layers of Convolutional Neural Network”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

RegressionOutputLayer

Regression output layer

Description

A regression output layer holds the name of the loss function the software uses for training the network for regression, and the response names.

Creation

Create a regression output layer using `regressionLayer`.

Properties

`LossFunction` — Loss function for training
`'mean-squared-error'`

Loss function the software uses for training, specified as `'mean-squared-error'`.

`ResponseNames` — Names of responses
`{}` (default) | cell array

Names of the responses, specified as a cell array. If you do not specify the response names, the software initially specifies the default value `{}`, and automatically assigns the response names at training time.

Data Types: `cell`

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Regression Output Layer

Create a regression output layer with the name `'routput'`.

```
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
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

See Also

`ClassificationOutputLayer` | `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”

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

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [(
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

References

[1] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807-814. 2010.

See Also

`BatchNormalizationLayer` | `ClippedReLULayer` | `DropoutLayer` | `ImageInputLayer` | `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”

Introduced in R2016a

leakyReluLayer

Leaky Rectified Linear Unit (ReLU) layer

Description

A leaky ReLU layer performs a simple threshold operation, where any input value less than zero is multiplied by a fixed scalar.

This operation is equivalent to:

$$f(x) = \begin{cases} x, & x \geq 0 \\ scale * x, & x < 0 \end{cases}.$$

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

`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`

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

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.

```
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.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    leakyReluLayer
```

```
    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 paddi
     3   ''   Leaky ReLU              Leaky ReLU with scale 0.01
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

References

[1] Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." In *Proc. ICML*, vol. 30, no. 1. 2013.

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”

Introduced in R2017b

clippedReluLayer

Clipped Rectified Linear Unit (ReLU) layer

Description

A clipped ReLU layer performs a simple 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:

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x < ceiling \\ ceiling, & x \ge ceiling \end{cases}.$$

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

`Ceiling` — Ceiling for input clipping

positive scalar

Ceiling for input clipping, specified as a positive scalar.

Example: `10`

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Clipped ReLU Layer

Create a clipped ReLU layer with the name `'clip1'` and the clipping ceiling equal to 10.

```
layer = clippedReluLayer(10,'Name','clip1')
```

```
layer =
  ClippedReLULayer with properties:

       Name: 'clip1'

   Hyperparameters
    Ceiling: 10
```

Include a clipped ReLU layer in a `Layer` array.

```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    clippedReluLayer(10)
    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 paddi
     3   ''   Clipped ReLU            Clipped ReLU with ceiling 10
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

References

[1] Hannun, Awni, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, et al. "Deep speech: Scaling up end-to-end speech recognition." Preprint, submitted 17 Dec 2014. http://arxiv.org/abs/1412.5567

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”

Introduced in R2017b

batchNormalizationLayer

Batch normalization layer

Description

A batch normalization layer normalizes each input channel across a mini-batch. 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 β and scales it by a learnable scale factor γ. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up training of convolutional neural networks and reduce the sensitivity to network initialization.

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 properties Name, Epsilon, Offset, OffsetLearnRateFactor, OffsetL2Factor, “Scale” on page 1-0 , ScaleLearnRateFactor, and ScaleL2Factor using name-value pairs. For example, `batchNormalizationLayer('Name','BN1')` creates a batch normalization layer with the name `'BN1'`. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

General Layer Properties

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

`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.

`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.

Properties for Learnable Parameters

`Offset` — Channel offsets
numeric array

Channel offsets β, specified as a numeric array of size 1-by-1-by-`NumChannels`.

The offsets are learnable parameters. When training a network, `trainNetwork` uses the `Offset` property as the initial value. If you do not specify an `Offset` value when creating a layer, then `trainNetwork` uses an array of all zeros as the initial value.

`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.

`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.

`Scale` — Channel scale factors

numeric array

Channel scale factors γ, specified as a numeric array of size 1-by-1-by-`NumChannels` with positive elements.

The channel scale factors are learnable parameters. When training a network, `trainNetwork` uses the `Scale` property as the initial value. If you do not specify a `Scale` value when creating a layer, then `trainNetwork` uses an array of all ones as the initial value.

`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.

`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.

Properties Determined During Training

`TrainedMean` — Input mean

numeric array

Input mean of each channel, specified as a numeric array of size 1-by-1-by-`NumChannels` and determined during network training.

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` and determined during network training.

The layer uses `TrainedVariance` (in place of the mini-batch variance) to normalize the input during prediction.

Examples

Create Batch Normalization Layer

Create a batch normalization layer with the name `'BN1'`.

```
layer = batchNormalizationLayer('Name','BN1')
```

```
layer =
  BatchNormalizationLayer with properties:
```

```
              Name: 'BN1'
       NumChannels: 'auto'
       TrainedMean: []
   TrainedVariance: []

  Hyperparameters
           Epsilon: 1.0000e-05

  Learnable Parameters
            Offset: []
             Scale: []

 Show all properties
```

Include batch normalization layers in a `Layer` array.

```
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 paddi
     3   ''   Batch Normalization     Batch normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     6   ''   Convolution             32 3x3 convolutions with stride [1  1] and paddi
```

```
 7   ''   Batch Normalization     Batch normalization
 8   ''   ReLU                    ReLU
 9   ''   Fully Connected         10 fully connected layer
10   ''   Softmax                 softmax
11   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Algorithms

A batch normalization normalizes its inputs x_i by first calculating the mean μ_B and variance σ_B^2 over a mini-batch and over each input channel. Then, it calculates the normalized activations as

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + }}.$$

Here, ϵ (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

$$y_i = \gamma \hat{x}_i + \beta.$$

Here, the offset β and scale factor γ (`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 `TrainedMean` and `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

[1] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *preprint, arXiv:1502.03167* (2015).

See Also

`Convolution2dLayer` | `FullyConnectedLayer` | `ReLULayer` | `trainNetwork` | `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”

Introduced in R2017b

softmaxLayer

Softmax layer

Description

A softmax layer applies a softmax function to the input.

Creation

Syntax

```
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

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Softmax Layer

Create a softmax layer with the name 'sm1'.

```
layer = softmaxLayer('Name','sm1')
```

```
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 paddi
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0
     5   ''   Fully Connected         10 fully connected layer
     6   ''   Softmax                 softmax
     7   ''   Classification Output   crossentropyex
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

Definitions

Softmax Function

For a classification problem with more than two classes, the output unit activation function is the softmax function:

$$P\left(c_r|\boldsymbol{x},\boldsymbol{\theta}\right)=\frac{P\left(\boldsymbol{x},\boldsymbol{\theta}|c_r\right)P\left(c_r\right)}{\sum_{j=1}^{k}P\left(\boldsymbol{x},\boldsymbol{\theta}|c_j\right)P\left(c_j\right)}=\frac{\exp\left(a_r\left(\boldsymbol{x},\boldsymbol{\theta}\right)\right)}{\sum_{j=1}^{k}\exp\left(a_j\left(\boldsymbol{x},\boldsymbol{\theta}\right)\right)},$$

where $0\le P\left(c_r|\boldsymbol{x},\boldsymbol{\theta}\right)\le 1$ and $\sum_{j=1}^{k}P\left(c_j\left|\boldsymbol{x},\boldsymbol{\theta}\right.\right)=1$. Also, $a_r=\ln\left(P\left(\boldsymbol{x},\boldsymbol{\theta}|c_r\right)P\left(c_r\right)\right)$, $P\left(\boldsymbol{x},\boldsymbol{\theta}|c_r\right)$ is the conditional probability of the sample given class *r*, and $P\left(c_r\right)$ 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

[1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

See Also

`ClassificationOutputLayer` | `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”

Introduced in R2016a

bilstmLayer

Bidirectional long short-term memory (BiLSTM) layer

Description

A bidirectional LSTM (BiLSTM) layer is a recurrent neural network (RNN) layer. The layer learns bidirectional long-term dependencies between time steps. These dependencies can be useful for 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 BiLSTM Parameters on page 1-872, Learn Rate and L2 Factors on page 1-872 using one or more name-value pair arguments. You can specify multiple name-value pair arguments. Enclose each property name in quotes.

Properties

BiLSTM Parameters

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

`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

`NumHiddenUnits` — Number of hidden units

positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

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.

Learn Rate and L2 Factors

`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 the 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. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (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]`

`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 learning rate for the biases in the layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in the 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 Layer input (Forward)
4 Output gate (Forward)
5 Input gate (Backward)
6 Forget gate (Backward)
7 Layer input (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]`

`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 Layer input (Forward)
4 Output gate (Forward)
5 Input gate (Backward)
6 Forget gate (Backward)
7 Layer input (Backward)
8 Output gate (Backward)

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: `0.1`

`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. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (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]`

`RecurrentWeightsLearnRateFactor` — Learning rate factor for recurrent weights

1 (default) | numeric scalar | 1-by-8 numeric vector

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. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (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. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (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]`

State Parameters

`CellState` — Initial value of cell state

numeric vector

Initial value of the cell state, specified as a `2*NumHiddenUnits`-by-1 numeric vector.

`HiddenState` — Initial value of hidden state

numeric vector

Initial value of the hidden state, specified as a `2*NumHiddenUnits`-by-1 numeric vector.

Weights

`Bias` — Layer biases

numeric vector

Layer biases, specified as an `8*NumHiddenUnits`-by-1 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. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (Backward)
8. Output gate (Backward)

`InputWeights` — Input weights

matrix

Input weights, specified as an `8*NumHiddenUnits`-by-`InputSize` 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. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (Backward)
8. Output gate (Backward)

`RecurrentWeights` — Recurrent weights

matrix

Recurrent weights, specified as an `8*NumHiddenUnits`-by-`NumHiddenUnits` 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:

1. Input gate (Forward)
2. Forget gate (Forward)
3. Layer input (Forward)
4. Output gate (Forward)
5. Input gate (Backward)
6. Forget gate (Backward)
7. Layer input (Backward)
8. Output gate (Backward)

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'

   Learnable Parameters
       InputWeights: []
    RecurrentWeights: []
                Bias: []

   State Parameters
        HiddenState: []
```

```
           CellState: []

  Show all properties
```

Include a bidirectional LSTM layer in a `Layer` array.

```
layers = [ ...
    sequenceInputLayer(12)
    bilstmLayer(100)
    fullyConnectedLayer(9)
    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
```

- “Sequence Classification Using Deep Learning”
- “Long Short-Term Memory Networks”
- “Specify Layers of Convolutional Neural Network”
- “Set Up Parameters and Train Convolutional Neural Network”
- “Deep Learning in MATLAB”

See Also

`classifyAndUpdateState` | `lstmLayer` | `predictAndUpdateState` | `resetState` | `sequenceInputLayer`

Topics

“Sequence Classification Using Deep Learning”
“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 R2018a

lstmLayer

Long short-term memory (LSTM) layer

Description

An LSTM layer is a recurrent neural network (RNN) layer that enables support for time series and sequence data in a network. The layer performs additive interactions, which can help improve gradient flow over long sequences during training. LSTM layers are best suited for learning *long-term dependencies* (dependencies from distant time steps).

Creation

Syntax

```
layer = lstmLayer(numHiddenUnits)
layer = lstmLayer(numHiddenUnits,Name,Value)
```

Description

`layer = lstmLayer(numHiddenUnits)` creates an LSTM layer and sets the `NumHiddenUnits` property.

`layer = lstmLayer(numHiddenUnits,Name,Value)` sets additional "LSTM Parameters" on page 1-883 properties as well as Learn Rate and L2 Factors on page 1-883 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 Parameters

`Name` — Layer name

`''` (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

`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

`NumHiddenUnits` — Number of hidden units

positive integer

Number of hidden units (also known as the hidden size), specified as a positive integer.

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.

Learn Rate and L2 Factors

`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 the 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. Layer input
4. 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 learning rate for the biases in the layer. For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in the 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. Layer input
4. Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

`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. Layer input
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. Layer input
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. Layer input

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 Layer input
4 Output gate

To specify the same value for all the matrices, specify a nonnegative scalar.

Example: 2

Example: `[1 2 1 1]`

State Parameters

`CellState` — Initial value of cell state

numeric vector

Initial value of the cell state, specified as a `NumHiddenUnits`-by-1 numeric vector.

`OutputState` — Initial value of the output state

numeric vector

Initial value of the output state, specified as a `NumHiddenUnits`-by-1 numeric vector.

Weights

`Bias` — Layer biases

numeric vector

Layer biases for the LSTM layer, specified as a `4*NumHiddenUnits`-by-1 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. Layer input
4. Output gate

`InputWeights` — Input weights

matrix

Input weights, specified as a `4*NumHiddenUnits`-by-`InputSize` 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. Layer input
4. Output gate

`RecurrentWeights` — Recurrent weights

matrix

Recurrent weights, specified as a `4*NumHiddenUnits`-by-`NumHiddenUnits` 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. Layer input
4. Output gate

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
          InputSize: 'auto'
      NumHiddenUnits: 100
          OutputMode: 'sequence'

   Learnable Parameters
       InputWeights: []
    RecurrentWeights: []
                Bias: []

   State Parameters
         HiddenState: []
           CellState: []

  Show all properties
```

Include an LSTM layer in a `Layer` array.

```
layers = [ ...
    sequenceInputLayer(12)
    lstmLayer(100)
    fullyConnectedLayer(9)
    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
```

Train Network for Sequence Classification

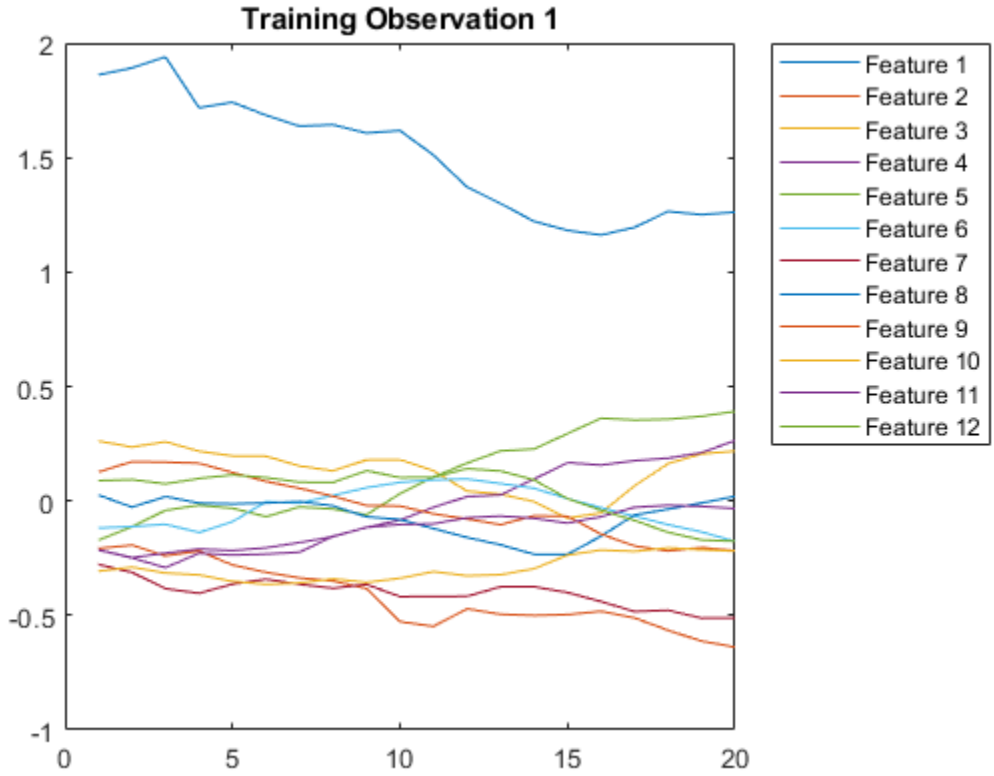
Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with feature dimension 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.

```
figure
plot(XTrain{1}')
title("Training Observation 1")
legend("Feature " + string(1:12),'Location','northeastoutside')
```



Define the LSTM network architecture. Specify the input size 12 (the dimension of the input data). Specify an LSTM layer to have 100 hidden units and output the last element of the sequence. Finally, specify 9 classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```
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
```

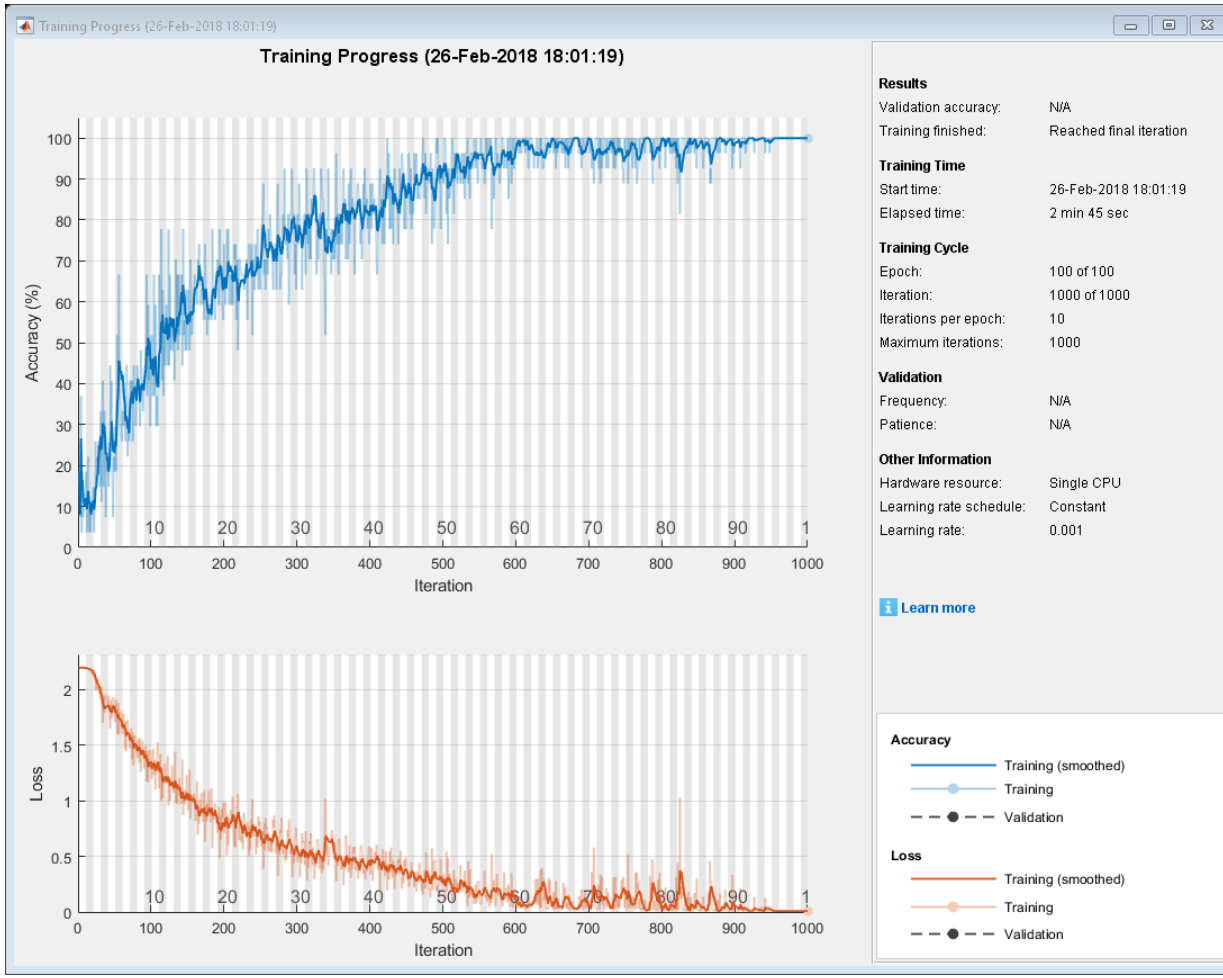
Specify the training options. Specify the solver to be `'adam'` and `'GradientThreshold'` to be 1. Set the mini-batch size to 27, and set the maximum number of epochs to 100.

Because the mini-batches are small with short sequences, training is better suited for the CPU. Specify `'ExecutionEnvironment'` to be `'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',0, ...
    '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. Set the mini-batch size to 27.

```
miniBatchSize = 27;
YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9270
```

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.

Specify the size of the sequence input layer to be the feature dimension of the input data. Specify the size of the fully connected layer to be the number of classes.

For the LSTM layer, choose an output size, and specify the output mode `'last'`.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    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 regression, use the same architecture for sequence-to-labe classification, but set the output mode of the LSTM layer to 'sequence'.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    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.

Specify the size of the sequence input layer to be the feature dimension of the input data. Specify the size of the fully connected layer to be the number of responses.

For the LSTM layer, choose an output size, and specify the output mode `'last'`.

```
inputSize = 12;
outputSize = 125;
numResponses = 1;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(outputSize,'OutputMode','last')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

To create an LSTM network for sequence-to-sequence regression, use the same architecture for sequence-to-one regression, but set the output mode of the LSTM layer to `'sequence'`.

```
inputSize = 12;
outputSize = 125;
numResponses = 1;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(outputSize,'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.

For sequence-to-label classification networks, the output mode of the last LSTM layer must be `'last'`.

```
inputSize = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    lstmLayer(numHiddenUnits2,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For sequence-to-sequence classification networks, the output mode of the last LSTM layer must be `'sequence'`.

```
inputSize = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

- “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”
- “Long Short-Term Memory Networks”
- “Deep Learning in MATLAB”

Definitions

Long Short-Term Memory Layer

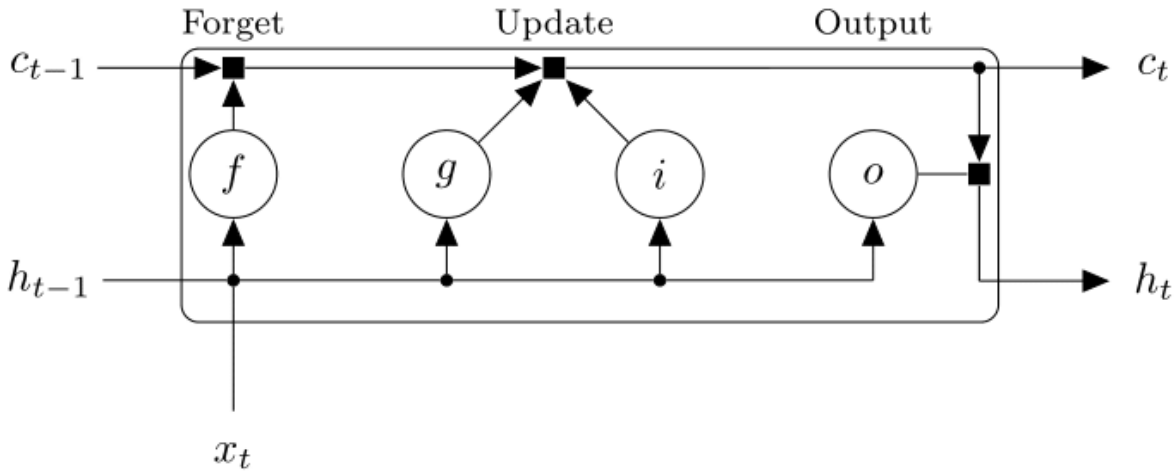
An LSTM layer is a recurrent neural network (RNN) layer that enables support for time series and sequence data in a network. The layer performs additive interactions, which can help improve gradient flow over long sequences during training. LSTM layers are best suited for learning *long-term dependencies* (dependencies from distant time steps).

The state of the layer consists of the *output state* (also known as the *hidden state*) and the *cell state*. The output 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, where the layer controls these updates using *gates*.

This table summarizes the components that control the cell state and output state of the layer.

Component	Purpose
Input gate (i)	Control level of cell state update
Forget gate (f)	Control level of cell state reset (forget)
Layer input (g)	Add information to cell state
Output gate (o)	Control level of cell state added to output state

This diagram illustrates the flow of data at time step t. The diagram highlights how the gates forget, update, and output the cell and output 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, layer input, 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 output (hidden) state at time step t is given by

$$h_t = o_t \odot \tanh(c_t).$$

This table shows the formula for each component at time step t.

Component	Formula
Input gate	$i_t = \sigma(W_i x_t + \mathrm{R}_i h_{t-1} + b_i)$
Forget gate	$f_t = \sigma(W_f x_t + \mathrm{R}_f h_{t-1} + b_f)$
Layer input	$g_t = \tanh(W_g x_t + \mathrm{R}_g h_{t-1} + b_g)$
Output gate	$o_t = \sigma(W_o x_t + \mathrm{R}_o h_{t-1} + b_o)$

Here, σ denotes the sigmoid function given by $\sigma(x) = (1+e^{-x})^{-1}$.

References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

[3] Hochreiter, S, and J. Schmidhuber, 1997. Long short-term memory. *Neural computation*, 9(8), pp.1735–1780.

See Also

`bilstmLayer` | `classifyAndUpdateState` | `predictAndUpdateState` | `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”
“Long Short-Term Memory Networks”
“Deep Learning in MATLAB”

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',Name)
```

Description

`layer = sequenceInputLayer(inputSize)` creates a sequence input layer and sets the InputSize property.

`layer = sequenceInputLayer(inputSize,'Name',Name)` creates a sequence input layer and sets the optional Name property.

Properties

`InputSize` — Size of input
positive integer

Size of the input, specified as a positive integer. `inputSize` is the number of dimensions of the input sequence at each time step.

Example: 100

Name — Layer name
'' (default) | character vector

Layer name, specified as a character vector. If `Name` is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

Examples

Create Sequence Input Layer

Create a sequence input layer with the name `'seq1'` and an input size of 12.

```
layer = sequenceInputLayer(12,'Name','seq1')
```

```
layer =
  SequenceInputLayer with properties:

         Name: 'seq1'
    InputSize: 12
```

Include an LSTM layer in a `Layer` array.

```
layers = [ ...
    sequenceInputLayer(12)
    lstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(9)
    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
```

Train Network for Sequence Classification

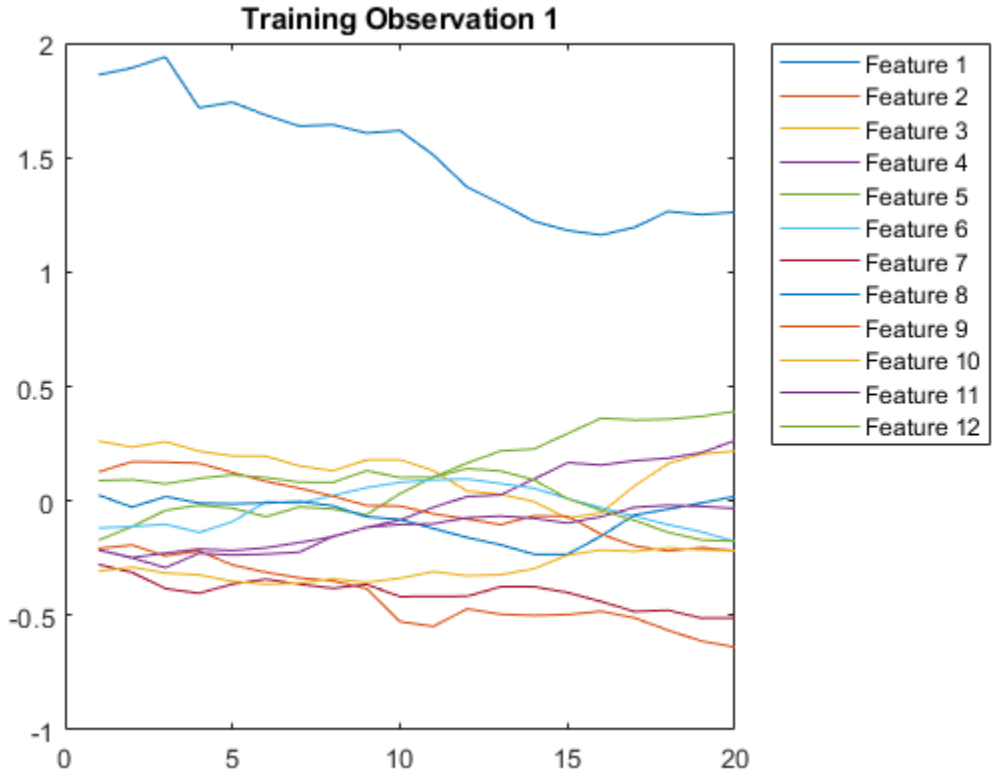
Train a deep learning LSTM network for sequence-to-label classification.

Load the Japanese Vowels data set as described in [1] and [2]. `XTrain` is a cell array containing 270 sequences of varying length with feature dimension 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.

```
figure
plot(XTrain{1}')
title("Training Observation 1")
legend("Feature " + string(1:12),'Location','northeastoutside')
```



Define the LSTM network architecture. Specify the input size 12 (the dimension of the input data). Specify an LSTM layer to have 100 hidden units and output the last element of the sequence. Finally, specify 9 classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```
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
```

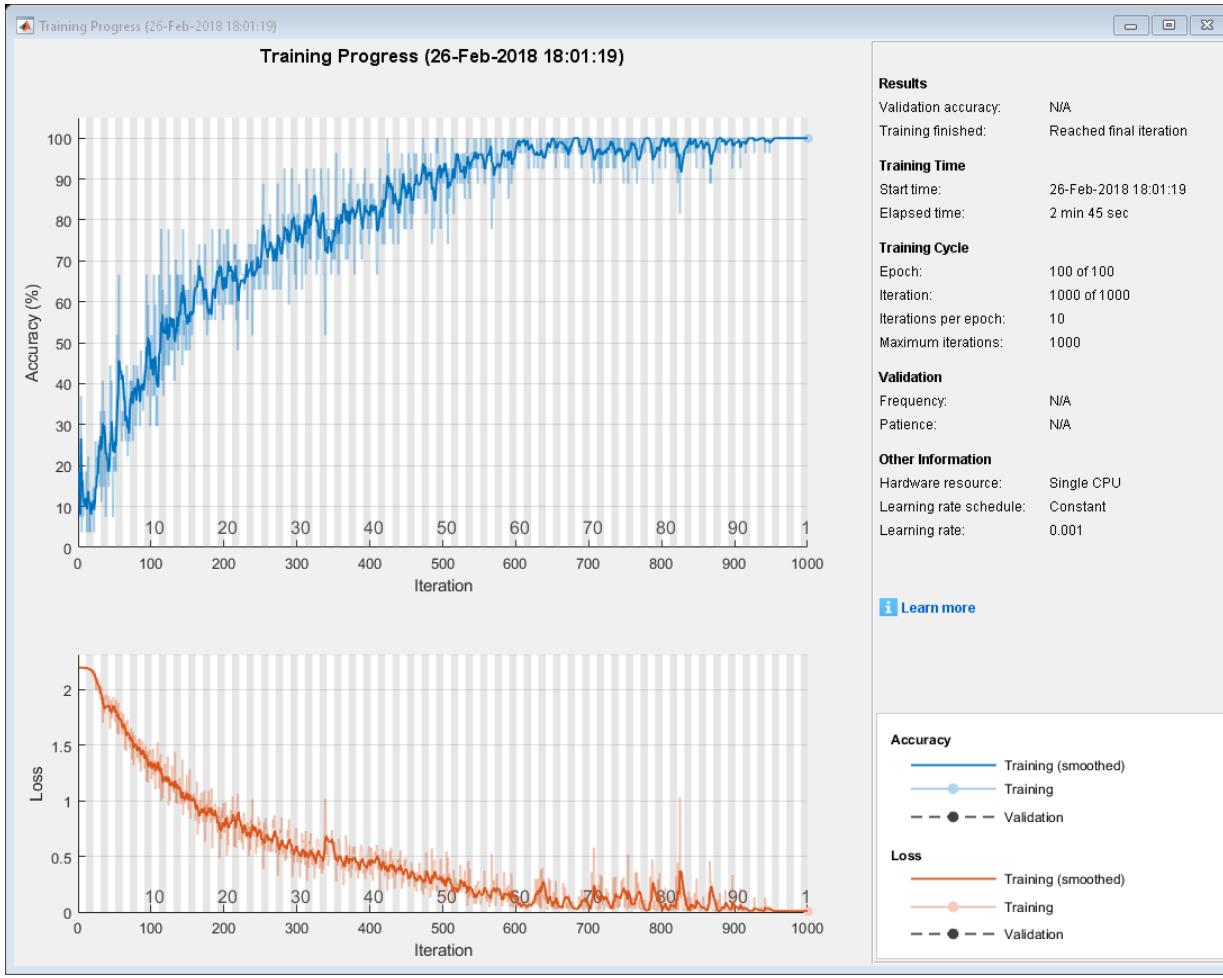
Specify the training options. Specify the solver to be `'adam'` and `'GradientThreshold'` to be 1. Set the mini-batch size to 27, and set the maximum number of epochs to 100.

Because the mini-batches are small with short sequences, training is better suited for the CPU. Specify `'ExecutionEnvironment'` to be `'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',0, ...
    '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. Set the mini-batch size to 27.

```
miniBatchSize = 27;
YPred = classify(net,XTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9270
```

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.

Specify the size of the sequence input layer to be the feature dimension of the input data. Specify the size of the fully connected layer to be the number of classes.

For the LSTM layer, choose an output size, and specify the output mode `'last'`.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    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 regression, use the same architecture for sequence-to-labe classification, but set the output mode of the LSTM layer to 'sequence'.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    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.

Specify the size of the sequence input layer to be the feature dimension of the input data. Specify the size of the fully connected layer to be the number of responses.

For the LSTM layer, choose an output size, and specify the output mode `'last'`.

```
inputSize = 12;
outputSize = 125;
numResponses = 1;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(outputSize,'OutputMode','last')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

To create an LSTM network for sequence-to-sequence regression, use the same architecture for sequence-to-one regression, but set the output mode of the LSTM layer to `'sequence'`.

```
inputSize = 12;
outputSize = 125;
numResponses = 1;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(outputSize,'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.

For sequence-to-label classification networks, the output mode of the last LSTM layer must be `'last'`.

```
inputSize = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    lstmLayer(numHiddenUnits2,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For sequence-to-sequence classification networks, the output mode of the last LSTM layer must be `'sequence'`.

```
inputSize = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

- “Sequence Classification Using Deep Learning”
- “Time Series Forecasting Using Deep Learning”
- “Sequence-to-Sequence Classification Using Deep Learning”
- “Long Short-Term Memory Networks”
- “Specify Layers of Convolutional Neural Network”
- “Set Up Parameters and Train Convolutional Neural Network”
- “Deep Learning in MATLAB”

References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`bilstmLayer` | `classifyAndUpdateState` | `lstmLayer` | `predictAndUpdateState` | `resetState`

Topics

“Sequence Classification Using Deep Learning”
“Time Series Forecasting Using Deep Learning”
“Sequence-to-Sequence Classification Using Deep Learning”
“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

Layer

Network layer for deep learning

Description

Layers that define the architecture of neural networks for deep learning.

Creation

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. Use the following functions to create different layer types.

Input Layers

Function	Description
`imageInputLayer`	An image input layer inputs images to a network and applies data normalization.
`sequenceInputLayer`	A sequence input layer inputs sequence data to a network.

Learnable Layers

Function	Description
`convolution2dLayer`	A 2-D convolutional layer applies sliding 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.
`transposedConv2dLayer`	A transposed 2-D convolution layer upsamples feature maps.
`fullyConnectedLayer`	A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.
`lstmLayer`	An LSTM layer is a recurrent neural network (RNN) layer that enables support for time series and sequence data in a network. The layer performs additive interactions, which can help improve gradient flow over long sequences during training. LSTM layers are best suited for learning *long-term dependencies* (dependencies from distant time steps).
`bilstmLayer`	A bidirectional LSTM (BiLSTM) layer is a recurrent neural network (RNN) layer. The layer learns bidirectional long-term dependencies between time steps. These dependencies can be useful for when you want the network to learn from the complete time series at each time step.

Activation Layers

Function	Description
`reluLayer`	A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.
`leakyReluLayer`	A leaky ReLU layer performs a simple threshold operation, where any input value less than zero is multiplied by a fixed scalar.
`clippedReluLayer`	A clipped ReLU layer performs a simple 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.

Normalization and Dropout Layers

Function	Description
`batchNormalizationLayer`	A batch normalization layer normalizes each input channel across a mini-batch. 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 β and scales it by a learnable scale factor γ. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up training of convolutional neural networks and reduce the sensitivity to network initialization.
`crossChannelNormalizationLayer`	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.
`dropoutLayer`	A dropout layer randomly sets input elements to zero with a given probability.

Pooling Layers

Function	Description
`averagePooling2dLayer`	An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.
`maxPooling2dLayer`	A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.
`maxUnpooling2dLayer`	A max unpooling layer unpools the output of a max pooling layer.

Combination Layers

Function	Description
`additionLayer`	An addition layer adds multiple inputs element-wise. Specify the number of inputs to the layer when you create it. The inputs have names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer to other layers using `connectLayers` or `disconnectLayers`. All inputs to an addition layer must have the same dimension.
`depthConcatenationLayer`	A depth concatenation layer takes multiple inputs that have the same height and width and concatenates them along the third dimension (the channel dimension). The inputs have names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer to other layers using `connectLayers` or `disconnectLayers`.

Output Layers

Function	Description
`softmaxLayer`	A softmax layer applies a softmax function to the input.
`classificationLayer`	A classification output layer holds the name of the loss function the software uses for training the network for multiclass classification, the size of the output, and the class labels.
`regressionLayer`	A regression output layer holds the name of the loss function the software uses for training the network for regression, and the response names.

For an example showing how to create a layer array, see “Construct Network Architecture” on page 1-915.

Alternatively, you can import layers from Caffe using `importCaffeLayers`.

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 paddi
     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 paddi
     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.

```
layers = [ ...
    imageInputLayer([28 28 3])
    convolution2dLayer([5 5],10)
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Display the image input layer by selecting the first layer.

```
layers(1)
```

```
ans =
  ImageInputLayer with properties:

                Name: ''
           InputSize: [28 28 3]

   Hyperparameters
    DataAugmentation: 'none'
       Normalization: 'zerocenter'
```

View the input size of the image input layer.

```
layers(1).InputSize
```

```
ans = 1×3

    28    28     3
```

Display the stride for the convolutional layer.

```
layers(2).Stride
```

```
ans = 1×2

     1     1
```

Access the bias learn rate factor for the fully connected layer.

```
layers(4).BiasLearnRateFactor
```

```
ans = 1
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”

See Also

`DAGNetwork` | `LayerGraph` | `averagePooling2dLayer` | `batchNormalizationLayer` | `classificationLayer` | `clippedReluLayer` | `convolution2dLayer` | `crossChannelNormalizationLayer` | `dropoutLayer` | `fullyConnectedLayer` | `imageInputLayer` | `importCaffeLayers` | `leakyReluLayer` | `maxPooling2dLayer` | `maxUnpooling2dLayer` | `regressionLayer` | `reluLayer` | `softmaxLayer` | `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”

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 1-920.
- Import a pretrained network from Keras using `importKerasNetwork`. For an example, see “Import and Plot Keras Network” on page 1-1092.
- Import a pretrained network from Caffe using `importCaffeNetwork`. For an example, see “Import Caffe Network” on page 1-745.
- Train or fine-tune a network using `trainNetwork`. For an example, see “Train Network for Image Classification” on page 1-922.

Note To learn about other pretrained networks, such as `googlenet` and `resnet50`, see “Pretrained Convolutional Neural Networks”.

Properties

`Layers` — Network layers
`Layer` array

Network layers, specified as a `Layer` array.

Object Functions

activations	Compute convolutional neural network layer activations
classify	Classify data using a trained deep learning neural network
predict	Predict responses using a trained deep learning neural network
predictAndUpdateState	Predict responses using a trained recurrent neural network and update the network state
classifyAndUpdateState	Classify data using a trained recurrent neural network and update the network state
resetState	Reset the state of a recurrent neural network

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.

```
net = alexnet
```

```
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.

```
net.Layers
```

```
ans =

  25x1 Layer array with layers:
```

```
     1   'data'     Image Input                   227x227x3 images with 'zerocenter' no
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 ch
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] an
     6   'conv2'    Convolution                   256 5x5x48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 ch
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] an
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Convolution                   384 3x3x192 convolutions with stride
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Convolution                   256 3x3x192 convolutions with stride
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] an
    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
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench', 'goldfis
```

You can view the names of the classes learned by the network by viewing the `ClassNames` property of the classification output layer (the final layer). View the first 10 classes by selecting the first 10 elements.

```
net.Layers(end).ClassNames(1:10)
```

```
ans =

  1×10 cell array

  Columns 1 through 4

    'tench'    'goldfish'    'great white shark'    'tiger shark'

  Columns 5 through 9

    'hammerhead'    'electric ray'    'stingray'    'cock'    'hen'
```

```
  Column 10

    'ostrich'
```

Import Layers from Caffe Network

Specify the example file `'digitsnet.prototxt'` to import.

```
protofile = 'digitsnet.prototxt';
```

Import the network layers.

```
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]
     3   'relu1'      ReLU                   ReLU
     4   'pool1'      Max Pooling            2x2 max pooling with stride [2  2] and pa
     5   'ip1'        Fully Connected        10 fully connected layer
     6   'loss'       Softmax                softmax
     7   'output'     Classification Output  crossentropyex with 'class1', 'class2', a
```

Train Network for Image Classification

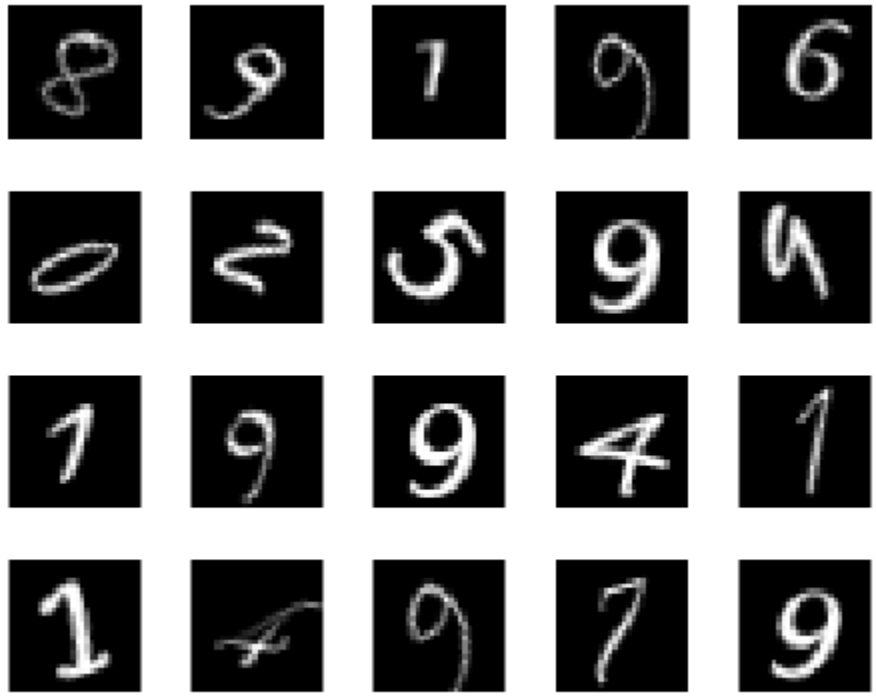
Load the data as an `ImageDatastore` object.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

The datastore contains 10,000 synthetic images of digits 0-9. 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.

```
figure
numImages = 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.

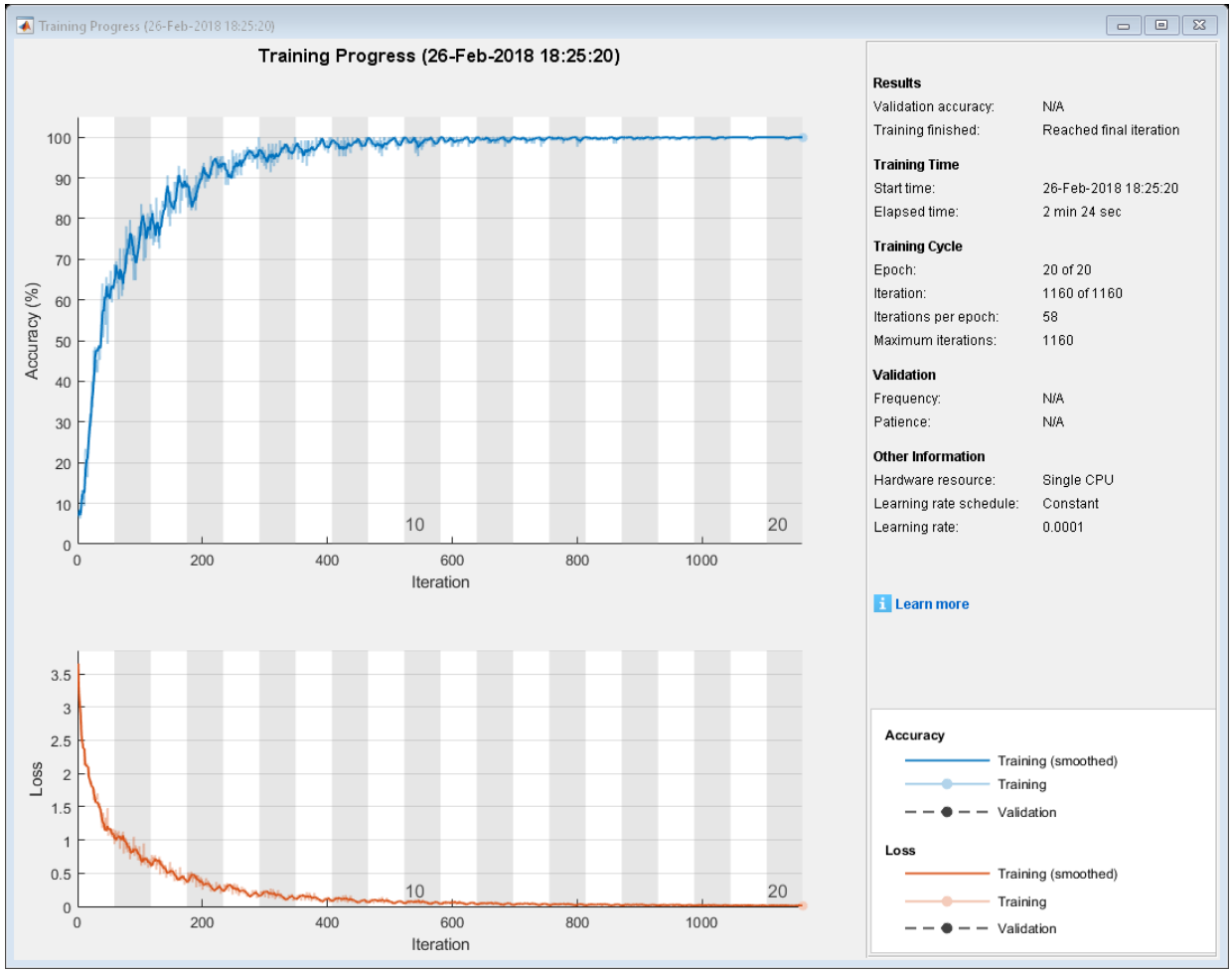
```
layers = [ ...
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

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',0, ...
    '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).

```
YPred = classify(net,imdsTest);
YTest = imdsTest.Labels;
```

Calculate the accuracy. 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 = 0.9896
```

- “Create Simple Deep Learning Network for Classification”
- “Train Convolutional Neural Network for Regression”
- “Sequence Classification Using Deep Learning”

See Also

`DAGNetwork` | `alexnet` | `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`.

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

`ImageDatastore` | `table` | cell array

Data to use for validation during training, specified as one of the following:

- `ImageDatastore` with `categorical` labels for image classification problems.
- `table`, where the first column contains either image paths or images, and the subsequent columns contain the responses. For an image classification problem, the response must be a `categorical` variable in the second table column. For a regression problem, 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.
- cell array `{X,Y}`, where `X` is a numeric array of images and `Y` contains the responses. The first three dimensions of `X` are the height, width, and channels, and the last dimension is the image index. For an image classification problem, `Y` must be a `categorical` vector. For a regression problem, `Y` must be a numeric array. For more information on the allowed shape of `Y`, see details on page 1-0 on the `trainNetwork` page.

During training, `trainNetwork` predicts the labels of the validation data and calculates the validation accuracy and validation loss. To specify the validation frequency, use the `'ValidationFrequency'` name-value pair argument. By default, if the validation loss is larger than or equal to the previously smallest loss five times in a row, then network training stops. To change the number of times that the validation loss is allowed to not decrease before training stops, use the `'ValidationPatience'` name-value pair argument.

`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.

`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 1-672.

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 1-675.

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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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. If no pool is open, then the software opens one based on your default parallel settings.
- `'parallel'` — Use a local parallel pool or compute cluster. If no pool is open, then the software opens 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 the training takes place on all cluster CPUs.

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 increasing the `MiniBatchSize` training option to offset the communication overhead.

To train long short-term memory (LSTM) networks, the hardware resource must be `'auto'`, `'cpu'`, or `'gpu'`.

Specify the execution environment using `trainingOptions`.

Data Types: `char`

`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:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`.

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.

```
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: 5
                        Shuffle: 'once'
                CheckpointPath: ''
          ExecutionEnvironment: 'auto'
                    WorkerLoad: []
                     OutputFcn: []
                         Plots: 'training-progress'
               SequenceLength: 'longest'
          SequencePaddingValue: 0
```

- “Create Simple Deep Learning Network for Classification”
- “Transfer Learning Using AlexNet”

- “Resume Training from a Checkpoint Network”
- “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”

See Also

`trainNetwork` | `trainingOptions`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from a Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Learn About Convolutional Neural Networks”
“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, L_2 regularization factor, and mini-batch size.

Creation

Create a `TrainingOptionsRMSProp` object using `trainingOptions` and specifying `'rmsprop'` as the `solverName`.

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

`ImageDatastore` | `table` | cell array

Data to use for validation during training, specified as one of the following:

- `ImageDatastore` with `categorical` labels for image classification problems.
- `table`, where the first column contains either image paths or images, and the subsequent columns contain the responses. For an image classification problem, the response must be a `categorical` variable in the second table column. For a regression problem, 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.
- cell array `{X,Y}`, where `X` is a numeric array of images and `Y` contains the responses. The first three dimensions of `X` are the height, width, and channels, and the last dimension is the image index. For an image classification problem, `Y` must be a `categorical` vector. For a regression problem, `Y` must be a numeric array. For more information on the allowed shape of `Y`, see details on page 1-0 on the `trainNetwork` page.

During training, `trainNetwork` predicts the labels of the validation data and calculates the validation accuracy and validation loss. To specify the validation frequency, use the `'ValidationFrequency'` name-value pair argument. By default, if the validation loss is larger than or equal to the previously smallest loss five times in a row, then network training stops. To change the number of times that the validation loss is allowed to not decrease before training stops, use the `'ValidationPatience'` name-value pair argument.

`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 1-672.

`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.

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 1-675.

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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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. If no pool is open, then the software opens one based on your default parallel settings.
- `'parallel'` — Use a local parallel pool or compute cluster. If no pool is open, then the software opens 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 the training takes place on all cluster CPUs.

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 increasing the `MiniBatchSize` training option to offset the communication overhead.

To train long short-term memory (LSTM) networks, the hardware resource must be `'auto'`, `'cpu'`, or `'gpu'`.

Specify the execution environment using `trainingOptions`.

Data Types: `char`

`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:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`.

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.

```
options = trainingOptions('rmsprop',...
    'InitialLearnRate',3e-4,...
    'SquaredGradientDecayFactor',0.99,...
    'MaxEpochs',20,...
    'MiniBatchSize',64,...
    'Plots','training-progress')
```

```
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: 5
                         Shuffle: 'once'
                  CheckpointPath: ''
            ExecutionEnvironment: 'auto'
                      WorkerLoad: []
                       OutputFcn: []
                           Plots: 'training-progress'
                  SequenceLength: 'longest'
```

```
    SequencePaddingValue: 0
```

- “Create Simple Deep Learning Network for Classification”
- “Transfer Learning Using AlexNet”
- “Resume Training from a Checkpoint Network”
- “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”

See Also

`trainNetwork` | `trainingOptions`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from a Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Learn About Convolutional Neural Networks”
“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`.

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

`ImageDatastore` | `table` | cell array

Data to use for validation during training, specified as one of the following:

- `ImageDatastore` with `categorical` labels for image classification problems.
- `table`, where the first column contains either image paths or images, and the subsequent columns contain the responses. For an image classification problem, the response must be a `categorical` variable in the second table column. For a regression problem, 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.
- cell array `{X,Y}`, where `X` is a numeric array of images and `Y` contains the responses. The first three dimensions of `X` are the height, width, and channels, and the last dimension is the image index. For an image classification problem, `Y` must be a `categorical` vector. For a regression problem, `Y` must be a numeric array. For more information on the allowed shape of `Y`, see details on page 1-0 on the `trainNetwork` page.

During training, `trainNetwork` predicts the labels of the validation data and calculates the validation accuracy and validation loss. To specify the validation frequency, use the `'ValidationFrequency'` name-value pair argument. By default, if the validation loss is larger than or equal to the previously smallest loss five times in a row, then network training stops. To change the number of times that the validation loss is allowed to not decrease before training stops, use the `'ValidationPatience'` name-value pair argument.

`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.

`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 1-672.

`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 1-672.

`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.

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 1-675.

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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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. If no pool is open, then the software opens one based on your default parallel settings.
- `'parallel'` — Use a local parallel pool or compute cluster. If no pool is open, then the software opens 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 the training takes place on all cluster CPUs.

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 increasing the `MiniBatchSize` training option to offset the communication overhead.

To train long short-term memory (LSTM) networks, the hardware resource must be `'auto'`, `'cpu'`, or `'gpu'`.

Specify the execution environment using `trainingOptions`.

Data Types: `char`

`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:

Field	Description
`Epoch`	Current epoch number
`Iteration`	Current iteration number
`TimeSinceStart`	Time in seconds since the start of training
`TrainingLoss`	Current mini-batch loss
`ValidationLoss`	Loss on the validation data
`BaseLearnRate`	Current base learning rate
`TrainingAccuracy`	Accuracy on the current mini-batch (classification networks)
`TrainingRMSE`	RMSE on the current mini-batch (regression networks)
`ValidationAccuracy`	Accuracy on the validation data (classification networks)
`ValidationRMSE`	RMSE on the validation data (regression networks)
`State`	Current training state, with a possible value of `"start"`, `"iteration"`, or `"done"`.

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: 5
                           Shuffle: 'once'
```

```
      CheckpointPath: ''
ExecutionEnvironment: 'auto'
          WorkerLoad: []
           OutputFcn: []
               Plots: 'training-progress'
      SequenceLength: 'longest'
SequencePaddingValue: 0
```

- “Create Simple Deep Learning Network for Classification”
- “Transfer Learning Using AlexNet”
- “Resume Training from a Checkpoint Network”
- “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”

References

[1] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

See Also

`trainNetwork` | `trainingOptions`

Topics

“Create Simple Deep Learning Network for Classification”
“Transfer Learning Using AlexNet”
“Resume Training from a Checkpoint Network”
“Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud”
“Learn About Convolutional Neural Networks”
“Specify Layers of Convolutional Neural Network”
“Set Up Parameters and Train Convolutional Neural Network”

Introduced in R2018a

activations

Compute convolutional neural network layer activations

You can extract features using a trained convolutional neural network (ConvNet, CNN) 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.

Syntax

```
features = activations(net,X,layer)
features = activations(net,X,layer,Name,Value)
```

Description

`features = activations(net,X,layer)` returns network activations for a specific layer using the trained network `net` and the data in `X`.

The function only supports networks with an image input layer.

`features = activations(net,X,layer,Name,Value)` returns network activations for a specific layer 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.

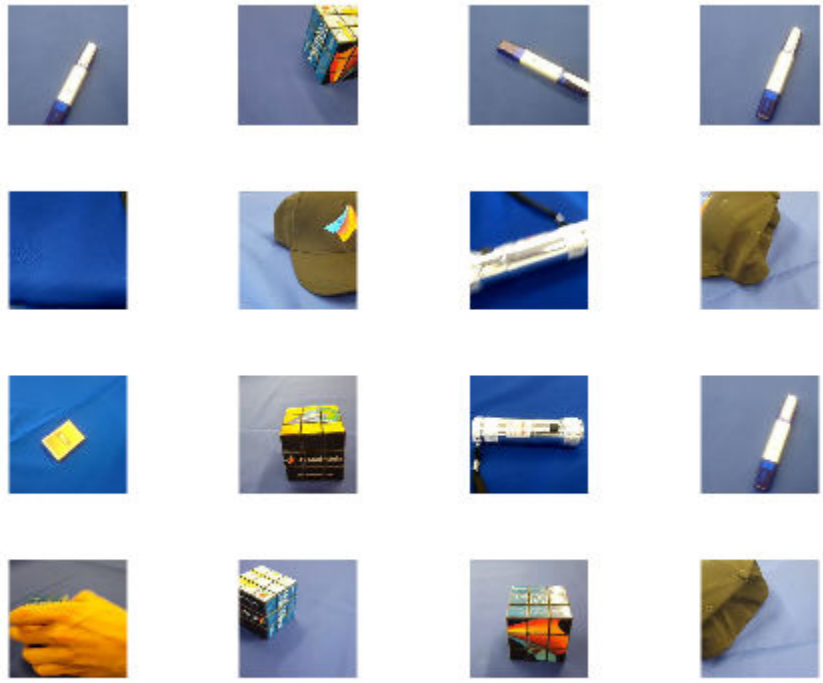
Load Data

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.

```
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 a pretrained AlexNet network. If Neural Network Toolbox Model *for AlexNet Network* 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                   227x227x3 images with 'zerocenter' no
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 ch
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] an
     6   'conv2'    Convolution                   256 5x5x48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 ch
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] an
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Convolution                   384 3x3x192 convolutions with stride
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Convolution                   256 3x3x192 convolutions with stride
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] an
    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
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench' and 999 o
```

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.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize =

   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`.

```
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.

```
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).

```
classifier = fitcecoc(featuresTrain,YTrain);
```

Classify Test Images

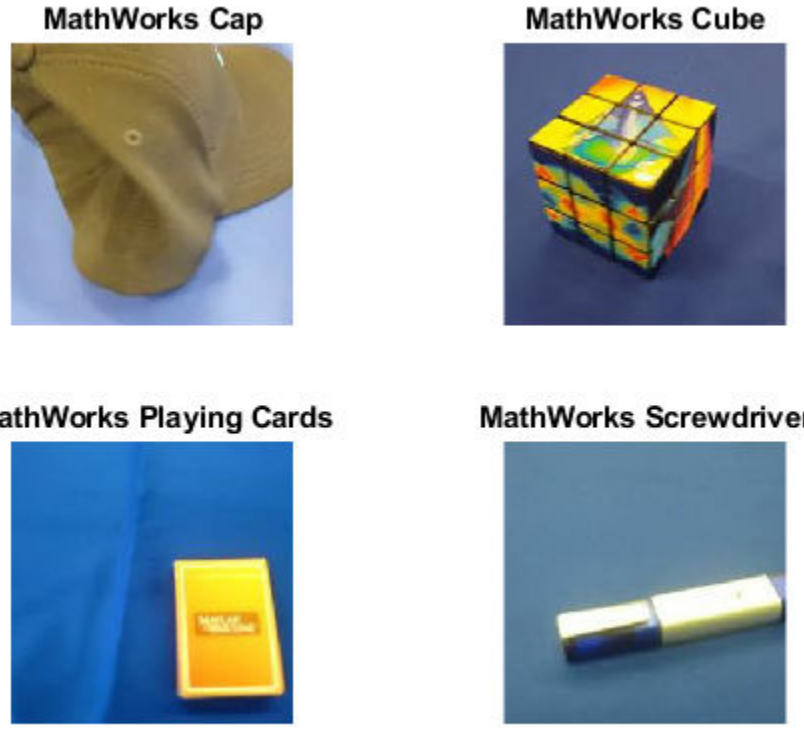
Classify the test images using the trained SVM model the features extracted from the test images.

```
YPred = predict(classifier,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(char(label))
end
```



Calculate the classification accuracy on the test set. Accuracy is the fraction of labels that the network predicts correctly.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 1
```

This SVM has high accuracy. If the accuracy is not high enough using feature extraction, then try transfer learning instead.

- “Transfer Learning Using AlexNet”

- “Visualize Activations of a Convolutional Neural Network”

Input Arguments

net — Trained network

`SeriesNetwork` object | `DAGNetwork` object

Trained network, specified as a `SeriesNetwork` or `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 Convolutional Neural Networks”.

`activations` only supports networks with an image input layer.

X — Image data

3-D array of a single image | 4-D array of images | `table` | `ImageDatastore` object | mini-batch datastore

Image data, specified as one of the following.

- 3-D array representing a single image. `X` has size *h*-by-*w*-by-*c*, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the image, respectively.
- 4-D array representing a stack of images. `X` has size *h*-by-*w*-by-*c*-by-*N*, where *N* is the number of images.
- `table`, where the first column contains either paths to images, or 3-D arrays representing images.
- `ImageDatastore`.
- mini-batch datastore. For more information on built-in and custom mini-batch datastores, see “Preprocess Images for Deep Learning”.

If the `'OutputAs'` value equals `'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.

layer — Layer to extract features from

numeric index | character vector

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 on 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 single 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 one of the following:

- `'channels'` — `Y` 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. *n* is the number of observations. Each *h*-by-*w*-by-*c* subarray is the output for a single observation.
- `'rows'` — `Y` is an *n*-by-*m* matrix, where *n* is the number of observations, and `m` is the number of output elements from the chosen layer.
- `'columns'` — `Y` is an *m*-by-*n* matrix, where *m* is the number of output elements from the chosen layer, and *n* is the number of observations. Each column of the matrix is the output for a single observation.

If the `'OutputAs'` value equals `'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`

`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 a network layer
n-by-*m* matrix | *m*-by-*n* matrix | *h*-by-*w*-by-*c*-by-*n* array

Activations from a network layer, returned as one of the following, depending on the value of the `'OutputAs'` name-value pair argument.

`trainedFeatures`	`'OutputAs'` value
n-by-*m* matrix	`'rows'`
m-by-*n* matrix	`'columns'`
h-by-*w*-by-*c*-by-*n* array	`'channels'`

Algorithms

All functions for deep learning training, prediction, and validation in Neural Network 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.

See Also

`classify` | `deepDreamImage` | `predict` | `trainNetwork`

Topics

“Transfer Learning Using AlexNet”
“Visualize Activations of a Convolutional Neural Network”
“Deep Learning in MATLAB”

Introduced in R2016a

classify

Classify data using a trained deep learning neural network

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.

Syntax

```
[YPred,scores] = classify(net,X)
[YPred,scores] = classify(net,C)
[YPred,scores] = classify( ___ ,Name,Value)
```

Description

`[YPred,scores] = classify(net,X)` predicts class labels for the image data in `X` using the trained network, `net`.

`[YPred,scores] = classify(net,C)` predicts class labels for the time series or sequence data in `C` using the trained LSTM network, `net`.

`[YPred,scores] = classify( ___ ,Name,Value)` predicts class labels with additional options specified by one or more name-value pair arguments.

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.

```
rng('default')
net = trainNetwork(XTrain,YTrain,layers,options);
```

```
Training on single CPU.
Initializing image normalization.
|=========================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate
|=========================================================================================
|       1 |           1 |       00:00:00 |        7.81% |       2.3026 |          0.010
|       2 |          50 |       00:00:04 |       33.59% |       2.2735 |          0.010
|       3 |         100 |       00:00:09 |       48.44% |       1.6613 |          0.010
|       4 |         150 |       00:00:13 |       64.06% |       1.1803 |          0.010
|       6 |         200 |       00:00:18 |       64.06% |       1.0499 |          0.010
|       7 |         250 |       00:00:23 |       76.56% |       0.8392 |          0.010
|       8 |         300 |       00:00:27 |       77.34% |       0.6981 |          0.010
|       9 |         350 |       00:00:32 |       77.34% |       0.7084 |          0.010
|      11 |         400 |       00:00:37 |       87.50% |       0.4902 |          0.010
|      12 |         450 |       00:00:42 |       91.41% |       0.3839 |          0.010
|      13 |         500 |       00:00:47 |       92.19% |       0.2986 |          0.010
|      15 |         550 |       00:00:51 |       93.75% |       0.2583 |          0.010
|      16 |         600 |       00:00:56 |       97.66% |       0.2009 |          0.010
|      17 |         650 |       00:01:01 |       92.97% |       0.2642 |          0.010
|      18 |         700 |       00:01:06 |       97.66% |       0.1448 |          0.010
|      20 |         750 |       00:01:10 |       96.88% |       0.1314 |          0.010
```

```
|      21 |          800 |       00:01:15 |       97.66% |       0.1233 |          0.010
|      22 |          850 |       00:01:19 |       98.44% |       0.1009 |          0.010
|      24 |          900 |       00:01:24 |      100.00% |       0.1051 |          0.010
|      25 |          950 |       00:01:29 |       97.66% |       0.1483 |          0.010
|      26 |         1000 |       00:01:33 |       99.22% |       0.0743 |          0.010
|      27 |         1050 |       00:01:38 |      100.00% |       0.0603 |          0.010
|      29 |         1100 |       00:01:42 |       99.22% |       0.0769 |          0.010
|      30 |         1150 |       00:01:47 |      100.00% |       0.0524 |          0.010
|      30 |         1170 |       00:01:49 |      100.00% |       0.0566 |          0.010
|=========================================================================================
```

Run the trained network on a test set.

```
[XTest,YTest]= digitTest4DArrayData;
YPred = classify(net,XTest);
```

Display the first 10 images in the test data and compare to the classification from `classify`.

```
[YTest(1:10,:) YPred(1:10,:)]
```

```
ans = 10x2 categorical array
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
     0      0
```

The results from `classify` match the true digits for the first ten images.

Calculate the accuracy over all test data.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9770
```

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.

```
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.

```
load JapaneseVowelsTest
```

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
     1      1
```

Calculate the classification accuracy of the predictions.

```
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.8595
```

- “Classify Image Using GoogLeNet”
- “Classify Webcam Images Using Deep Learning”

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 `alexnet` function) or by training your own network using `trainNetwork`.

X — Image data

3-D array of a single image | 4-D array of images | `table` | `ImageDatastore` object | mini-batch datastore

Image data, specified as one of the following.

- 3-D array representing a single image. `X` has size *h*-by-*w*-by-*c*, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the image, respectively.
- 4-D array representing a stack of images. `X` has size *h*-by-*w*-by-*c*-by-*N*, where *N* is the number of images.
- `table`, where the first column contains either paths to images, or 3-D arrays representing images.
- `ImageDatastore`.
- mini-batch datastore. For more information on built-in and custom mini-batch datastores, see “Preprocess Images for Deep Learning”.

C — Sequence or time series data

cell array of matrices | matrix

Sequence or time series data, specified as a matrix representing a single time series, or a cell array of matrices representing multiple time series.

- If `C` is a matrix representing a single time series, then `C` is a *D*-by-*S* matrix, where *D* is the number of data points per time step, and *S* is the number of time steps.
- If `C` is a cell array of time series, then `C` is an *N*-by-1 cell array, where *N* is the number of observations. Each entry of `C` is a time series represented by a matrix, with rows corresponding to data points and columns corresponding to time steps.

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.

Example: `'MiniBatchSize',256`

`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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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`

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.

Task	Format
Image classification	*N*-by-1 categorical vector of labels, where *N* is the number of observations.
Sequence-to-label classification	

Task	Format
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.

For sequence-to-sequence classification problems with one observation, `C` 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`.

Task	Format
Image classification	*N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes
Sequence-to-label classification	
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 responses. Each sequence has the same number of time steps as the corresponding input sequence.

For sequence-to-sequence classification problems with one observation, `C` 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 Neural Network 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 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

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

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

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.

Syntax

```
YPred = predict(net,X)
YPred = predict(net,C)
YPred = predict( ___ ,Name,Value)
```

Description

`YPred = predict(net,X)` predicts responses for the image data in `X` using the trained network `net`.

`YPred = predict(net,C)` predicts responses for the sequence or time series data in `C` using the trained LSTM network `net`.

`YPred = predict( ___ ,Name,Value)` predicts responses with additional options specified by one or more name-value pair arguments.

Examples

Predict Output Scores Using a 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.

```
rng('default')
net = trainNetwork(XTrain,YTrain,layers,options);
```

```
Training on single CPU.
Initializing image normalization.
|=========================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate
|=========================================================================================
|       1 |           1 |       00:00:00 |        7.81% |       2.3026 |          0.010
|       2 |          50 |       00:00:04 |       33.59% |       2.2735 |          0.010
|       3 |         100 |       00:00:08 |       48.44% |       1.6613 |          0.010
|       4 |         150 |       00:00:12 |       64.06% |       1.1803 |          0.010
|       6 |         200 |       00:00:17 |       64.06% |       1.0499 |          0.010
|       7 |         250 |       00:00:22 |       76.56% |       0.8392 |          0.010
|       8 |         300 |       00:00:26 |       77.34% |       0.6981 |          0.010
|       9 |         350 |       00:00:31 |       77.34% |       0.7084 |          0.010
|      11 |         400 |       00:00:35 |       87.50% |       0.4902 |          0.010
|      12 |         450 |       00:00:40 |       91.41% |       0.3839 |          0.010
|      13 |         500 |       00:00:45 |       92.19% |       0.2986 |          0.010
|      15 |         550 |       00:00:49 |       93.75% |       0.2583 |          0.010
|      16 |         600 |       00:00:54 |       97.66% |       0.2009 |          0.010
|      17 |         650 |       00:00:59 |       92.97% |       0.2642 |          0.010
|      18 |         700 |       00:01:04 |       97.66% |       0.1448 |          0.010
|      20 |         750 |       00:01:09 |       96.88% |       0.1314 |          0.010
```

```
|      21 |          800 |       00:01:14 |       97.66% |       0.1233 |          0.010
|      22 |          850 |       00:01:19 |       98.44% |       0.1009 |          0.010
|      24 |          900 |       00:01:23 |      100.00% |       0.1051 |          0.010
|      25 |          950 |       00:01:28 |       97.66% |       0.1483 |          0.010
|      26 |         1000 |       00:01:33 |       99.22% |       0.0743 |          0.010
|      27 |         1050 |       00:01:37 |      100.00% |       0.0603 |          0.010
|      29 |         1100 |       00:01:42 |       99.22% |       0.0769 |          0.010
|      30 |         1150 |       00:01:47 |      100.00% |       0.0524 |          0.010
|      30 |         1170 |       00:01:48 |      100.00% |       0.0566 |          0.010
|=========================================================================================
```

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

    0.9989    0.0000    0.0004    0.0004    0.0000    0.0000    0.0001    0.0000    0.0
    0.8827    0.0000    0.0151    0.0001    0.0000    0.0002    0.0031    0.0001    0.0
    0.9999    0.0000    0.0001    0.0000    0.0000    0.0000    0.0000    0.0000    0.0
    0.9723    0.0000    0.0000    0.0000    0.0000    0.0000    0.0025    0.0000    0.0
```

```
    0.9434    0.0000    0.0091    0.0005    0.0000    0.0000    0.0001    0.0005    0.0
    0.9791    0.0000    0.0003    0.0000    0.0000    0.0000    0.0020    0.0000    0.0
    0.9938    0.0000    0.0001    0.0000    0.0000    0.0000    0.0039    0.0000    0.0
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0
    0.9338    0.0000    0.0022    0.0004    0.0003    0.0024    0.0003    0.0000    0.0
    0.9090    0.0000    0.0054    0.0028    0.0001    0.0006    0.0392    0.0000    0.0
```

`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
```

Load the test data.

```
load JapaneseVowelsTest
```

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    0.0
    0.9868    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0010    0.0
    0.9924    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0006    0.0
    0.9896    0.0000    0.0000    0.0000    0.0006    0.0009    0.0001    0.0007    0.0
    0.9965    0.0000    0.0000    0.0000    0.0007    0.0009    0.0000    0.0003    0.0
    0.9888    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0008    0.0
    0.9886    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0008    0.0
    0.9982    0.0000    0.0000    0.0000    0.0006    0.0007    0.0000    0.0001    0.0
    0.9883    0.0000    0.0000    0.0000    0.0006    0.0010    0.0001    0.0008    0.0
    0.9959    0.0000    0.0000    0.0000    0.0007    0.0011    0.0000    0.0004    0.0
```

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 `alexnet` function) or by training your own network using `trainNetwork`.

X — Image data

3-D array of a single image | 4-D array of images | `table` | `ImageDatastore` object | mini-batch datastore

Image data, specified as one of the following.

- 3-D array representing a single image. `X` has size *h*-by-*w*-by-*c*, where *h*, *w*, and *c* correspond to the height, width, and the number of channels of the image, respectively.
- 4-D array representing a stack of images. `X` has size *h*-by-*w*-by-*c*-by-*N*, where *N* is the number of images.
- `table`, where the first column contains either paths to images, or 3-D arrays representing images.
- `ImageDatastore`.
- mini-batch datastore. For more information on built-in and custom mini-batch datastores, see “Preprocess Images for Deep Learning”.

C — Sequence or time series data

cell array of matrices | matrix

Sequence or time series data, specified as a matrix representing a single time series, or a cell array of matrices representing multiple time series.

- If `C` is a matrix representing a single time series, then `C` is a *D*-by-*S* matrix, where *D* is the number of data points per time step, and *S* is the number of time steps.
- If `C` is a cell array of time series, then `C` is an *N*-by-1 cell array, where *N* is the number of observations. Each entry of `C` is a time series represented by a matrix, with rows corresponding to data points and columns corresponding to time steps.

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.

Example: `'MiniBatchSize',256`

`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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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`

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.

Task	Format
Image classification	*N*-by-*K* matrix, where *N* is the number of observations, and *K* is the number of classes
Sequence-to-label classification	
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 responses. Each sequence has the same number of time steps as the corresponding input sequence.

The following table describes the format for regression problems.

Task	Format
Image Regression	• *N*-by-*r* matrix, where *N* is the number of observations and *r* is the number of responses. • *h*-by-*w*-by-*c*-by-*N* numeric array, where *N* is the number of observations and *h*-by-*w*-by-*c* is the image size of a single response.
Sequence-to-one regression	*N*-by-*r* matrix, where *N* is the number of observations 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 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.

For sequence-to-sequence regression problems with one observation, `C` can be a matrix. In this case, `YPred` is a matrix of responses.

Algorithms

If the image data contains `NaN`s, `predict` propagates them through the network. If the network has ReLU layers, these layers ignore `NaN`s. 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 Neural Network 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

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

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 user-defined PReLU layer.

To use the example layer `examplePreluLayer`, add the example folder to the path.

```
exampleFolder = genpath(fullfile(matlabroot,'examples','nnet'));
addpath(exampleFolder)
```

Create a layer array including a user-defined pReLU layer.

```
layers = [ ...
    imageInputLayer([28 28 1])
```

```
    convolution2dLayer(5,20)
    batchNormalizationLayer
    examplePreluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the learn rate factor of the `'Alpha'` learnable parameter of the `examplePreluLayer` to 2.

```
layers(4) = setLearnRateFactor(layers(4),'Alpha',2);
```

View the updated learn rate factor.

```
factor = getLearnRateFactor(layers(4),'Alpha')
```

```
factor = 2
```

Remove the example folder from the path using `rmpath`.

```
rmpath(exampleFolder)
```

Input Arguments

`layer` — Input layer

scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name

character vector

Parameter name, specified as a character vector.

Example: `'Alpha'`

Data Types: `char`

`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 user-defined PReLU layer.

To use the example layer `examplePreluLayer`, add the example folder to the path.

```
exampleFolder = genpath(fullfile(matlabroot,'examples','nnet'));
addpath(exampleFolder)
```

Create a layer array including a user-defined pReLU layer.

```
layers = [ ...
    imageInputLayer([28 28 1])
```

```
    convolution2dLayer(5,20)
    batchNormalizationLayer
    examplePreluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the learn rate factor of the `'Alpha'` learnable parameter of the `examplePreluLayer` to 2.

```
layers(4) = setLearnRateFactor(layers(4),'Alpha',2);
```

View the updated learn rate factor.

```
factor = getLearnRateFactor(layers(4),'Alpha')
```

```
factor = 2
```

Remove the example folder from the path using `rmpath`.

```
rmpath(exampleFolder)
```

Input Arguments

`layer` — Input layer
scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name
character vector

Parameter name, specified as a character vector.

Example: `'Alpha'`

Data Types: `char`

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.

To use the example layer `examplePreluLayer`, add the example folder to the path.

```
exampleFolder = genpath(fullfile(matlabroot,'examples','nnet'));
addpath(exampleFolder)
```

Create a layer array including a user-defined pReLU layer.

```
layers = [ ...
    imageInputLayer([28 28 1])
```

```
    convolution2dLayer(5,20)
    batchNormalizationLayer
    examplePreluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the L2 regularization factor of the `'Alpha'` learnable parameter of the `examplePreluLayer` to 2.

```
layers(4) = setL2Factor(layers(4),'Alpha',2);
```

View the updated L2 regularization factor.

```
factor = getL2Factor(layers(4),'Alpha')
```

```
factor = 2
```

Remove the example folder from the path using `rmpath`.

```
rmpath(exampleFolder)
```

Input Arguments

`layer` — Input layer

scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name

character vector

Parameter name, specified as a character vector.

Example: `'Alpha'`

Data Types: `char`

`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

```
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.

To use the example layer `examplePreluLayer`, add the example folder to the path.

```
exampleFolder = genpath(fullfile(matlabroot,'examples','nnet'));
addpath(exampleFolder)
```

Create a layer array including a user-defined pReLU layer.

```
layers = [ ...
    imageInputLayer([28 28 1])
```

```
    convolution2dLayer(5,20)
    batchNormalizationLayer
    examplePreluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the L2 regularization factor of the `'Alpha'` learnable parameter of the `examplePreluLayer` to 2.

```
layers(4) = setL2Factor(layers(4),'Alpha',2);
```

View the updated L2 regularization factor.

```
factor = getL2Factor(layers(4),'Alpha')
```

```
factor = 2
```

Remove the example folder from the path using `rmpath`.

```
rmpath(exampleFolder)
```

Input Arguments

`layer` — Input layer
scalar `Layer` object

Input layer, specified as a scalar `Layer` object.

`parameterName` — Parameter name
character vector

Parameter name, specified as a character vector.

Example: `'Alpha'`

Data Types: `char`

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

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.

Syntax

```
[updatedNet,YPred] = classifyAndUpdateState(recNet,C)
[updatedNet,YPred] = classifyAndUpdateState(___,Name,Value)
[updatedNet,YPred,scores] = classifyAndUpdateState(___)
```

Description

`[updatedNet,YPred] = classifyAndUpdateState(recNet,C)` classifies the data in `C` 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 1-998

`[updatedNet,YPred,scores] = classifyAndUpdateState(___)` uses any of the arguments in the previous syntaxes, returns a matrix of classification scores, and updates the network state.

Examples

Classify and Update Network State

Classify data using a recurrent neural network and update the network state.

To reproduce the results in this example, set `rng` to `'default'`.

```
rng('default')
```

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.

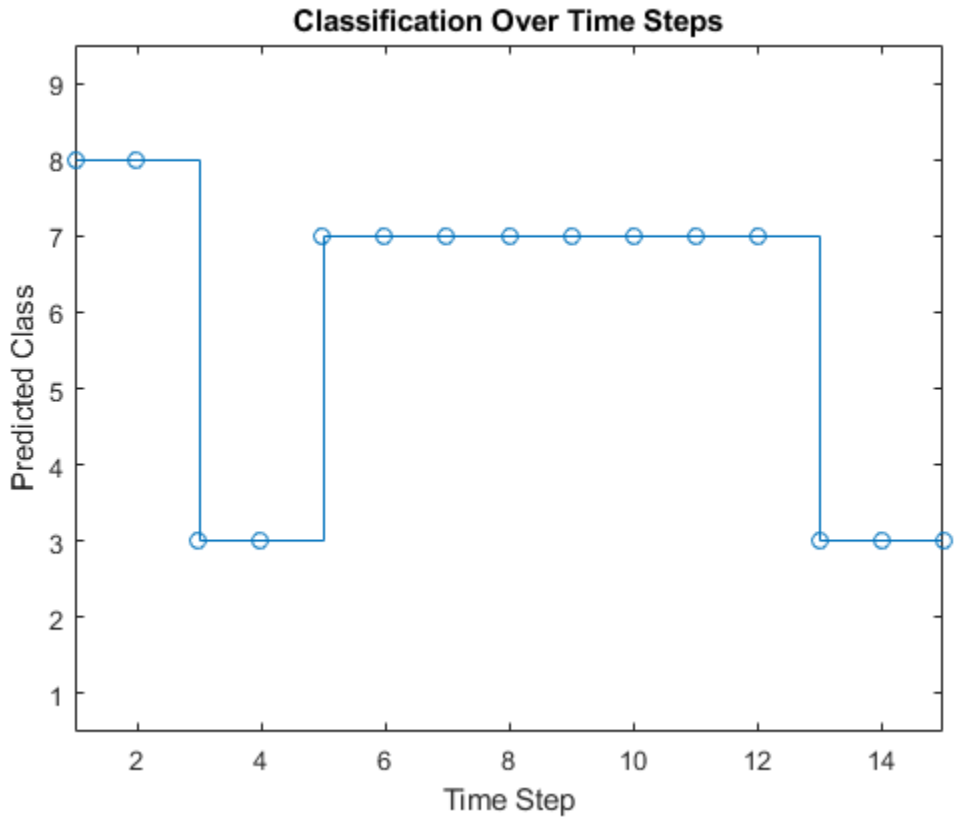
```
load JapaneseVowelsTest
```

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.

```
figure
stairs(labels, '-o')
xlim([1 numTimeSteps])
xlabel("Time Step")
ylabel("Predicted Class")
title("Classification Over Time Steps")
```

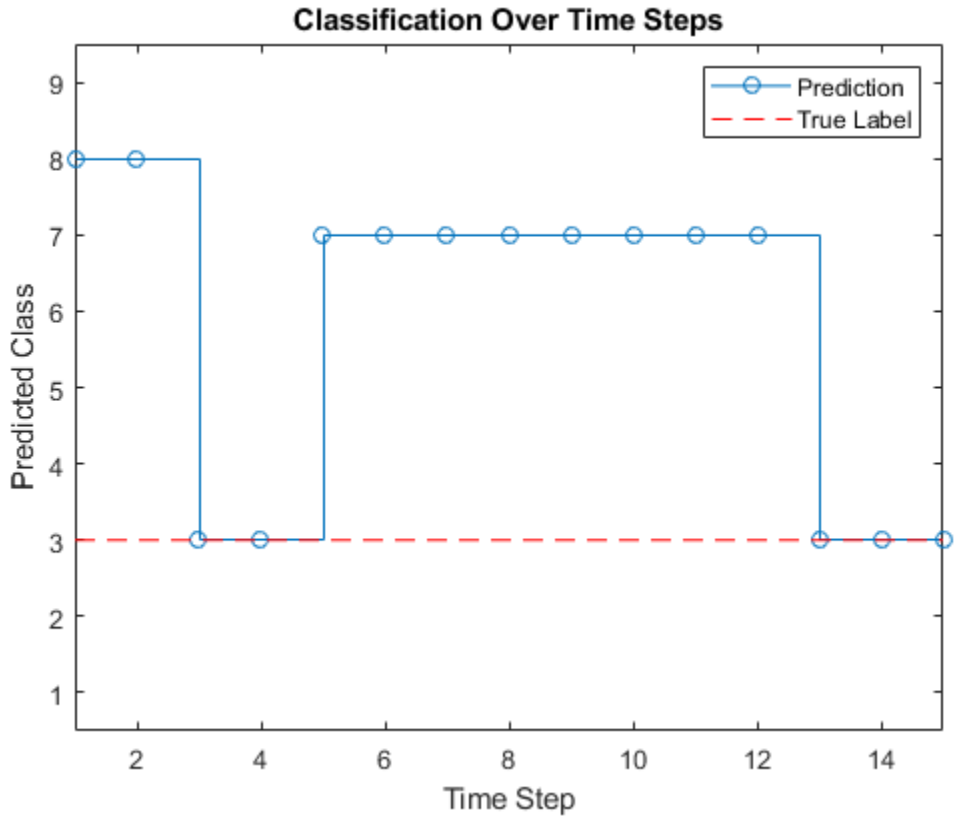


Compare the predictions with the true label. Plot a horizontal line showing the true label of the observation.

```
trueLabel = YTest(94)
```

```
trueLabel = 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

Trained recurrent neural network, specified as a `SeriesNetwork` 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).

`C` — Sequence or time series data
cell array of matrices | matrix

Sequence or time series data, specified as a matrix representing a single time series, or a cell array of matrices representing multiple time series.

- If `C` is a matrix representing a single time series, then `C` is a *D*-by-*S* matrix, where *D* is the number of data points per time step, and *S* is the number of time steps.
- If `C` is a cell array of time series, then `C` is an *N*-by-1 cell array, where *N* is the number of observations. Each entry of `C` is a time series represented by a matrix, with rows corresponding to data points and columns corresponding to time steps.

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[updatedNet, YPred] = classifyAndUpdateState(recNet,C,'MiniBatchSize',27)` classifies data 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.

Example: `'MiniBatchSize',256`

`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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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`

Output Arguments

`updatedNet` — Updated network
`SeriesNetwork` object

Updated network, returned as a `SeriesNetwork` object.

`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.

For sequence-to-sequence classification problems with one observation, `C` 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.

Task	Format
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 responses. Each sequence has the same number of time steps as the corresponding input sequence.

For sequence-to-sequence classification problems with one observation, `C` 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 Neural Network 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

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`bilstmLayer` | `classify` | `lstmLayer` | `predict` | `predictAndUpdateState` | `resetState` | `sequenceInputLayer`

Topics

“Sequence Classification Using Deep Learning”
“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

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.

Syntax

```
[updatedNet,YPred] = predictAndUpdateState(recNet,C)
[updatedNet,YPred] = predictAndUpdateState( ___ ,Name,Value)
```

Description

`[updatedNet,YPred] = predictAndUpdateState(recNet,C)` predicts responses for data in `C` 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.

Examples

Predict and Update Network State

Predict responses using a trained recurrent neural network and update the network state.

To reproduce the results in this example, set `rng` to `'default'`.

```
rng('default')
```

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.

```
load JapaneseVowelsTest
```

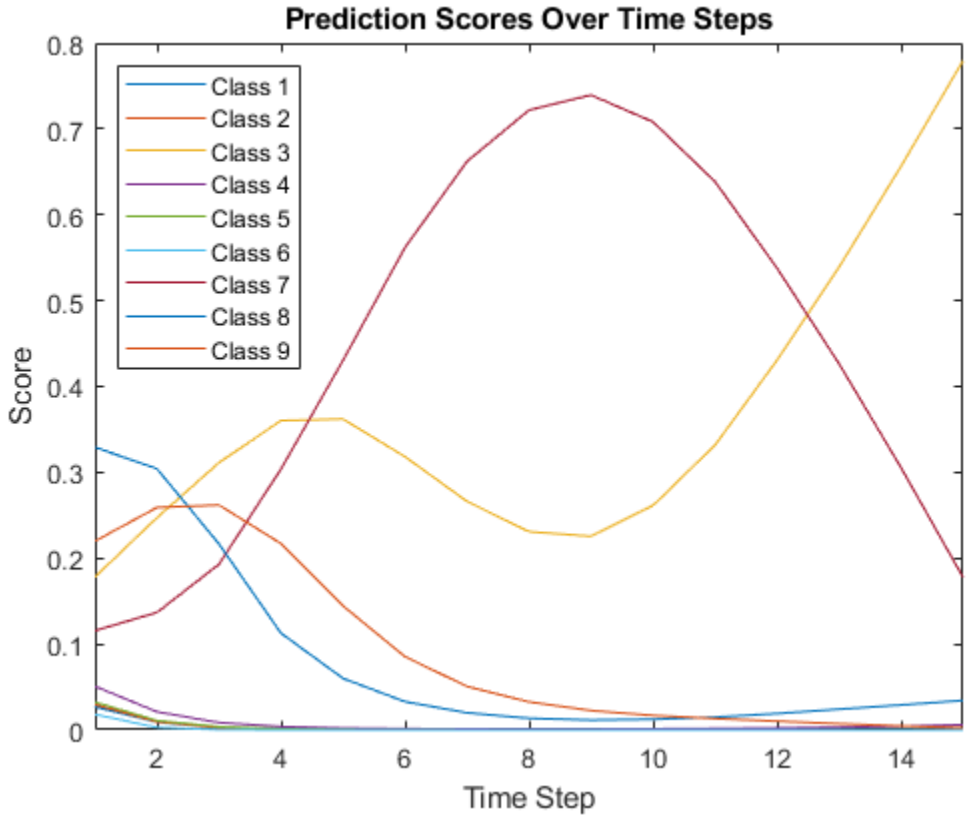
Loop over the time steps in a sequence. Predict the scores of each time step and update the network state.

```
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.

```
figure
lines = plot(scores');
xlim([1 numTimeSteps])
legend("Class " + net.Layers(end).ClassNames, ...
    'Location', 'northwest')
xlabel("Time Step")
```

```
ylabel("Score")
title("Prediction Scores Over Time Steps")
```

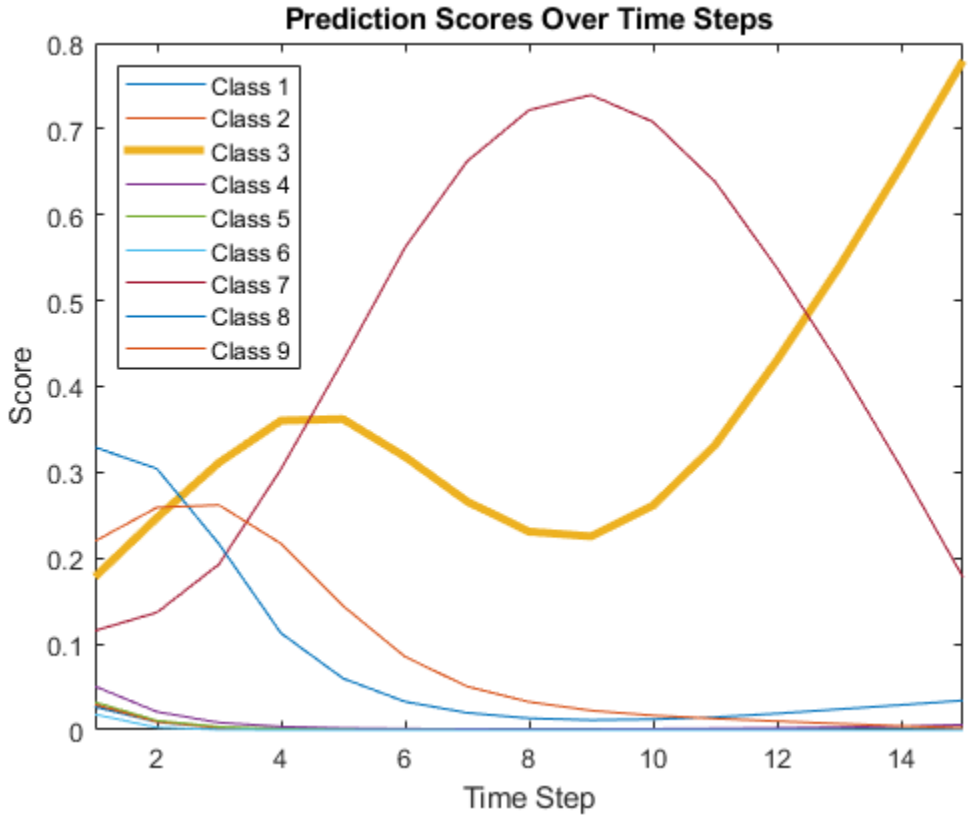


Highlight the prediction scores over time steps for the correct class.

```
trueLabel = YTest(94)
```

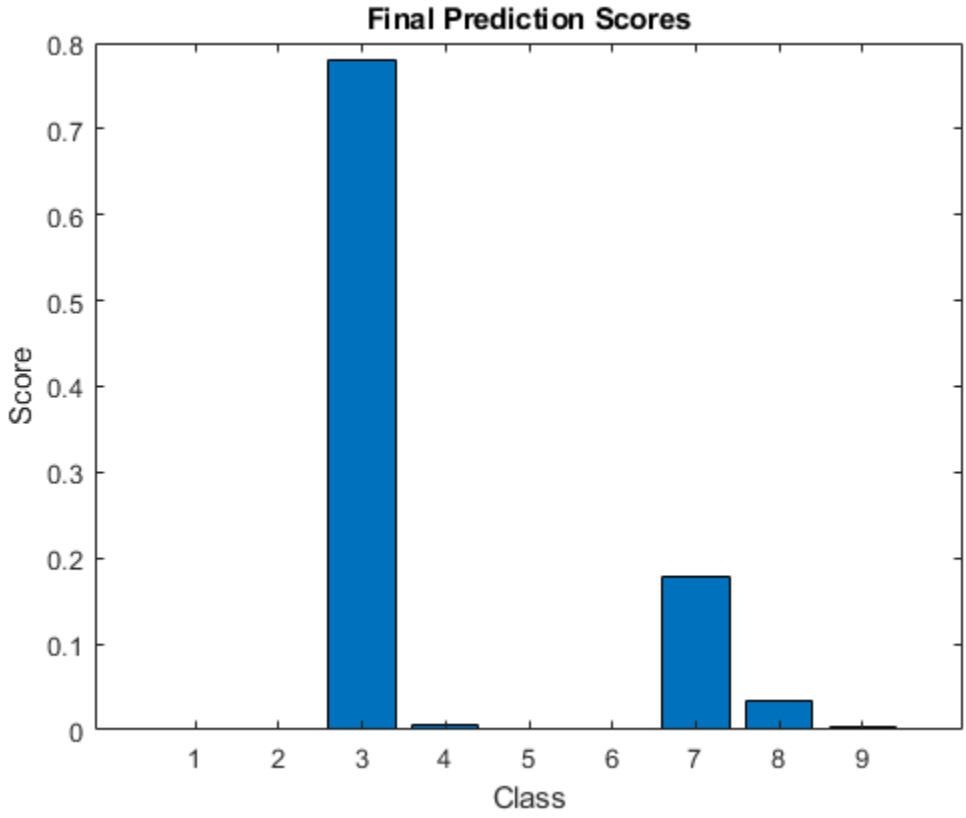
```
trueLabel = categorical
     3
```

```
lines(trueLabel).LineWidth = 3;
```



Display the final time step prediction in a bar chart.

```
figure
bar(score)
title("Final Prediction Scores")
xlabel("Class")
ylabel("Score")
```



Input Arguments

`recNet` — Trained recurrent neural network
`SeriesNetwork` object

Trained recurrent neural network, specified as a `SeriesNetwork` 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).

C — Sequence or time series data
cell array of matrices | matrix

Sequence or time series data, specified as a matrix representing a single time series, or a cell array of matrices representing multiple time series.

- If C is a matrix representing a single time series, then C is a *D*-by-*S* matrix, where *D* is the number of data points per time step, and *S* is the number of time steps.
- If C is a cell array of time series, then C is an *N*-by-1 cell array, where *N* is the number of observations. Each entry of C is a time series represented by a matrix, with rows corresponding to data points and columns corresponding to time steps.

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 single 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.

Example: `'MiniBatchSize',256`

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.
- `'shortest'` — Truncate sequences in each mini-batch to have the same length as the shortest sequence.
- Positive integer — Pad sequences in each mini-batch to have the same length as the longest sequence, then split into smaller sequences of the specified length. If splitting occurs, then the function creates extra mini-batches.

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`

Output Arguments

`updatedNet` — Updated network
`SeriesNetwork` object

Updated network, returned as a `SeriesNetwork` object.

`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.

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 responses. Each sequence has the same number of time steps as the corresponding input sequence.

For sequence-to-sequence classification problems with one observation, `C` 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.

Task	Format
Sequence-to-one regression	*N*-by-*r* matrix, where *N* is the number of observations 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 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.

For sequence-to-sequence problems with one observation, `C` can be a matrix. In this case, `YPred` is a matrix of responses.

Algorithms

All functions for deep learning training, prediction, and validation in Neural Network 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

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

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”
“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 to an initial state of zeros.

Examples

Reset Network State

Reset the network state between sequence predictions.

To reproduce the results in this example, set `rng` to `'default'`.

```
rng('default')
```

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.

```
load JapaneseVowelsTest
```

Classify a sequence and update the network state. For reproducibility, set `rng` to `'shuffle'`.

```
rng('shuffle')
X = XTest{94};
[net,label] = classifyAndUpdateState(net,XTest{94});
```

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

Trained recurrent neural network, specified as a `SeriesNetwork` 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

Updated network, returned as a `SeriesNetwork` object.

References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`bilstmLayer` | `classifyAndUpdateState` | `lstmLayer` | `predictAndUpdateState` | `sequenceInputLayer`

Topics

“Sequence Classification Using Deep Learning”
“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

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.

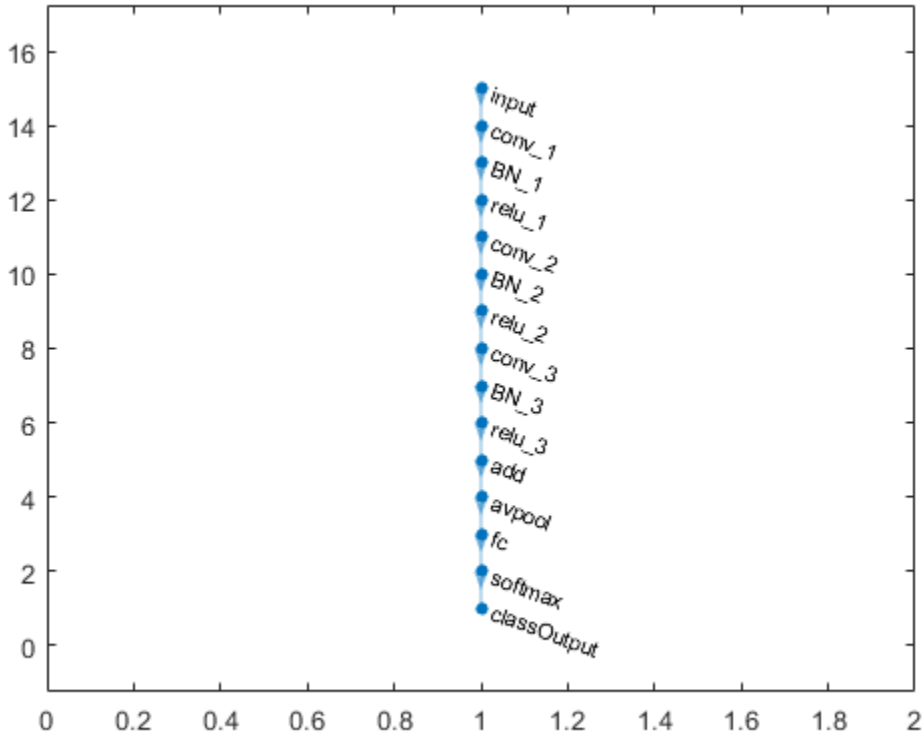
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs that the addition layer should sum. All layers must have names and all names must be unique.

```
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')];
```

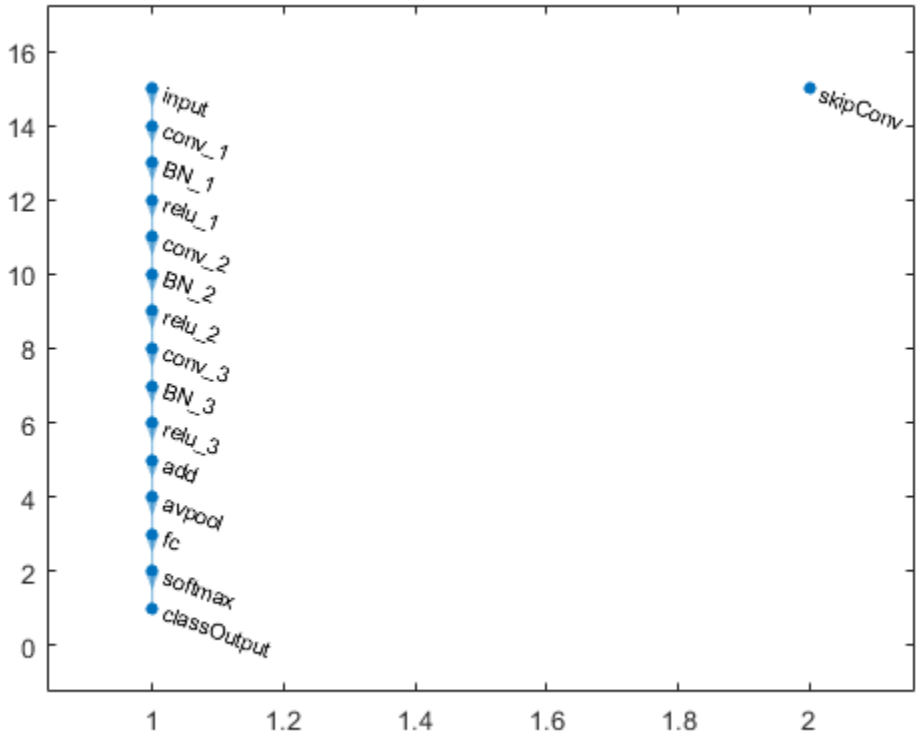
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```



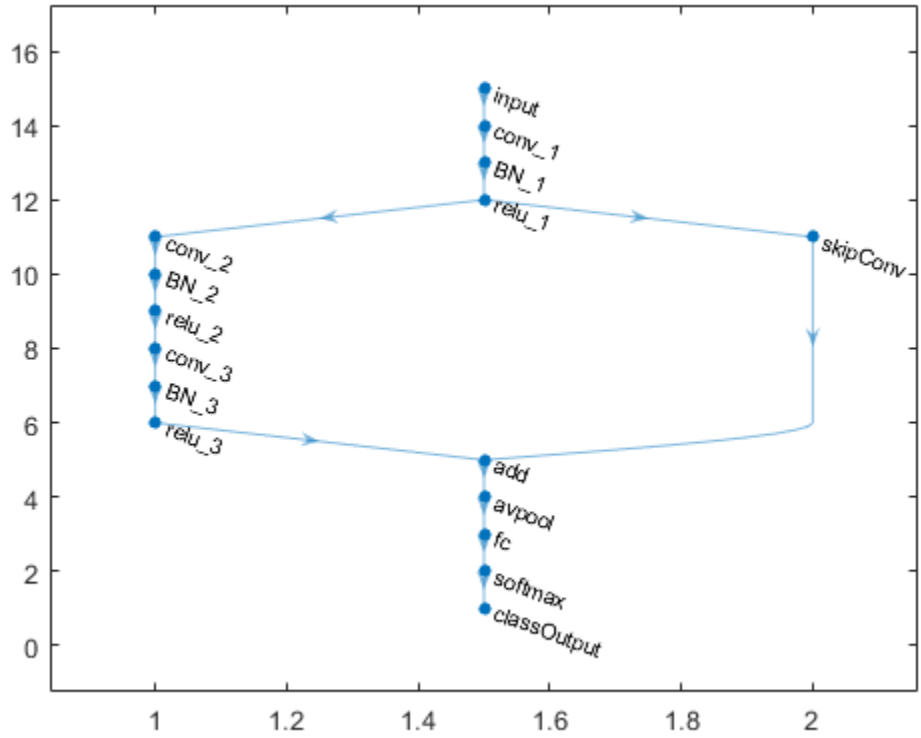
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This enables the addition layer to add the outputs of the `'skipConv'` and `'relu_3'` layers. To check that the layer has been added, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` to the `'add'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'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 correctly connected, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```



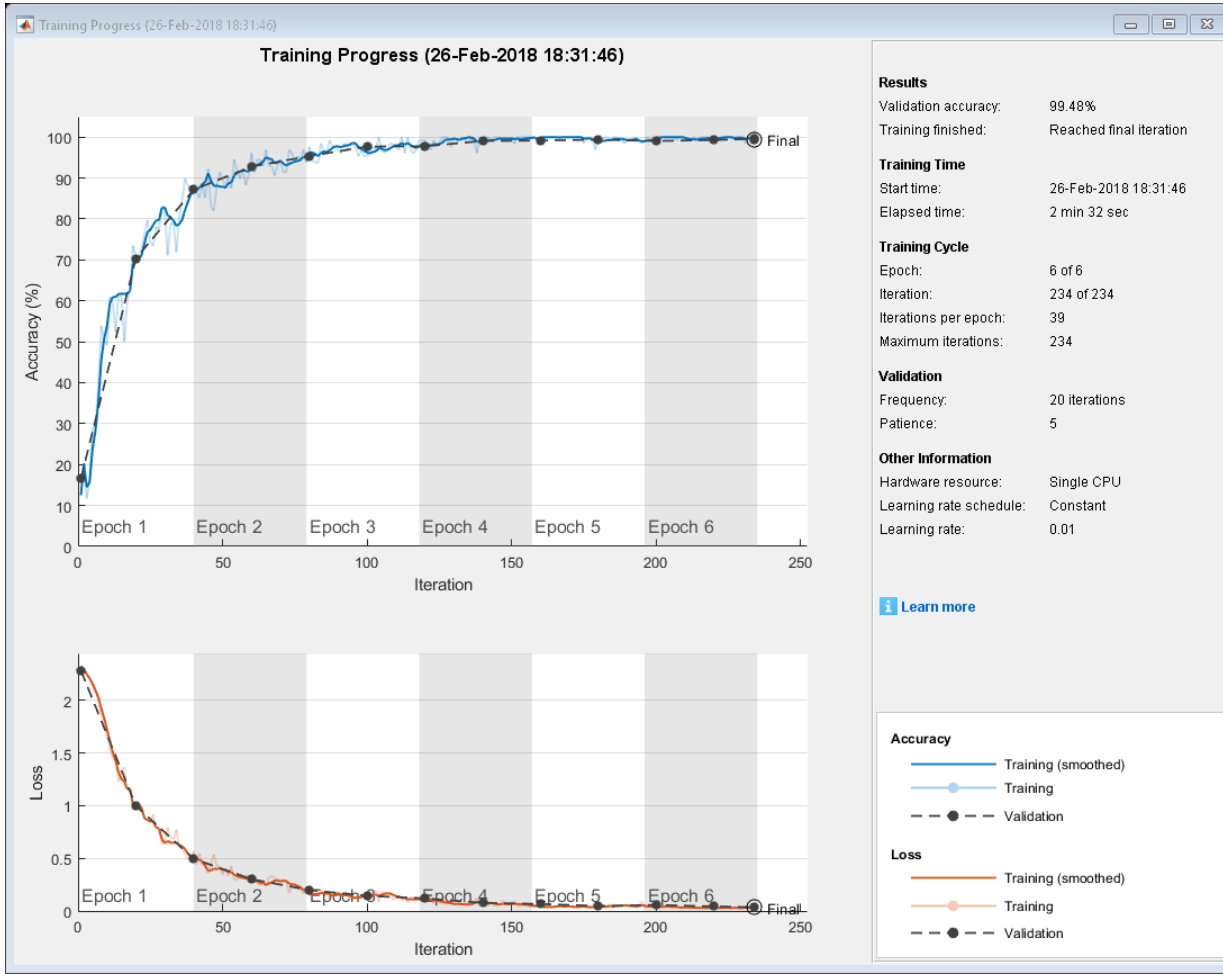
Load training and validation data, consisting of 28-by-28 grayscale images of digits.

```
[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',6,...
    'Shuffle','every-epoch',...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',20,...
    'Verbose',false,...
```

```
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



The trained 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.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9948
```

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.

Output Arguments

`newlgraph` — Output layer graph
`LayerGraph` object

Output layer graph, returned as a `LayerGraph` object.

See Also

`connectLayers` | `disconnectLayers` | `layerGraph` | `plot` | `removeLayers`

Topics

“Create and Train DAG Network for Deep Learning”
“Transfer Learning Using GoogLeNet”

Introduced in R2017b

removeLayers

Remove layers from layer graph

Syntax

```
newlgraph = removeLayers(lgraph,layerNames)
```

Description

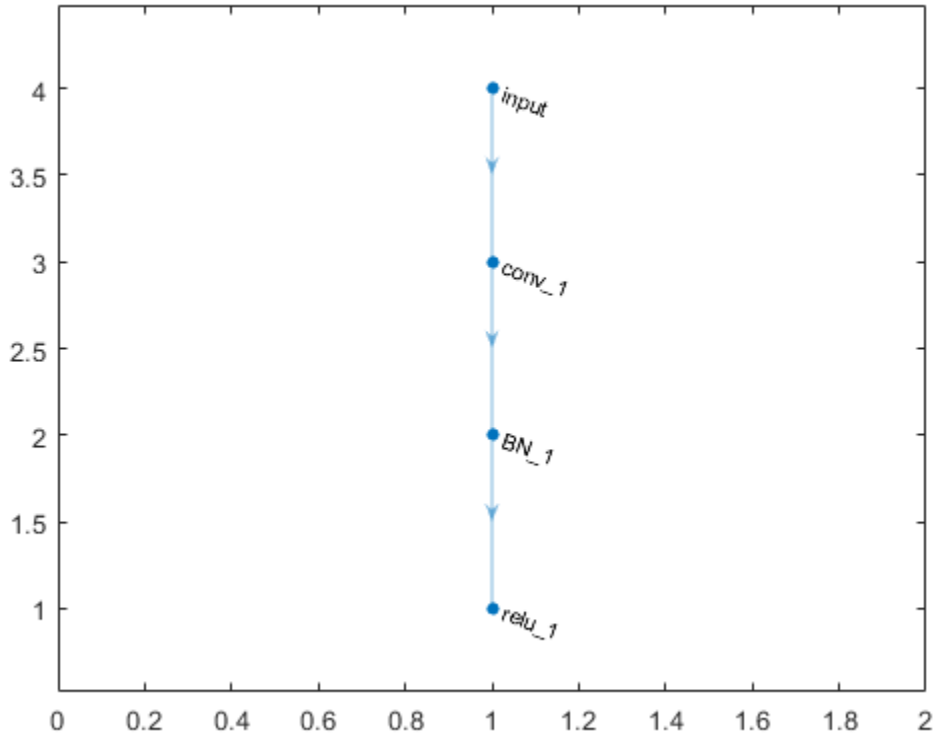
`newlgraph = removeLayers(lgraph,layerNames)` removes the layers specified by `layerNames` from the layer graph `lgraph`. Any connections to the removed layers are also removed.

Examples

Remove Layer from Layer Graph

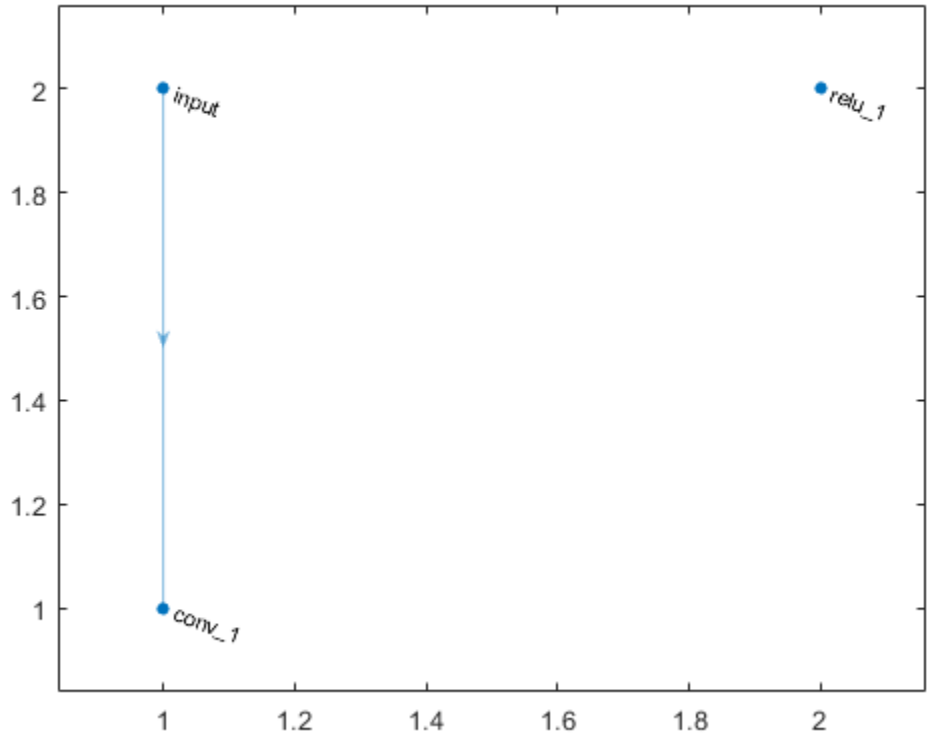
Create a layer graph from an array of layers.

```
layers = [
    imageInputLayer([28 28 1],'Name','input')
    convolution2dLayer(3,16,'Padding','same','Name','conv_1')
    batchNormalizationLayer('Name','BN_1')
    reluLayer('Name','relu_1')];

lgraph = layerGraph(layers);
figure
plot(lgraph)
```



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

Names of layers to remove, specified as a character vector or a cell array of character vectors.

To remove a single layer from the layer graph, specify the name of the layer as a character vector.

To remove multiple layers, specify the layer names as a cell array of character vectors, where each element of the cell 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` | `connectLayers` | `disconnectLayers` | `layerGraph` | `plot`

Topics

“Create and Train DAG Network for Deep Learning”
“Transfer Learning Using GoogLeNet”

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`, but includes the new connection.

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.

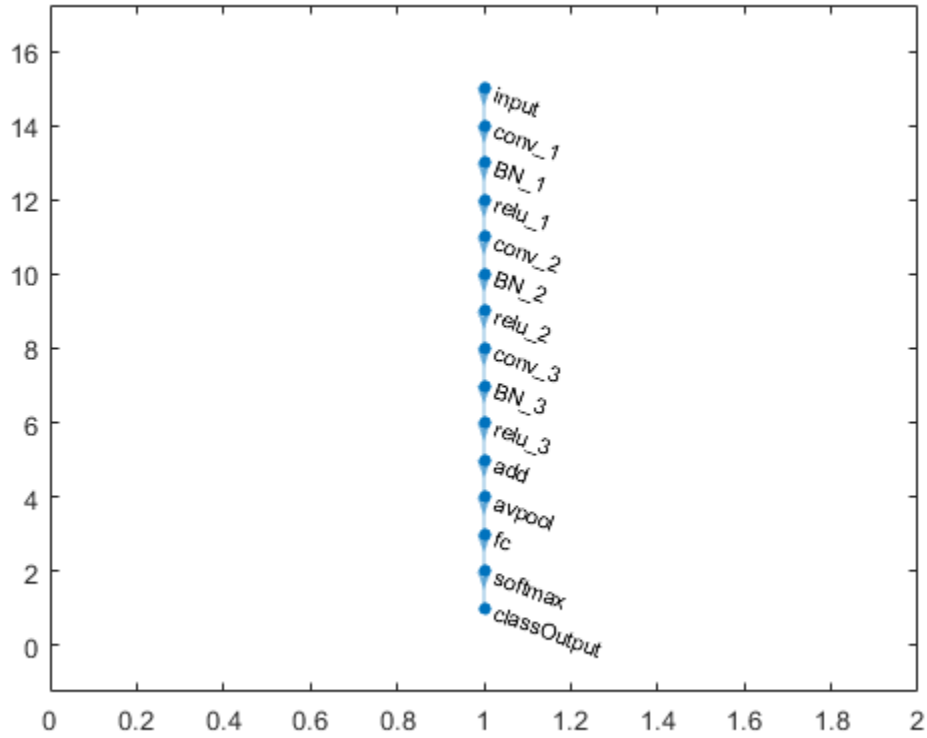
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs that the addition layer should sum. All layers must have names and all names must be unique.

```
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')];
```

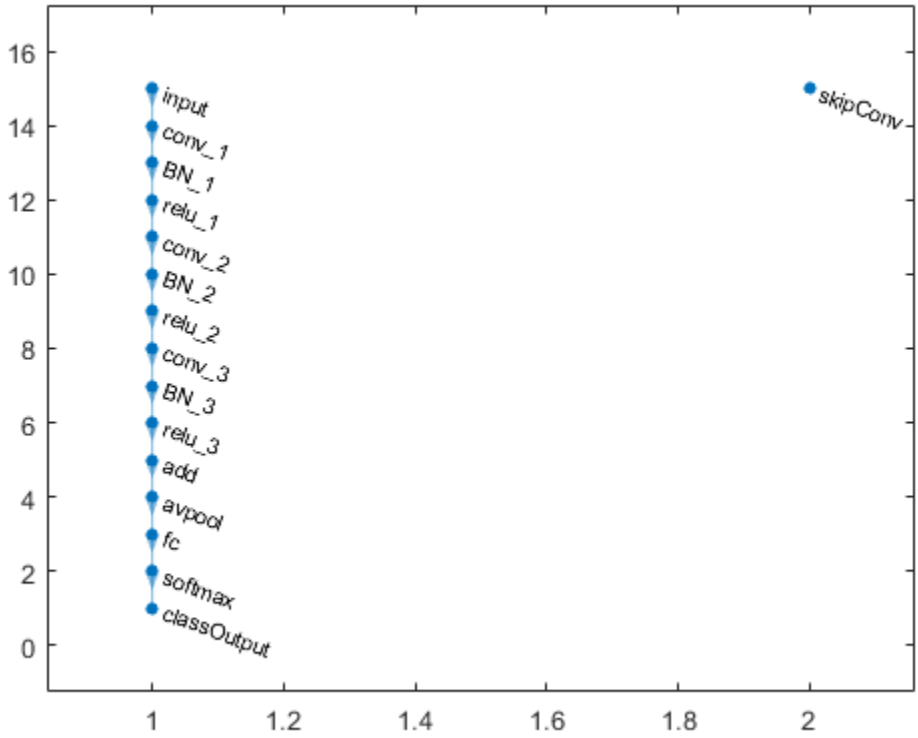
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```



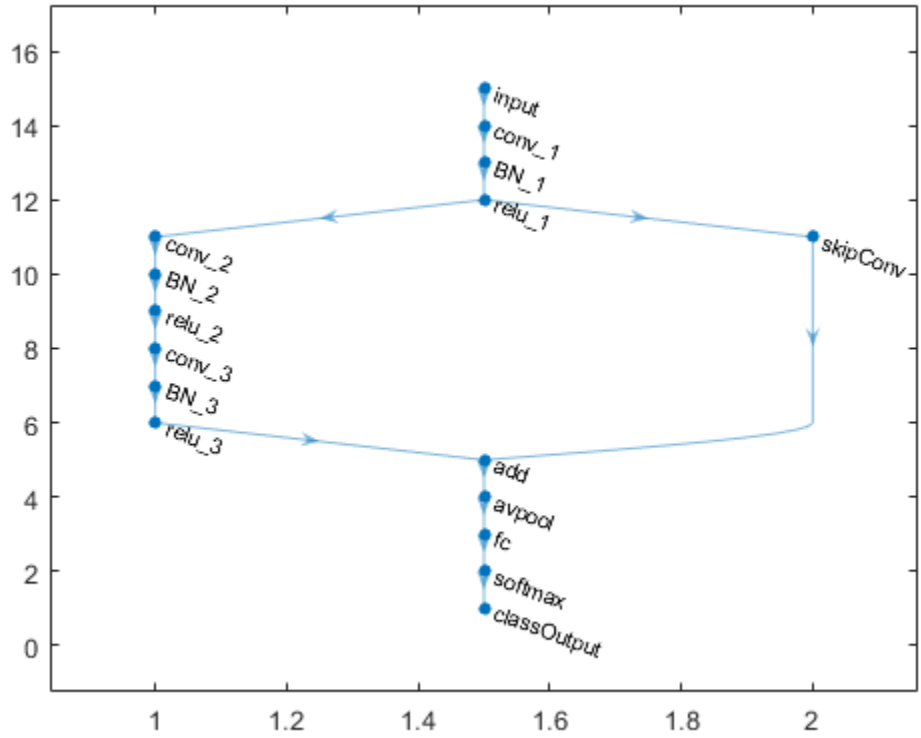
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This enables the addition layer to add the outputs of the `'skipConv'` and `'relu_3'` layers. To check that the layer has been added, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` to the `'add'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'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 correctly connected, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```



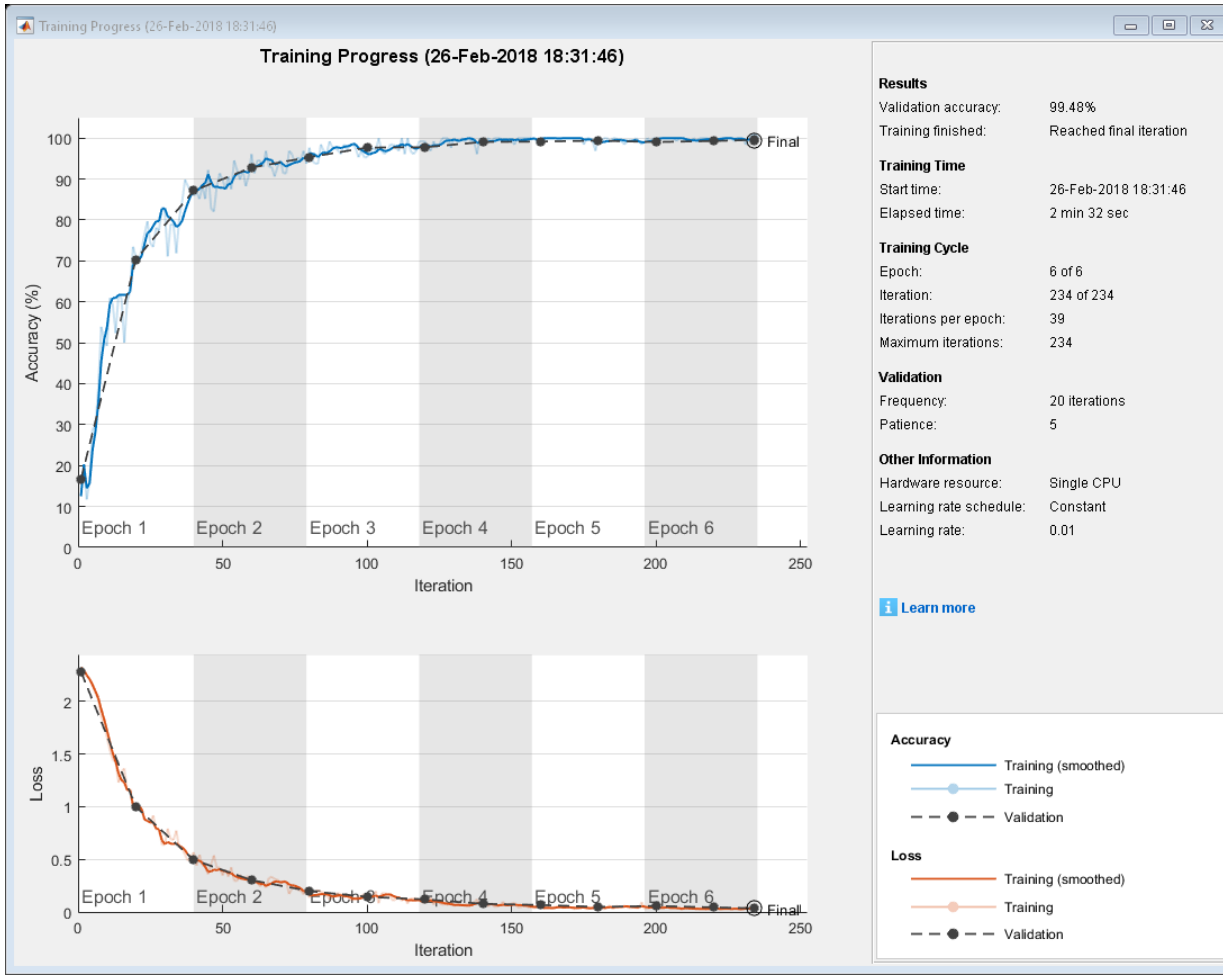
Load training and validation data, consisting of 28-by-28 grayscale images of digits.

```
[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',6,...
    'Shuffle','every-epoch',...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',20,...
    'Verbose',false,...
```

```
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



The trained 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.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9948
```

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

Connection source, specified as a character vector.

- 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 "/", followed by the name of the layer output. That is, `s` is on the form `'layerName/outputName'`.

Example: `'conv1'`

Example: `'mpool/indices'`

`d` — Connection destination

character vector

Connection destination, specified as a character vector.

- 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 "/", followed by the name of the layer output. That is, `d` is on the form `'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` | `disconnectLayers` | `layerGraph` | `plot` | `removeLayers`

Topics

"Create and Train DAG Network for Deep Learning"
"Transfer Learning Using GoogLeNet"

Introduced in R2017b

disconnectLayers

Disconnect layers in layer graph

Syntax

```
newlgraph = disconnectLayers(lgraph,s,d)
```

Description

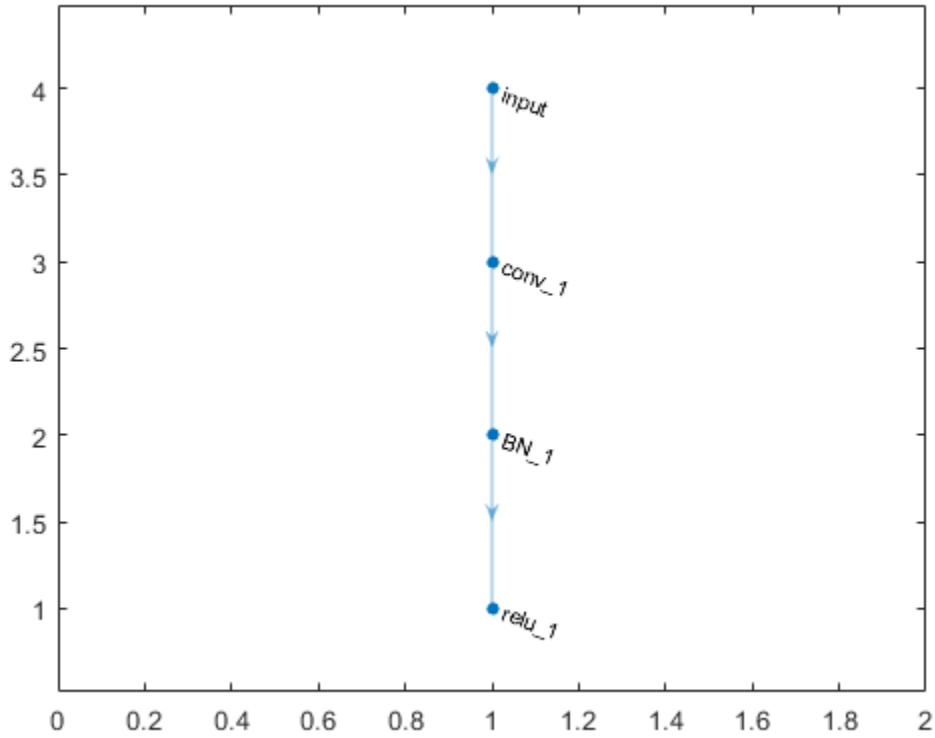
`newlgraph = disconnectLayers(lgraph,s,d)` disconnects the source layer `s` from the destination layer `d` in the layer graph `lgraph`. The new layer graph, `newlgraph`, contains the same layers as `lgraph`, but excludes the connection between `s` and `d`.

Examples

Disconnect Layers in Layer Graph

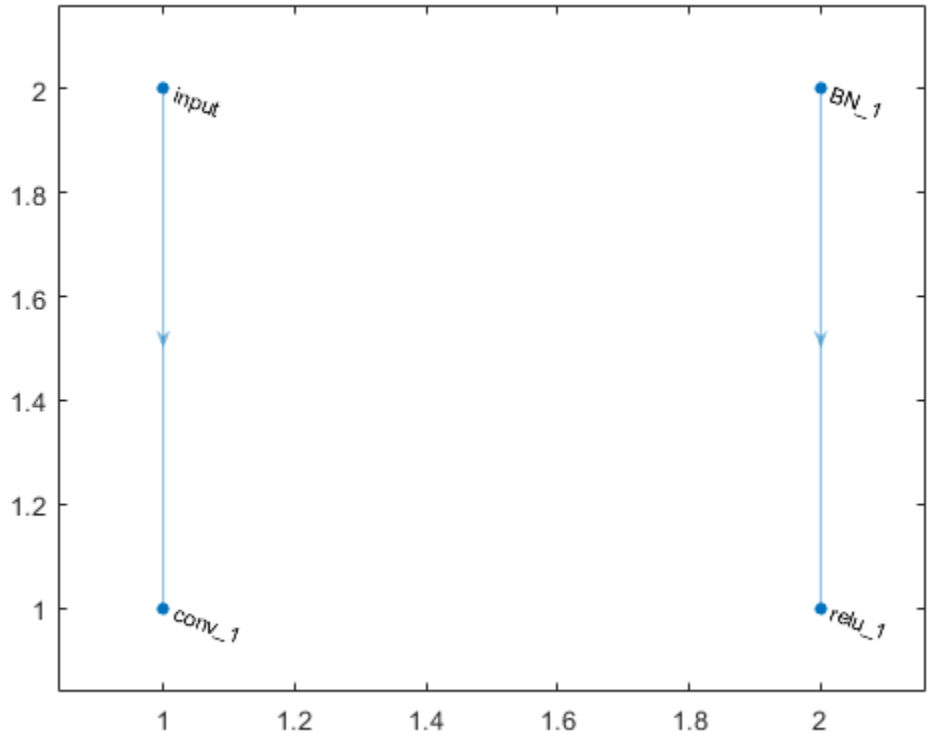
Create a layer graph from an array of layers.

```
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.

```
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

Connection source, specified as a character vector.

- 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 "/", followed by the name of the layer output. That is, `s` is on the form `'layerName/outputName'`.

Example: `'conv1'`

Example: `'mpool/indices'`

`d` — Connection destination
character vector

Connection destination, specified as a character vector.

- 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 "/", followed by the name of the layer output. That is, `d` is on the form `'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` | `connectLayers` | `layerGraph` | `plot` | `removeLayers`

Topics

"Create and Train DAG Network for Deep Learning"
"Transfer Learning Using GoogLeNet"

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`. `plot` labels each layer by its name and displays all layer connections.

`plot(dagNet)` plots a diagram of the layers in the DAG network `dagNet`.

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.

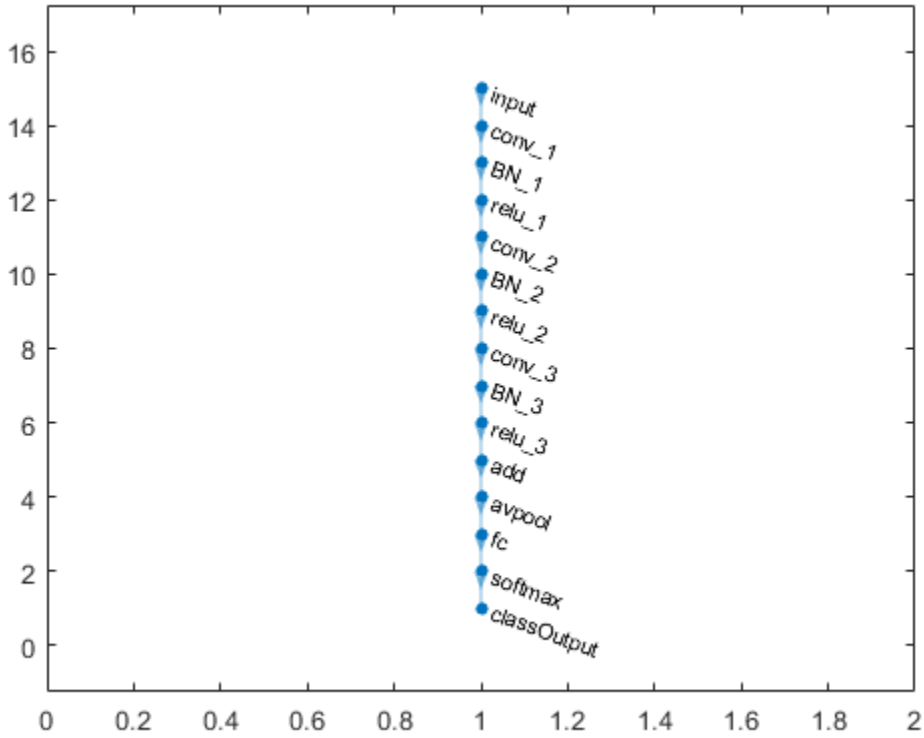
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs that the addition layer should sum. All layers must have names and all names must be unique.

```
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')];
```

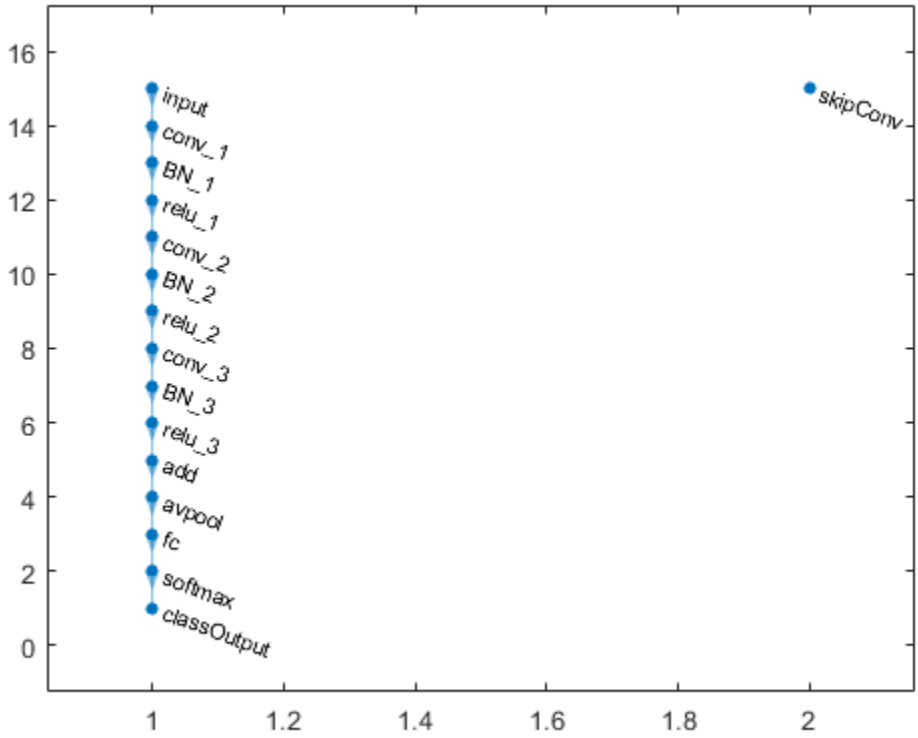
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```



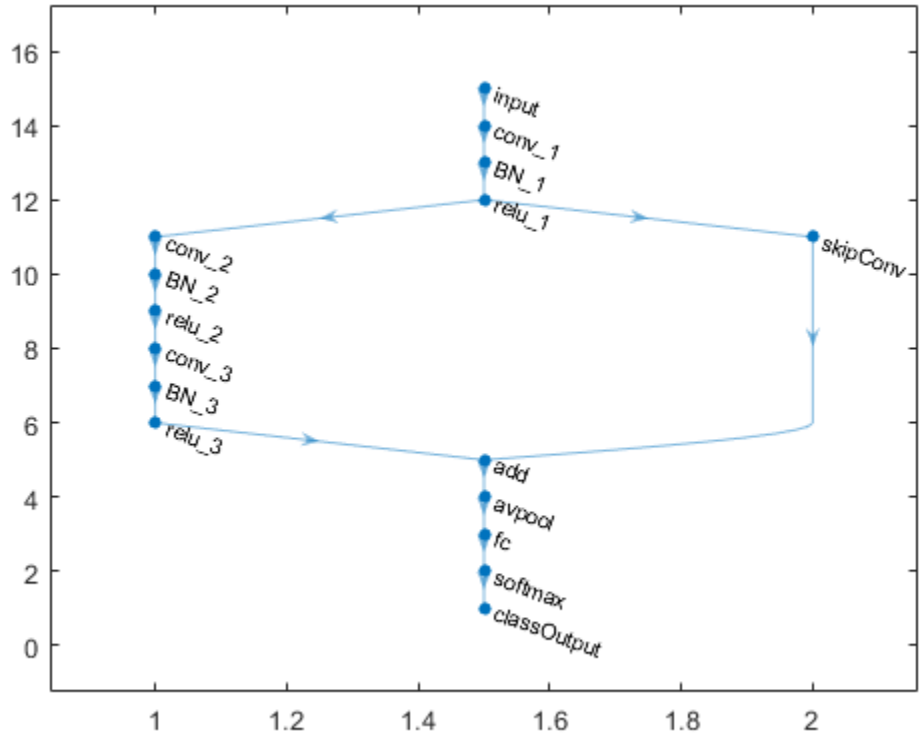
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This enables the addition layer to add the outputs of the `'skipConv'` and `'relu_3'` layers. To check that the layer has been added, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` to the `'add'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'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 correctly connected, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```



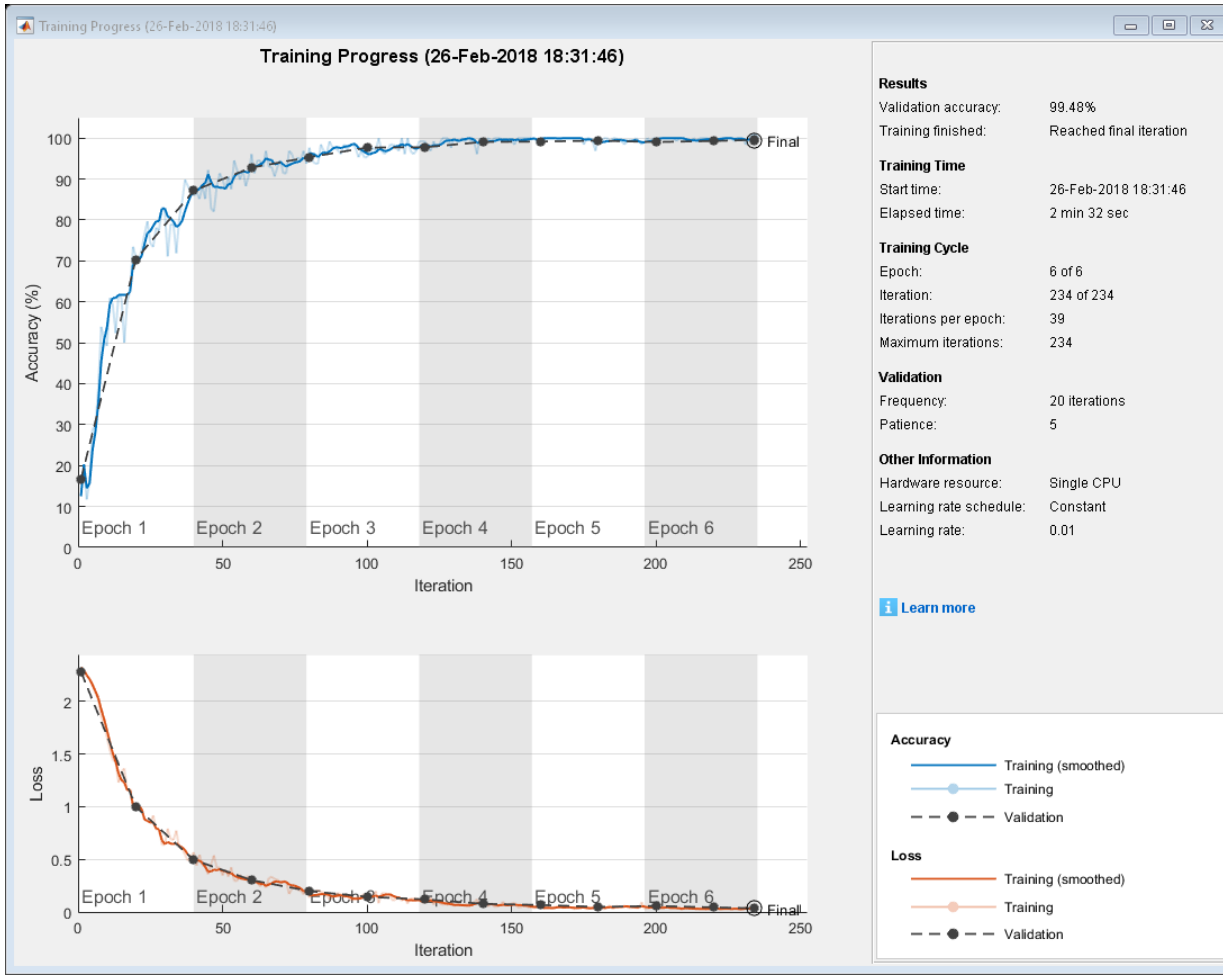
Load training and validation data, consisting of 28-by-28 grayscale images of digits.

```
[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',6,...
    'Shuffle','every-epoch',...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',20,...
    'Verbose',false,...
```

```
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



The trained 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.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9948
```

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` | `connectLayers` | `disconnectLayers` | `layerGraph` | `removeLayers`

Topics

“Create and Train DAG Network for Deep Learning”
“Transfer Learning Using GoogLeNet”

Introduced in R2017b

layerGraph

Graph of network layers for deep learning

Description

A layer graph describes the architecture of a directed acyclic graph (DAG) network for deep learning. After you create a `LayerGraph` object, you can use object functions to add layers to a graph, connect and disconnect layers in a graph, remove layers from a graph, and plot the graph. To train the network, use the layer graph as the layers on page 1-0 input argument to `trainNetwork`.

Creation

Syntax

```
lgraph = layerGraph
lgraph = layerGraph(larray)
lgraph = layerGraph(dagNet)
```

Description

`lgraph = layerGraph` creates an empty layer graph.

`lgraph = layerGraph(larray)` creates a layer graph from the array of network layers `larray`. The layers in `lgraph` are connected one after the other, in the same order as in `larray`. All layers must have unique nonempty names.

`lgraph = layerGraph(dagNet)` extracts the layer graph of the DAG network `dagNet`. For example, you can extract the layer graph of a pretrained network to perform transfer learning.

Input Arguments

`larray` — Network layers
`Layer` array

Network layers, specified as a `Layer` array.

`dagNet` — DAG network
`DAGNetwork` object

DAG network, specified as a `DAGNetwork` object.

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 input or output of the layer.

Data Types: `table`

Object Functions

addLayers	Add layers to layer graph
removeLayers	Remove layers from layer graph
connectLayers	Connect layers in layer graph
disconnectLayers	Disconnect layers in layer graph
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:

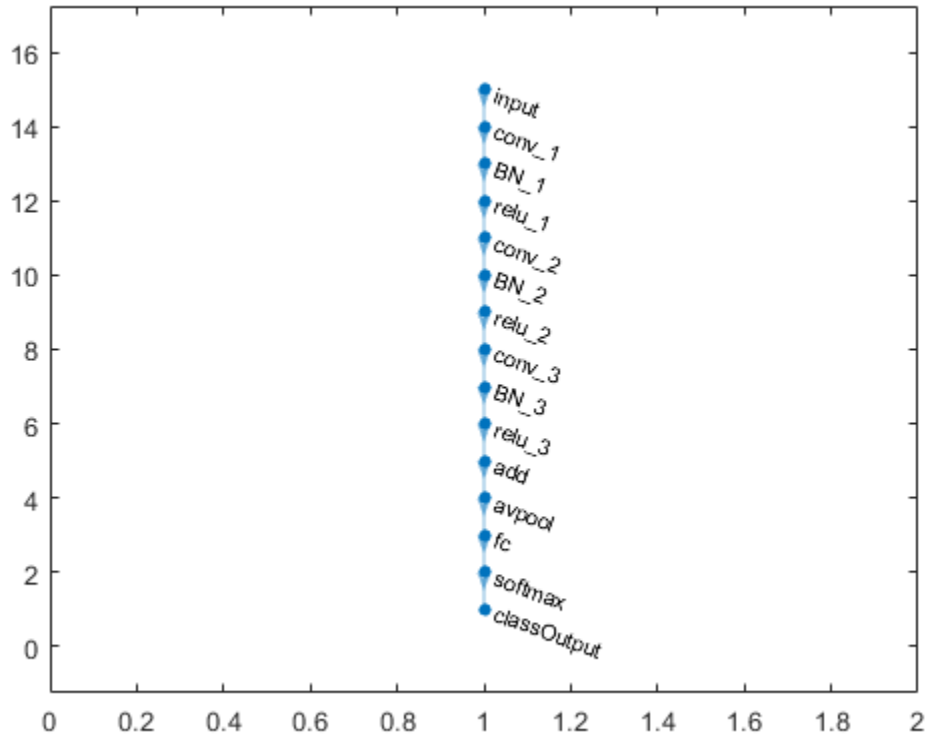
- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs that the addition layer should 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')];
```

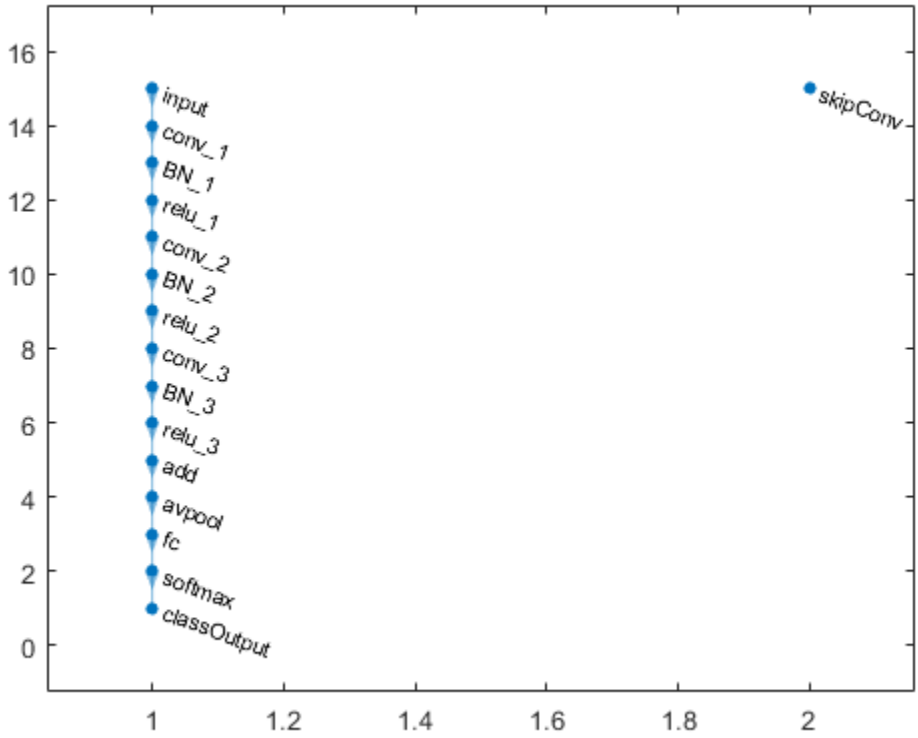
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```



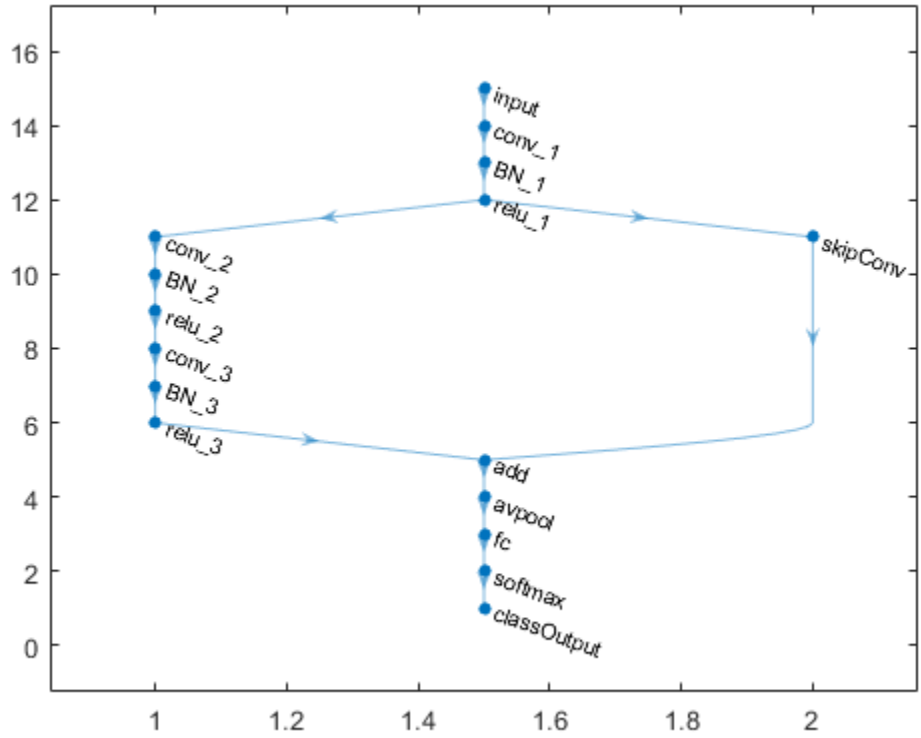
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This enables the addition layer to add the outputs of the `'skipConv'` and `'relu_3'` layers. To check that the layer has been added, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` to the `'add'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'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 correctly connected, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```



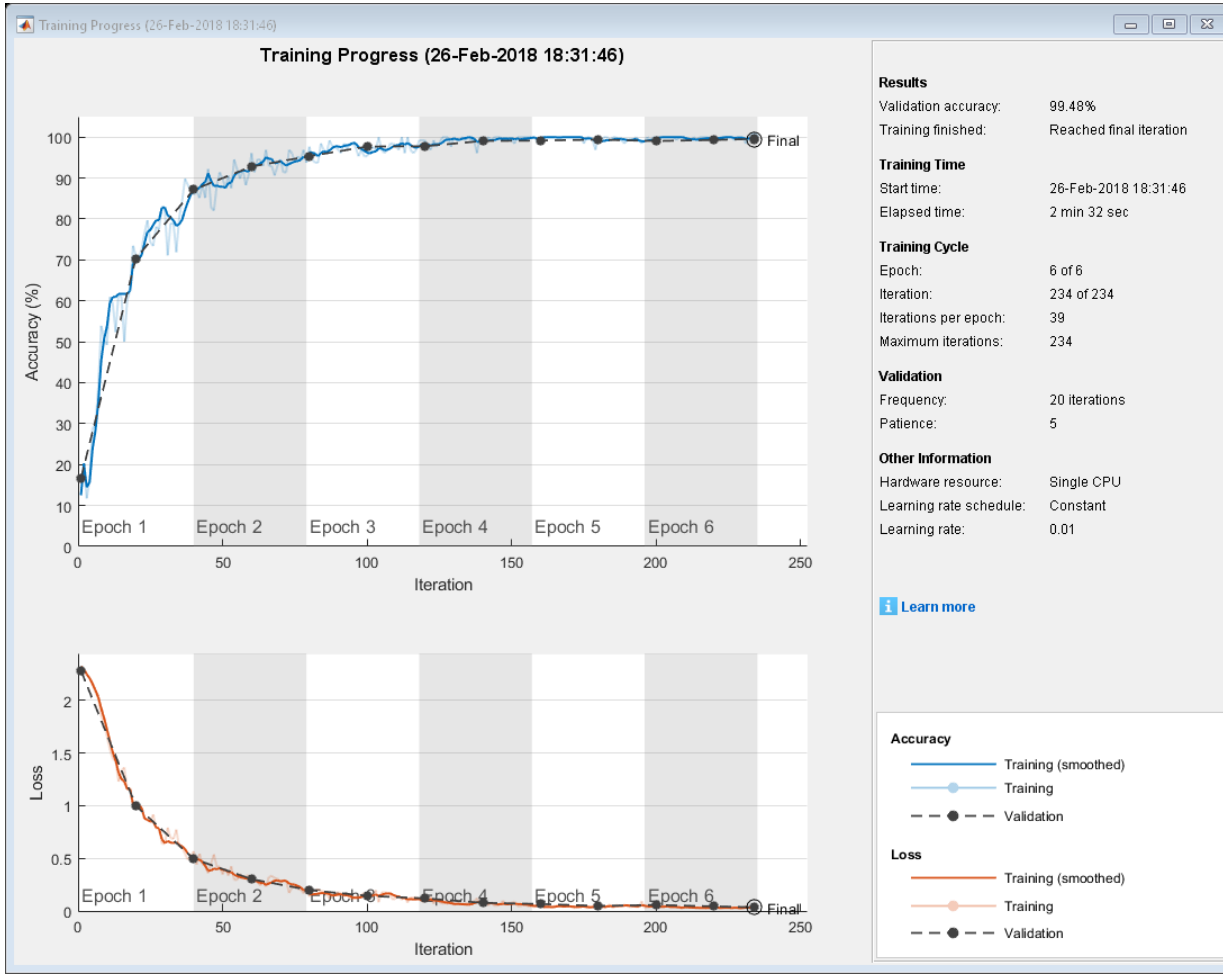
Load training and validation data, consisting of 28-by-28 grayscale images of digits.

```
[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',6,...
    'Shuffle','every-epoch',...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',20,...
    'Verbose',false,...
```

```
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



The trained 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.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9948
```

- “Create Simple Deep Learning Network for Classification”

Tips

- Layer graphs cannot describe 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` | `connectLayers` | `depthConcatenationLayer` | `disconnectLayers` | `googlenet` | `inceptionv3` | `plot` | `removeLayers` | `resnet50` | `trainNetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Create and Train DAG Network for Deep Learning”
“Transfer Learning Using GoogLeNet”

Introduced in R2017b

DAGNetwork

Directed acyclic graph (DAG) network for deep learning

Description

A DAG network is a neural network for deep learning that can have its layers arranged as a directed acyclic graph. DAG networks can have a more complex architecture where layers can have inputs from, or outputs to, multiple layers. A `DAGNetwork` object has a single input layer and a single output layer.

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 1-1079.
- Import a pretrained network from Keras using `importKerasNetwork`. For an example, see “Import and Plot Keras Network” on page 1-1092.
- Train or fine-tune a network using `trainNetwork`. For an example, see “Transfer Learning Using GoogLeNet”.

Note To learn about other pretrained networks, see “Pretrained Convolutional 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 input or output of the layer.

Data Types: `table`

Object Functions

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.

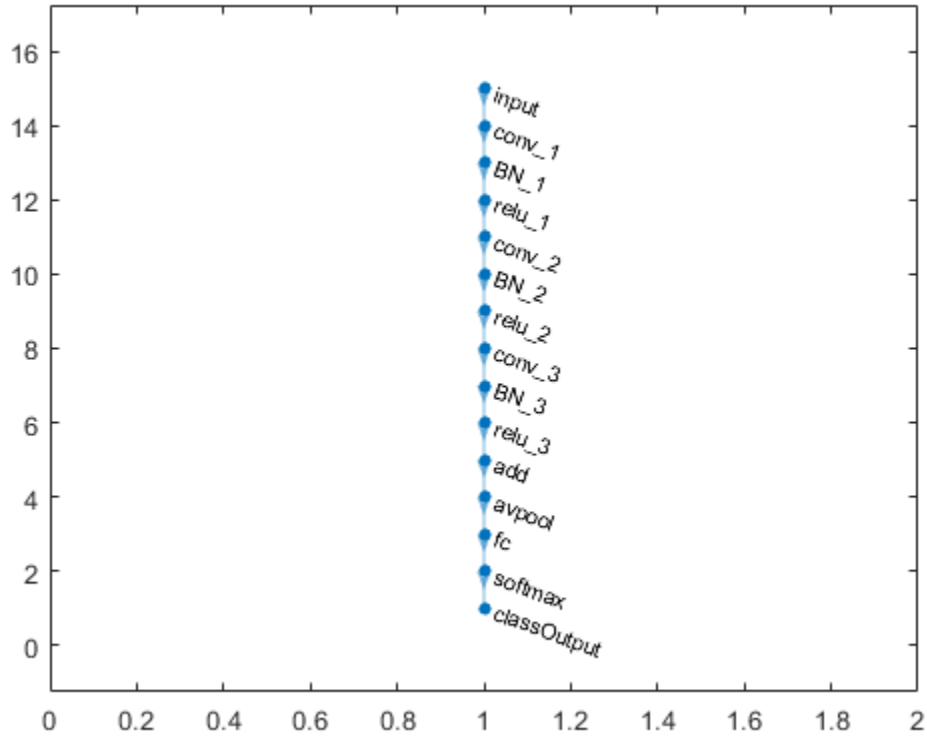
Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs that the addition layer should sum. All layers must have names and all names must be unique.

```
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')];
```

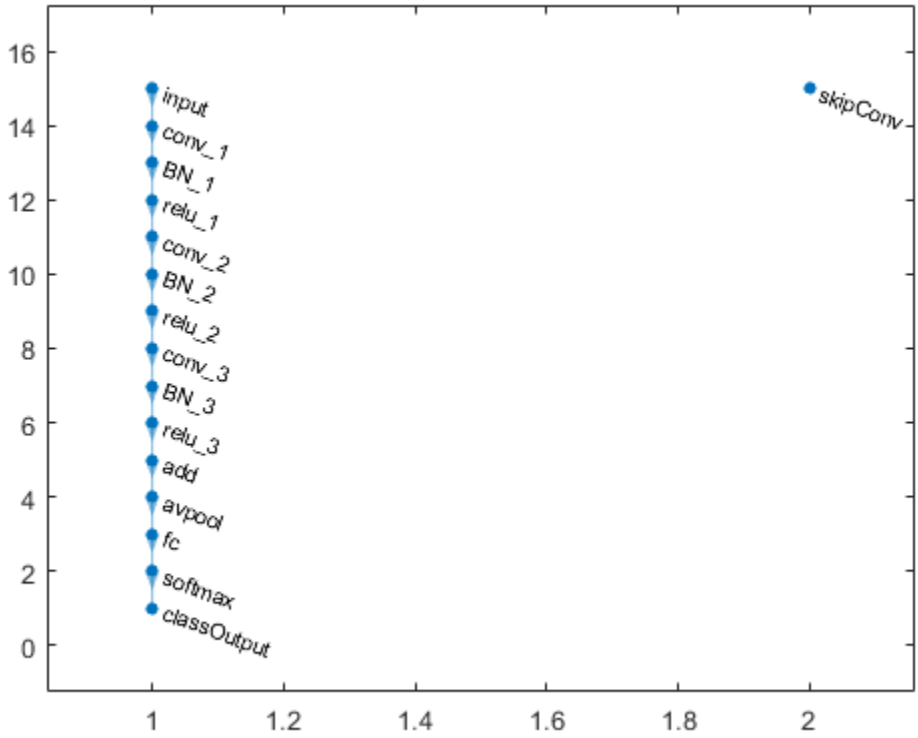
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```



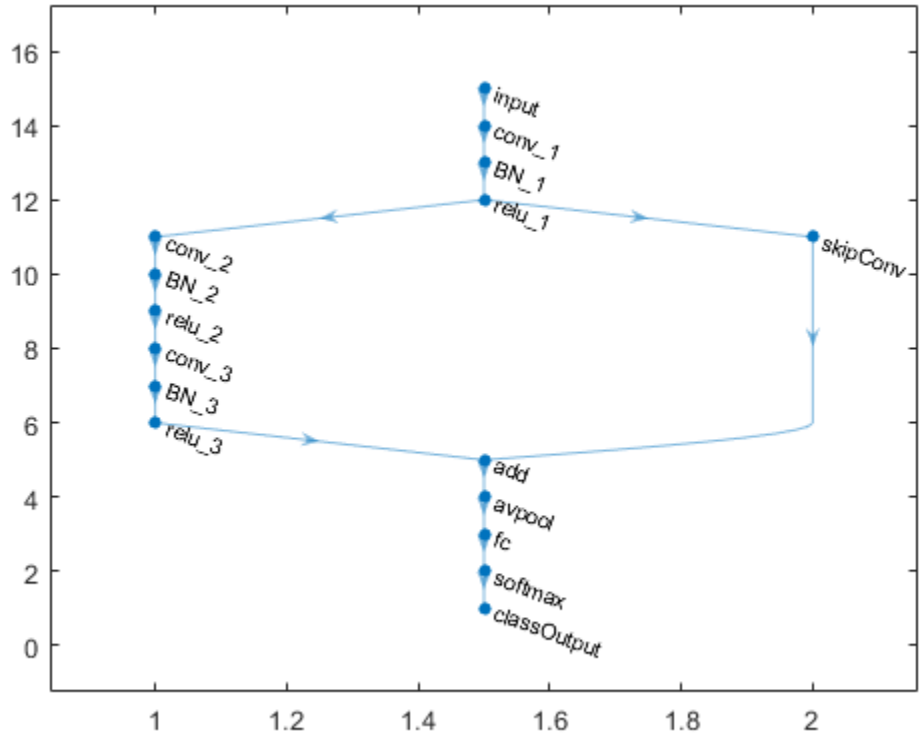
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This enables the addition layer to add the outputs of the `'skipConv'` and `'relu_3'` layers. To check that the layer has been added, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` to the `'add'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'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 correctly connected, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```



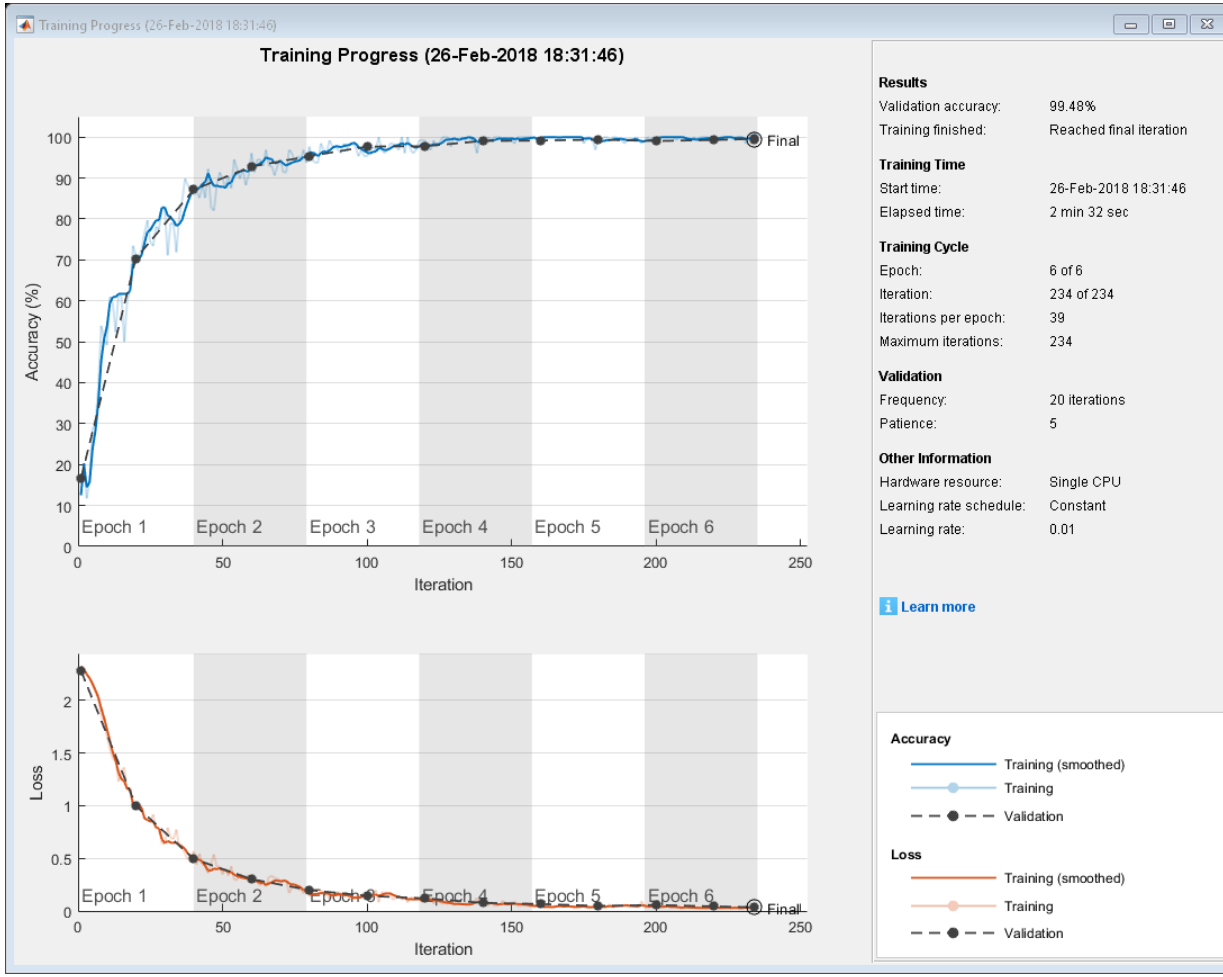
Load training and validation data, consisting of 28-by-28 grayscale images of digits.

```
[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',6,...
    'Shuffle','every-epoch',...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',20,...
    'Verbose',false,...
```

```
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



The trained 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.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9948
```

See Also

`SeriesNetwork` | `classify` | `googlenet` | `importKerasNetwork` | `inceptionv3` | `layerGraph` | `plot` | `predict` | `resnet50` | `trainNetwork` | `trainingOptions`

Topics

“Deep Learning in MATLAB”
“Classify Image Using GoogLeNet”
“Create and Train DAG Network for Deep Learning”
“Transfer Learning Using GoogLeNet”

Introduced in R2017b

additionLayer

Addition layer

Description

An addition layer adds multiple inputs element-wise. Specify the number of inputs to the layer when you create it. The inputs have names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer to other layers 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 with number of inputs equal to `numInputs`.

`layer = additionLayer(numInputs,'Name',Name)` sets the Name property. To create a network containing an addition layer you must specify a layer name.

Properties

`Name` — Layer name
`''` (default) | character vector

Layer name, specified as a character vector.

Data Types: `char`

`NumInputs` — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer.

The inputs have names `'in1','in2',...,'inN'`, where `N` equals `NumInputs`. For example, if `NumInputs` equals 3, then the inputs have names `'in1','in2'`, and `'in3'`. Use the input names when connecting or disconnecting the layer to other layers using `connectLayers` or `disconnectLayers`.

Examples

Create Addition Layer

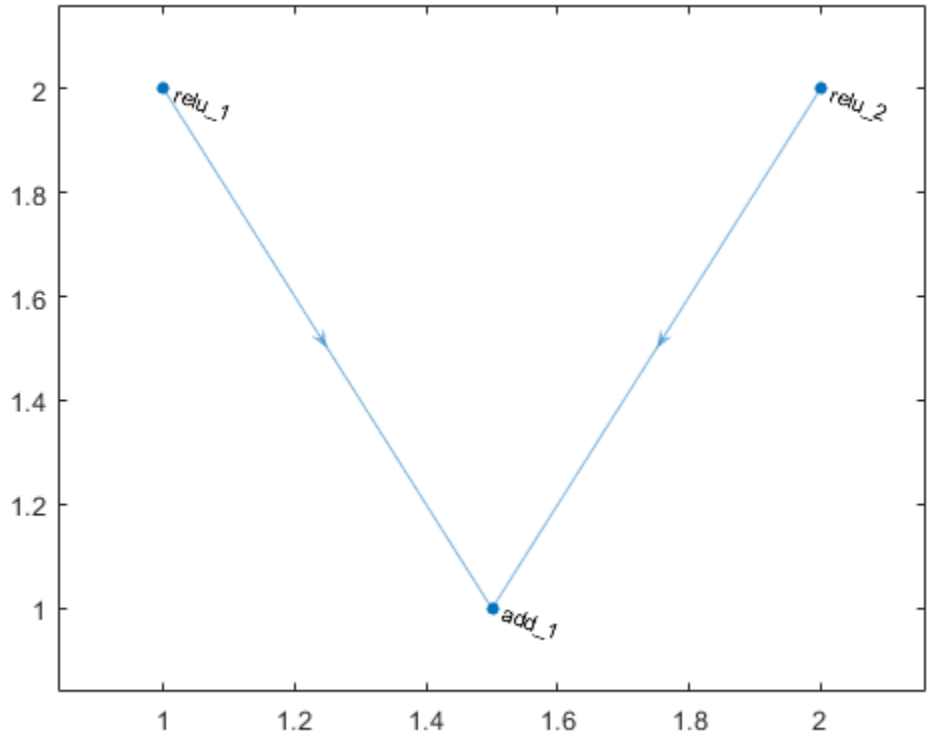
Create an addition layer with two inputs and the name `'add_1'`.

```
add = additionLayer(2,'Name','add_1')
```

```
add =
  AdditionLayer with properties:

         Name: 'add_1'
    NumInputs: 2
```

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:

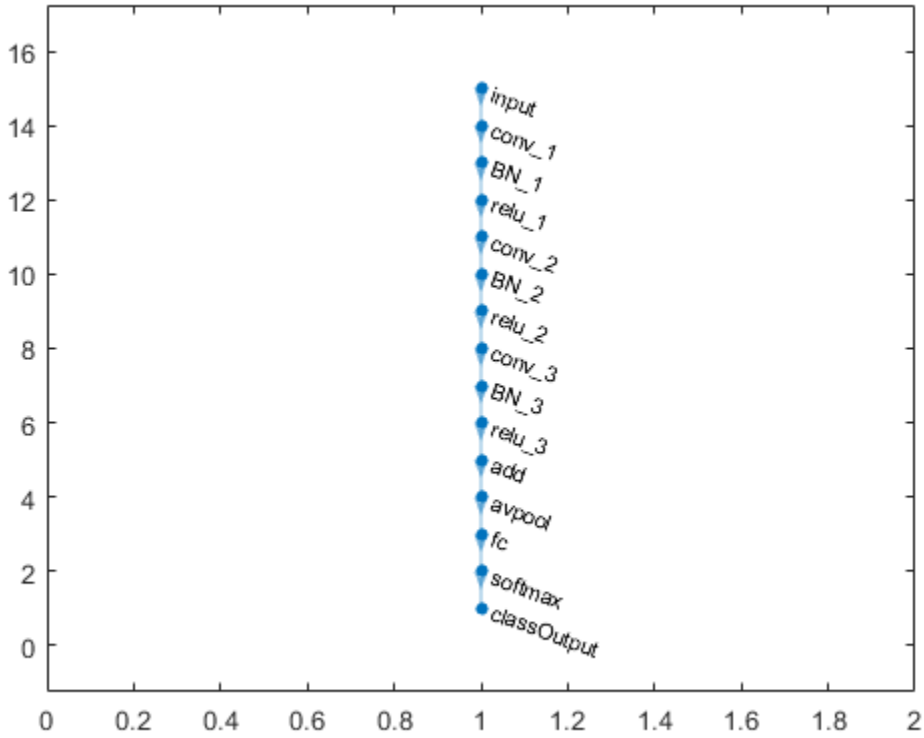
- A main branch with layers connected sequentially.
- A *shortcut connection* containing a single 1-by-1 convolutional layer. Shortcut connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network.

Create the main branch of the network as a layer array. The addition layer sums multiple inputs element-wise. Specify the number of inputs that the addition layer should 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')];
```

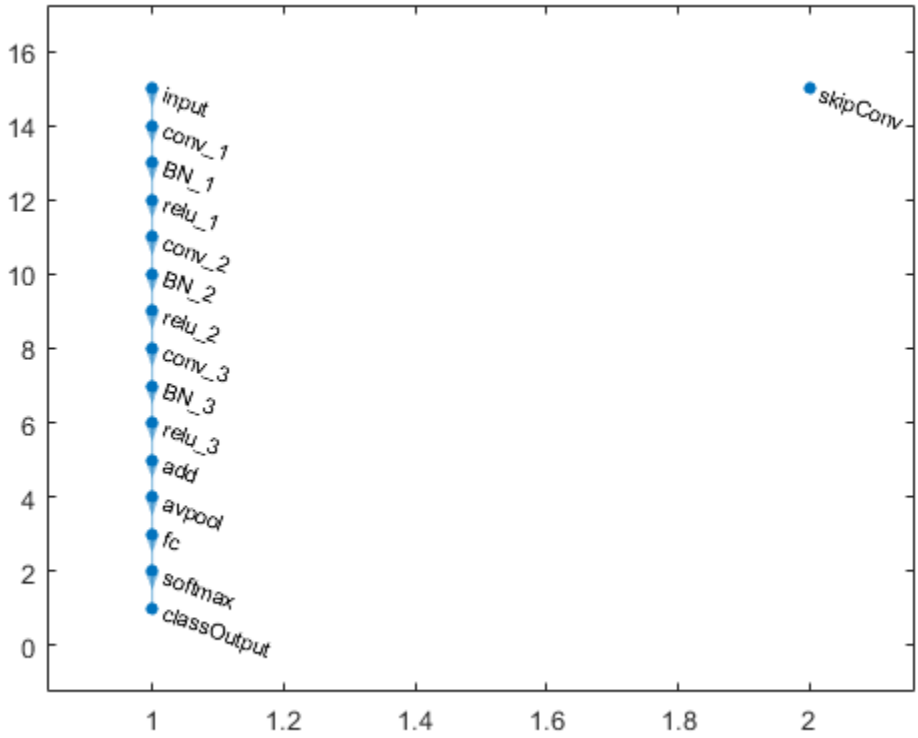
Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure
plot(lgraph)
```



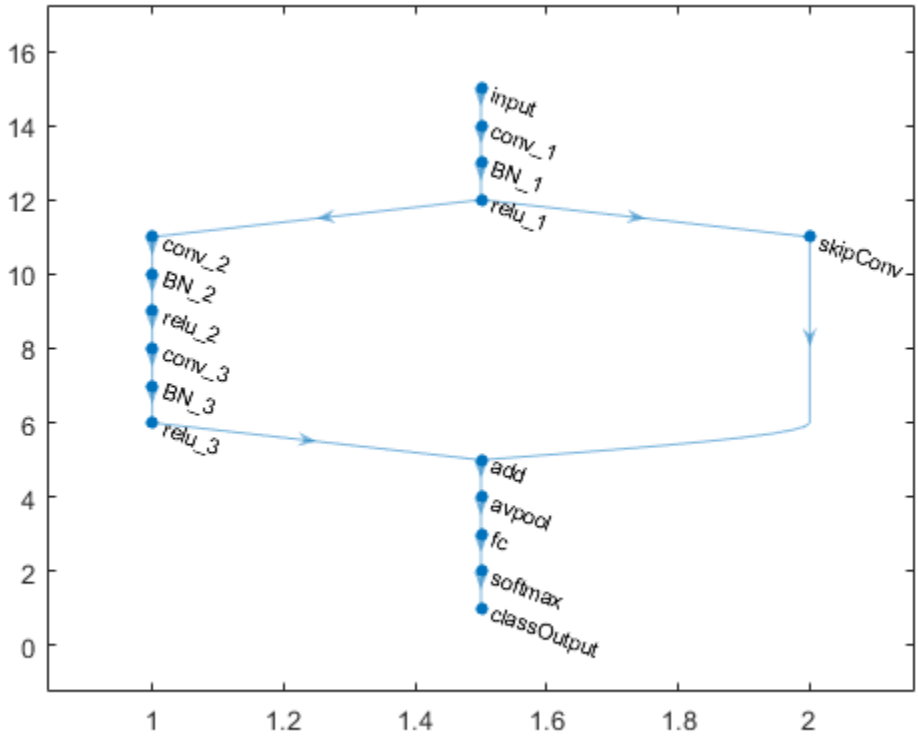
Create the 1-by-1 convolutional layer and add it to the layer graph. Specify the number of convolutional filters and the stride so that the activation size matches the activation size of the `'relu_3'` layer. This enables the addition layer to add the outputs of the `'skipConv'` and `'relu_3'` layers. To check that the layer has been added, plot the layer graph.

```
skipConv = convolution2dLayer(1,32,'Stride',2,'Name','skipConv');
lgraph = addLayers(lgraph,skipConv);
figure
plot(lgraph)
```



Create the shortcut connection from the `'relu_1'` to the `'add'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'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 correctly connected, plot the layer graph.

```
lgraph = connectLayers(lgraph,'relu_1','skipConv');
lgraph = connectLayers(lgraph,'skipConv','add/in2');
figure
plot(lgraph);
```



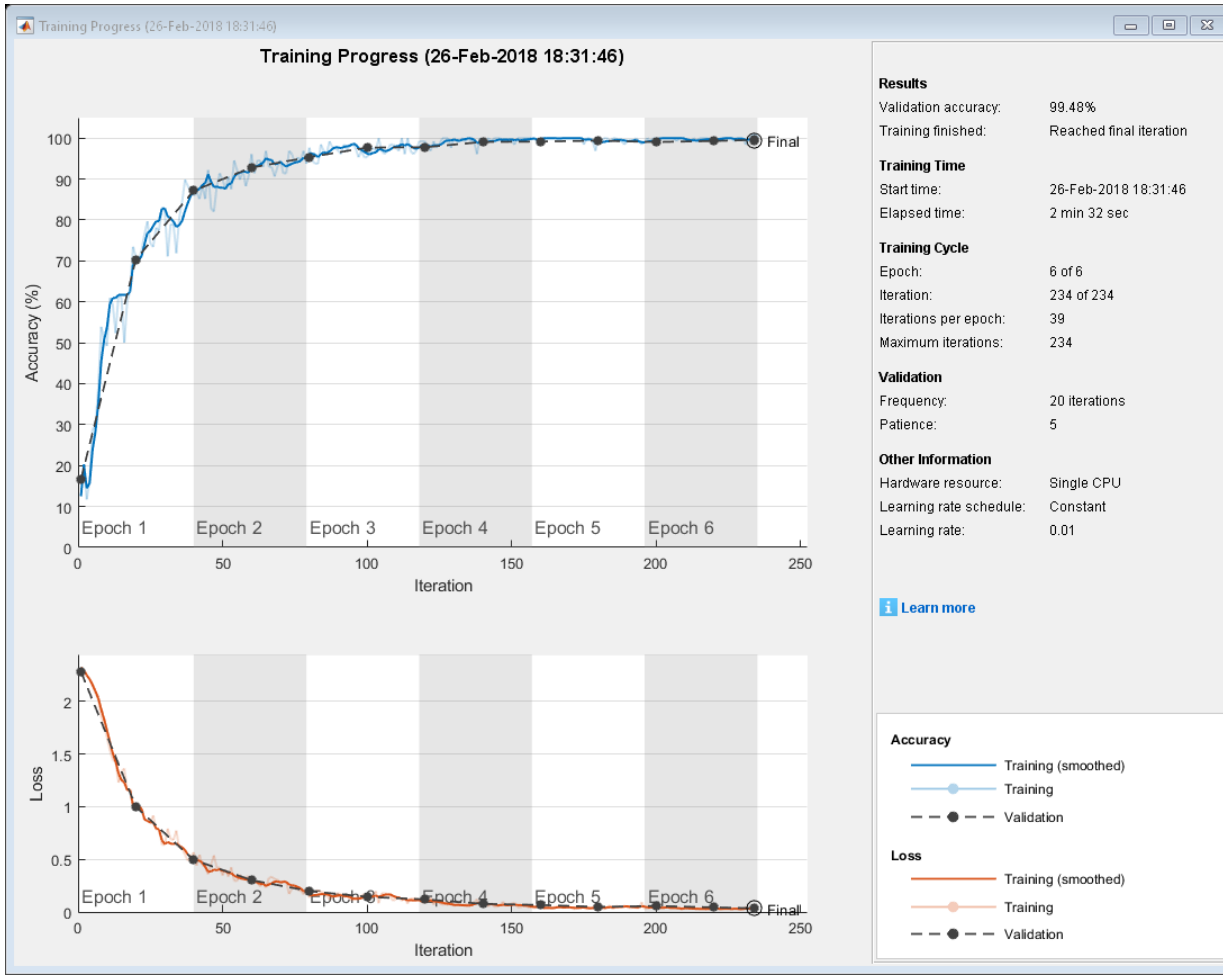
Load training and validation data, consisting of 28-by-28 grayscale images of digits.

```
[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',6,...
    'Shuffle','every-epoch',...
    'ValidationData',{XValidation,YValidation},...
    'ValidationFrequency',20,...
    'Verbose',false,...
```

```
    'Plots','training-progress');
net = trainNetwork(XTrain,YTrain,lgraph,options);
```



The trained 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.

```
YPredicted = classify(net,XValidation);
accuracy = mean(YPredicted == YValidation)
```

```
accuracy = 0.9948
```

- “Create Simple Deep Learning Network for Classification”

See Also

`depthConcatenationLayer` | `layerGraph` | `trainNetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

depthConcatenationLayer

Depth concatenation layer

Description

A depth concatenation layer takes multiple inputs that have the same height and width and concatenates them along the third dimension (the channel dimension). The inputs have names `'in1','in2',...,'inN'`, where `N` is the number of inputs. Use the input names when connecting or disconnecting the layer to other layers 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.

`layer = depthConcatenationLayer(numInputs,'Name',Name)` sets the Name property. To create a network containing a depth concatenation layer you must specify a layer name.

Properties

Name — Layer name
`''` (default) | character vector

Layer name, specified as a character vector.

Data Types: `char`

`NumInputs` — Number of inputs
positive integer

Number of inputs to the layer, specified as a positive integer.

The inputs have names `'in1','in2',...,'inN'`, where `N` equals `NumInputs`. For example, if `NumInputs` equals 3, then the inputs have names `'in1','in2'`, and `'in3'`. Use the input names when connecting or disconnecting the layer to other layers using `connectLayers` or `disconnectLayers`.

Examples

Create Depth Concatenation Layer

Create a depth concatenation layer with two inputs and the name `'concat_1'`.

```
layer = depthConcatenationLayer(2,'Name','concat_1')
```

```
layer =
  DepthConcatenationLayer with properties:

         Name: 'concat_1'
    NumInputs: 2
```

- “Create Simple Deep Learning Network for Classification”

See Also

`additionLayer` | `connectLayers` | `disconnectLayers` | `layerGraph` | `trainNetwork`

Topics

“Create Simple Deep Learning Network for Classification”
“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”

“Set Up Parameters and Train Convolutional Neural Network”
“Specify Layers of Convolutional Neural Network”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

googlenet

Pretrained GoogLeNet convolutional neural network

Syntax

```
net = googlenet
```

Description

`net = googlenet` returns a pretrained GoogLeNet model. This model is trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The model is trained on more than a 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.

This function requires the Neural Network Toolbox Model *for GoogLeNet Network* support package. If this support package is not installed, then the function provides a download link.

Examples

Download GoogLeNet Support Package

Download and install the Neural Network Toolbox Model *for GoogLeNet Network* support package.

Type `googlenet` at the command line.

```
googlenet
```

If the Neural Network 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.

```
googlenet

ans =

  DAGNetwork with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
```

- “Classify Image Using GoogLeNet”
- “Transfer Learning Using GoogLeNet”
- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”
- “Create and Train DAG Network for Deep Learning”

Output Arguments

`net` — Pretrained GoogLeNet convolutional neural network
`DAGNetwork` object

Pretrained GoogLeNet convolutional neural network, returned as a `DAGNetwork` object.

References

[1] *ImageNet*. http://www.image-net.org

[2] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.

[3] *BVLC GoogLeNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

See Also

`DAGNetwork` | `alexnet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `inceptionv3` | `layerGraph` | `plot` | `resnet50` | `trainNetwork` | `vgg16` | `vgg19`

Topics

“Classify Image Using GoogLeNet”
“Transfer Learning Using GoogLeNet”
“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

resnet50

Pretrained ResNet-50 convolutional neural network

Syntax

```
net = resnet50
```

Description

`net = resnet50` returns a pretrained ResNet-50 model. This model is trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The model is trained on more than a 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.

This function requires the Neural Network Toolbox Model *for ResNet-50 Network* support package. If this support package is not installed, then the function provides a download link.

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 “Transfer Learning Using GoogLeNet”. Load the ResNet-50 model instead of GoogLeNet and change the names of the layers that you remove and connect to match the names of the ResNet-50 layers: remove the `'ClassificationLayer_fc1000'`, `'fc1000_softmax'`, and `'fc1000'`layers, and connect to the `'avg_pool'` layer.

Examples

Download ResNet-50 Support Package

Download and install the Neural Network Toolbox Model *for ResNet-50 Network* support package.

Type `resnet50` at the command line.

```
resnet50
```

If the Neural Network 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: [177×1 nnet.cnn.layer.Layer]
    Connections: [192×2 table]
```

- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”
- “Transfer Learning Using GoogLeNet”
- “Create and Train DAG Network for Deep Learning”

Output Arguments

net — Pretrained ResNet-50 convolutional neural network

`DAGNetwork` object

Pretrained ResNet-50 convolutional neural network, returned as a `DAGNetwork` object.

References

[1] *ImageNet*. http://www.image-net.org

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

See Also

`DAGNetwork` | `alexnet` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `layerGraph` | `plot` | `resnet101` | `trainNetwork` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Transfer Learning Using GoogLeNet”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

resnet101

Pretrained ResNet-101 convolutional neural network

Syntax

```
net = resnet101
```

Description

`net = resnet101` returns a pretrained ResNet-101 model. This model is trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The model is trained on more than a 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.

This function requires the Neural Network Toolbox Model *for ResNet-101 Network* support package. If this support package is not installed, then the function provides a download link.

You can use `classify` to classify new images using the ResNet-101 model. Follow the steps of “Classify Image Using GoogLeNet” and replace GoogLeNet with ResNet-101.

To retrain the network on a new classification task, follow the steps of “Transfer Learning Using GoogLeNet”. Load the ResNet-101 model instead of GoogLeNet and change the names of the layers that you remove and connect to match the names of the ResNet-101 layers: remove the `'fc1000'`, `'prob'`, and `'ClassificationLayer_predictions'` layers, and connect to the `'pool5'` layer.

Examples

Download ResNet-101 Support Package

Download and install the Neural Network Toolbox Model *for ResNet-101 Network* support package.

Type `resnet101` at the command line.

```
resnet101
```

If the Neural Network 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.

```
resnet101

ans =

  DAGNetwork with properties:

         Layers: [347×1 nnet.cnn.layer.Layer]
    Connections: [379×2 table]
```

- "Deep Learning in MATLAB"
- "Pretrained Convolutional Neural Networks"
- "Transfer Learning Using GoogLeNet"
- "Create and Train DAG Network for Deep Learning"

Output Arguments

net — Pretrained ResNet-101 convolutional neural network

`DAGNetwork` object

Pretrained ResNet-101 convolutional neural network, returned as a `DAGNetwork` object.

References

[1] *ImageNet*. http://www.image-net.org

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

See Also

`DAGNetwork` | `alexnet` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `inceptionv3` | `layerGraph` | `plot` | `resnet50` | `trainNetwork` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Transfer Learning Using GoogLeNet”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

inceptionv3

Pretrained Inception-v3 convolutional neural network

Syntax

```
net = inceptionv3
```

Description

`net = inceptionv3` returns a pretrained Inception-v3 model. This model is trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The model is trained on more than a 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.

This function requires the Neural Network Toolbox Model *for Inception-v3 Network* support package. If this support package is not installed, then the function provides a download link.

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 “Transfer Learning Using GoogLeNet”. Load the Inception-v3 model instead of GoogLeNet and change the names of the layers that you remove and connect to match the names of the Inception-v3 layers: remove the `'predictions'`, `'predictions_softmax'`, and `'ClassificationLayer_predictions'` layers, and connect to the `'avg_pool'` layer.

Examples

Download Inception-v3 Support Package

Download and install the Neural Network Toolbox Model *for Inception-v3 Network* support package.

Type `inceptionv3` at the command line.

```
inceptionv3
```

If the Neural Network 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: [316×1 nnet.cnn.layer.Layer]
    Connections: [350×2 table]
```

- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”
- “Transfer Learning Using GoogLeNet”
- “Create and Train DAG Network for Deep Learning”

Output Arguments

net — Pretrained Inception-v3 convolutional neural network

`DAGNetwork` object

Pretrained Inception-v3 convolutional neural network, returned as a `DAGNetwork` object.

References

[1] *ImageNet*. http://www.image-net.org

[2] Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818-2826. 2016.

See Also

`DAGNetwork` | `alexnet` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `importKerasNetwork` | `layerGraph` | `plot` | `resnet50` | `trainNetwork` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”
“Transfer Learning Using GoogLeNet”
“Create and Train DAG Network for Deep Learning”

Introduced in R2017b

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`.

- 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.

Note This function requires Neural Network 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 TensorFlow-Keras network architecture and weights with additional options specified by one or more of the name-value pair arguments.

For example, `importKerasNetwork(modelfile,'WeightFile',weights)` imports the network architecture from the model file `modelfile` and weights from the weight file `weights`. In this case, `modelfile` can be in the HDF5 or JSON (`.json`) format, and the weight file must be in the HDF5 format.

Examples

Download Importer *for TensorFlow-Keras Models* Support Package

Download and install Neural Network Toolbox Importer *for TensorFlow-Keras Models* support package.

You can install the support package from the Add-Ons gallery. Select **Get Add-Ons** from the **Add-Ons** drop-down menu of the MATLAB® desktop. The add-on files are in the "MathWorks Features" section. Choose Neural Network Toolbox Importer *for TensorFlow-Keras Models*. To install the support package, click the link, and then click **Install**. To learn more about finding and installing add-ons, see “Get Add-Ons” (MATLAB).

Import and Plot Keras Network

Specify file(s) to import.

```
modelfile = 'digitsDAGnet.h5';
```

This is a directed acyclic graph convolutional neural network trained on the digits data.

Import the network.

```
net = importKerasNetwork(modelfile)
```

```
Warning: Saved Keras networks do not include class names. Class names will be set to "1
N is the number of units in the softmax layer.  To specify class names, use the 'ClassN
> In nnet.internal.cnn.keras.createDAGNetwork (line 19)
  In nnet.internal.cnn.keras.importKerasNetwork (line 31)
  In importKerasNetwork (line 78)

net =

  DAGNetwork with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
```

Display the network layers in the Command Window.

```
net.Layers
```

```
ans =
```

```
  13x1 Layer array with layers:

     1   'input_1'                             Image Input             28x28x1 images
     2   'conv2d_1'                            Convolution             20 7x7x1 convolut
     3   'conv2d_1_relu'                       ReLU                    ReLU
     4   'conv2d_2'                            Convolution             20 3x3x1 convolut
     5   'conv2d_2_relu'                       ReLU                    ReLU
     6   'max_pooling2d_1'                     Max Pooling             2x2 max pooling w
     7   'max_pooling2d_2'                     Max Pooling             2x2 max pooling w
     8   'flatten_1'                           Flatten C-style         Flatten activatio
     9   'flatten_2'                           Flatten C-style         Flatten activatio
    10   'concatenate_1'                       Depth concatenation     Depth concatenati
    11   'dense_1'                             Fully Connected         10 fully connecte
    12   'activation_1_softmax'                Softmax                 softmax
    13   'ClassificationLayer_activation_1'    Classification Output   crossentropyex wi
```

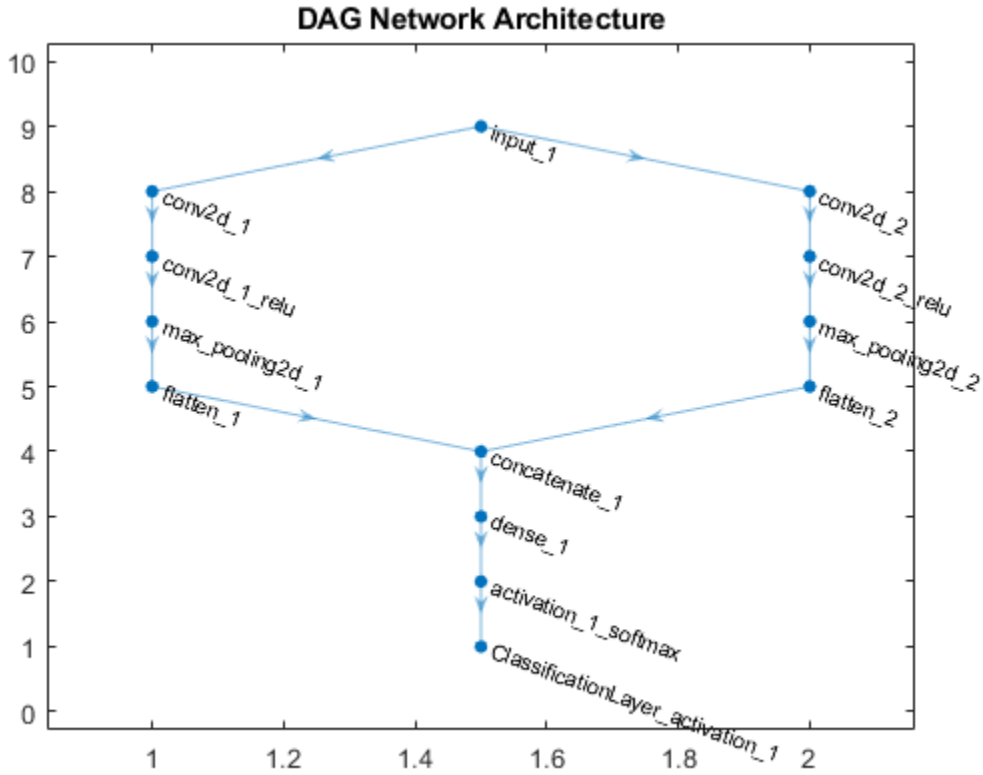
Display the network connections in the Command Window.

```
net.Connections
```

```
ans =

  13×2 table

           Source                  Destination
    ____________________    ____________________________________

    'input_1'               'conv2d_1'
    'input_1'               'conv2d_2'
    'conv2d_1'              'conv2d_1_relu'
    'conv2d_1_relu'         'max_pooling2d_1'
    'conv2d_2'              'conv2d_2_relu'
    'conv2d_2_relu'         'max_pooling2d_2'
    'max_pooling2d_1'       'flatten_1'
    'max_pooling2d_2'       'flatten_2'
    'flatten_1'             'concatenate_1/in1'
    'flatten_2'             'concatenate_1/in2'
    'concatenate_1'         'dense_1'
    'dense_1'               'activation_1_softmax'
    'activation_1_softmax'  'ClassificationLayer_activation_1'
```

Plot the network architecture.

```
if isa(net,'DAGNetwork')
    figure;
    plot(net);
    title('DAG Network Architecture');
end
```



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 class names. Class names will be set to "1
where N is the number of classes in the classification output layer of the network.  To
names, use the 'ClassNames' argument.
> In nnet.internal.cnn.keras.createDAGNetwork (line 19)
  In nnet.internal.cnn.keras.importKerasNetwork (line 31)
  In importKerasNetwork (line 84)

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,'ClassNames',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)])
```

Input Arguments

`modelfile` — Name of Keras model file

character vector

Name of the Keras model file containing the trained network architecture, and possibly the weights, specified as a character vector. 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. In this case, you must supply the weights in an HDF5 file, using the `'WeighFile'` name-value pair argument.

`modelfile` 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: `'digitsnet.h5'`

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importKerasNetwork(modelfile,'OutputLayerType','classification','ClassNames',imagelabels)` specifies `importKerasNetwork` to put an output layer for a classification problem at the end of the Keras layers and provides the class names in `imagelabels`.

`WeightFile` — Weight file name
character vector

Weight file name, specified as a character vector. `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. To import network layers without weights, see `importKerasLayers`.

Example: `'WeightFile','weights.h5'`

Data Types: `char`

`OutputLayerType` — Type of output layer
`'classification'` | `'regression'`

Type of the output layer that `importKerasNetworks` appends to the end of the imported network architecture when `modelfile` does not specify a loss function, specified as `'classification'` or `'regression'`.

Example: `'OutputLayerType','regression'`

Data Types: `char`

`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 gray 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]`

Data Types: `double`

`ClassNames` — Class names
`string(1:K)` (default) | string array | cell array of character vectors

Class names for classification problems, specified as a string array or a cell array of character vectors. If you do not provide class names, `importKerasNetwork` assigns string array of integers 1 to K as class names, where K is the number of classes.

Example: `'ClassNames',{'class1','class2','class3'}`

Data Types: `cell` | `string`

Output Arguments

`net` — Pretrained Keras network
`SeriesNetwork` object | `DAGNetwork` object

Pretrained Keras network, returned as a `SeriesNetwork` object when Keras network is of type 'Sequential', or returned as a `DAGNetwork` object when Keras network is of type 'Model'.

Tips

- `importKerasNetwork` can import a network with the following Keras layer types. 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

Keras Layer	NNET Layer Function
Add	`additionLayer`

Activation, with activation names:
• relu
• linear
• softmax
• sigmoid
• tanh
	`reluLayer`
	`softmaxLayer`
	`nnet.keras.layer.SigmoidLayer`
	`nnet.keras.layer.TanhLayer`

AveragePooling2D	`averagePooling2dLayer`
BatchNormalization	`BatchNormalizationLayer`
Concatenate	`depthConcatenationLayer`
Conv2D	`convolution2dLayer`
Conv2DTranspose	`transposedConv2dLayer`
Dense	`fullyConnectedLayer`
Dropout	`dropoutLayer`
Flatten	`nnet.keras.layer.FlattenCStyleLayer`
GlobalAveragePooling2D	`nnet.keras.layer.GlobalAveragePooling2DLayer`
Input	`imageInputLayer`
LeakyReLU	`leakyReluLayer`
MaxPooling2D	`maxPooling2dLayer`
ZeroPadding2D	`nnet.keras.layer.ZeroPadding2DLayer`

- 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, converting the images from BGR images to RGB.

- To resize images, use `imresize`. For example, `imresize(image,[227,227,3])`.
- To convert images from BGR to RGB format, use `flip`. For example, `flip(image, 3)`.

For more information on preprocessing images for training and prediction, see "Preprocess Images for Deep Learning".

References

[1] *Keras: The Python Deep Learning library*. https://keras.io

See Also

`alexnet` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasLayers` | `resnet50` | `vgg16` | `vgg19`

Topics

"Preprocess Images for Deep Learning"

Introduced in R2017b

importKerasLayers

Import series network or directed acyclic graph layers from Keras network

Syntax

```
layers = importKerasLayers(modelfile)
layers = importKerasLayers(modelfile,'Name',Value)
```

Description

`layers = importKerasLayers(modelfile)` imports the layers of a 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`.

Note This function requires Neural Network 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 a TensorFlow-Keras network architecture and weights with additional options specified by one or more of the name-value pair arguments.

For example, `importKerasLayers(modelfile,'ImportWeights',true)` imports the network architecture and the weights from the model file `modelfile`.

Examples

Download and install Neural Network Toolbox Importer *for TensorFlow-Keras Models* support package

1. From the MATLAB® desktop, select the **Home** tab, and in the **Environment** section, click the **Add-Ons** drop-down menu.

2. Select **Get Add-Ons**.

3. From the Add-On Explorer window, scroll to the **MathWorks Features** section and view the add-on file options.

4. Select the support package Neural Network Toolbox Importer *for TensorFlow-Keras Models*.

5. Click the link, and then click **Install**.

To learn more about finding and installing add-ons, see “Get Add-Ons” (MATLAB).

Import Layers from Keras Network and Plot Architecture

Specify the network file to import layers from.

```
modelfile = 'digitsDAGnet.h5';
```

Import network layers.

```
layers = importKerasLayers(modelfile)
```

```
layers =

  LayerGraph with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
```

View the layers.

```
layers.Layers
```

```
ans =

  13x1 Layer array with layers:

     1   'input_1'                                Image Input             28x28x1 images
     2   'conv2d_1'                               Convolution             20 7x7 convolutio
     3   'conv2d_1_relu'                          ReLU                    ReLU
     4   'conv2d_2'                               Convolution             20 3x3 convolutio
     5   'conv2d_2_relu'                          ReLU                    ReLU
     6   'max_pooling2d_1'                        Max Pooling             2x2 max pooling w
```

```
     7   'max_pooling2d_2'                    Max Pooling             2x2 max pooling w
     8   'flatten_1'                          Flatten C-style         Flatten activatio
     9   'flatten_2'                          Flatten C-style         Flatten activatio
    10   'concatenate_1'                      Depth concatenation     Depth concatenati
    11   'dense_1'                            Fully Connected         10 fully connecte
    12   'activation_1_softmax'               Softmax                 softmax
    13   'ClassificationLayer_activation_1'   Classification Output   crossentropyex
```

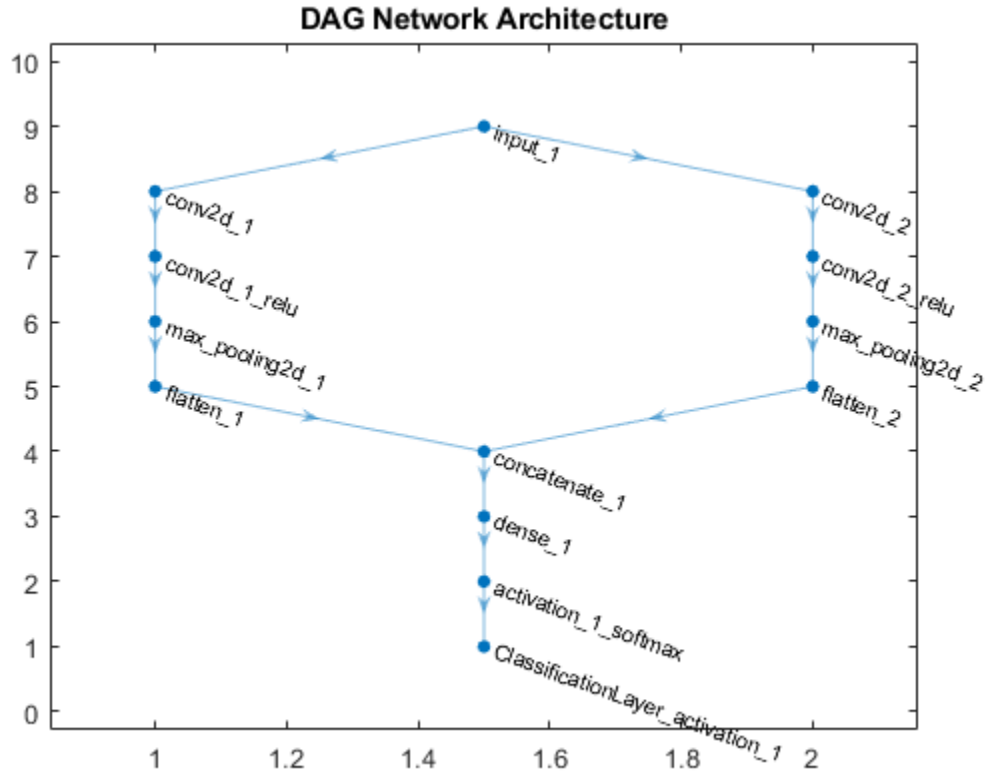
View the layer connections.

```
layers.Connections
```

```
ans =

  13×2 table

           Source                  Destination
    ____________________    ____________________________________

    'input_1'               'conv2d_1'
    'input_1'               'conv2d_2'
    'conv2d_1'              'conv2d_1_relu'
    'conv2d_1_relu'         'max_pooling2d_1'
    'conv2d_2'              'conv2d_2_relu'
    'conv2d_2_relu'         'max_pooling2d_2'
    'max_pooling2d_1'       'flatten_1'
    'max_pooling2d_2'       'flatten_2'
    'flatten_1'             'concatenate_1/in1'
    'flatten_2'             'concatenate_1/in2'
    'concatenate_1'         'dense_1'
    'dense_1'               'activation_1_softmax'
    'activation_1_softmax'  'ClassificationLayer_activation_1'
```

Plot the network architecture.

```
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.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos', ...
                  'nndatasets','DigitDataset');
digitData = imageDatastore(digitDatasetPath, ...
          'IncludeSubfolders',true,'LabelSource','foldernames');
```

Partition the dataset into training and test sets.

```
rng(1) % For reproducibility
trainingFileSplitRatio = 0.6;
[trainDigitData,testDigitData] = splitEachLabel(digitData, ...
       trainingFileSplitRatio,'randomize');
```

Set the training options.

```
options = trainingOptions('sgdm','MaxEpochs',20, ...
       'InitialLearnRate',0.001);
```

Train network using training data.

```
convnet = trainNetwork(trainDigitData,layers,options);
```

```
Training on single GPU.
|=========================================================================================
|   Epoch   |   Iteration   |   Time Elapsed   |   Mini-batch   |   Mini-batch   |   Ba
|           |               |    (hh:mm:ss)    |      Loss      |    Accuracy    |
|=========================================================================================
|         1 |             1 |         00:00:00 |         5.1335 |         13.28% |
|         2 |            50 |         00:00:09 |         1.1377 |         67.97% |
|         3 |           100 |         00:00:15 |         0.3182 |         91.41% |
|         4 |           150 |         00:00:20 |         0.0670 |         98.44% |
|         5 |           200 |         00:00:25 |         0.0617 |         99.22% |
|         6 |           250 |         00:00:31 |         0.0310 |         99.22% |
|         7 |           300 |         00:00:36 |         0.0100 |        100.00% |
|         8 |           350 |         00:00:42 |         0.0100 |        100.00% |
|         9 |           400 |         00:00:47 |         0.1060 |         99.22% |
|        10 |           450 |         00:00:54 |         0.0044 |        100.00% |
|        11 |           500 |         00:00:59 |         0.0020 |        100.00% |
```

```
|        12 |         550 |         00:01:04 |         0.0022 |        100.00% |
|        14 |         600 |         00:01:09 |         0.0012 |        100.00% |
|        15 |         650 |         00:01:14 |         0.0016 |        100.00% |
|        16 |         700 |         00:01:20 |         0.0013 |        100.00% |
|        17 |         750 |         00:01:24 |         0.0011 |        100.00% |
|        18 |         800 |         00:01:29 |         0.0008 |        100.00% |
|        19 |         850 |         00:01:33 |         0.0017 |        100.00% |
|        20 |         900 |         00:01:38 |         0.0007 |        100.00% |
|        20 |         920 |         00:01:40 |         0.0007 |        100.00% |
|=========================================================================================
```

Run the trained network on the test set that was not used to train the network and predict the image labels (digits).

```
YTest = classify(convnet,testDigitData);
TTest = testDigitData.Labels;
```

Calculate the accuracy.

```
accuracy = sum(YTest == TTest)/numel(TTest)
```

```
accuracy =

    0.9900
```

Import Keras Network Architecture and Weights from Same File

Specify the network file to import layers and weights from.

```
modelfile = 'digitsDAGnet.h5';
```

Import the network architecture and weights from the files you specified.

```
layers = importKerasLayers(modelfile,'ImportWeights',true);
```

`importKerasLayers` also imports the layers with their weights from the same HDF5 file.

You can access the weights in a particular layer by using dot notation. For example,

```
layers.Layers(2,1).Weights;
```

Import Keras Network Architecture and Weights from Separate Files

Specify the network file to import layers from and the file containing weights.

```
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.

```
layers = importKerasLayers(modelfile,'ImportWeights',true, ...
      'WeightFile',weights,'OutputLayerType','classification')
```

```
layers =

  LayerGraph with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
```

Find and Replace Placeholder Layers with Custom Layers

Specify the Keras network file to import layers from.

```
modelfile = 'digitsDAGnetwithnoise.h5';
```

Import network architecture.

```
layers = importKerasLayers(modelfile)
```

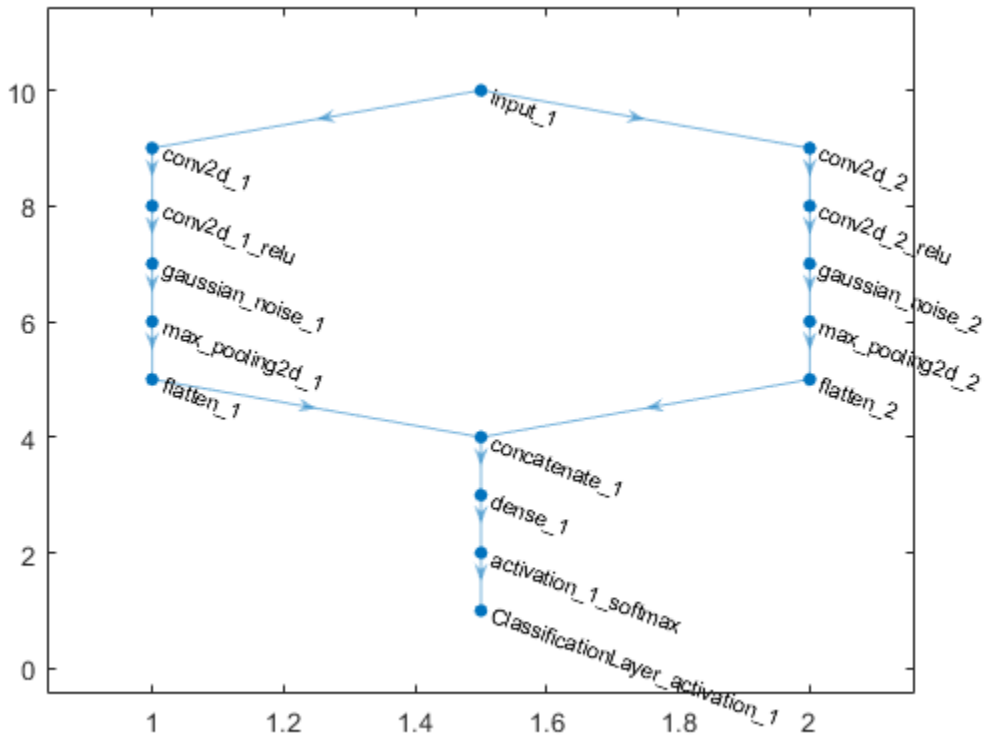
```
Warning: Unable to import some Keras layers, because they are not yet supported by the
Toolbox. They have been replaced by placeholder layers. To find these layers, call the
findPlaceholderLayers on the returned object.
> In nnet.internal.cnn.keras.importKerasLayers (line 26)
  In importKerasLayers (line 102)

layers =
```

```
  LayerGraph with properties:

         Layers: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]
```

This is Directed Acyclic Graph (DAG) network includes some layer types that are not yet supported by the Neural Network Toolbox. The `importKerasLayers` function places these layers with a placeholder layer and returns a warning message.

Plot the DAG network architecture.

```
figure()
plot(layers)
```



Take a closer look at the layers.

```
layers.Layers
```

```
ans =

  15x1 Layer array with layers:

     1   'input_1'                             Image Input             28x28x1 images
     2   'conv2d_1'                            Convolution             20 7x7 convolutio
     3   'conv2d_1_relu'                       ReLU                    ReLU
     4   'conv2d_2'                            Convolution             20 3x3 convolutio
     5   'conv2d_2_relu'                       ReLU                    ReLU
     6   'gaussian_noise_1'                    PLACEHOLDER LAYER       Placeholder for
     7   'gaussian_noise_2'                    PLACEHOLDER LAYER       Placeholder for
     8   'max_pooling2d_1'                     Max Pooling             2x2 max pooling w
     9   'max_pooling2d_2'                     Max Pooling             2x2 max pooling w
    10   'flatten_1'                           Flatten C-style         Flatten activatio
    11   'flatten_2'                           Flatten C-style         Flatten activatio
    12   'concatenate_1'                       Depth concatenation     Depth concatenati
    13   'dense_1'                             Fully Connected         10 fully connecte
    14   'activation_1_softmax'                Softmax                 softmax
    15   'ClassificationLayer_activation_1'    Classification Output   crossentropyex
```

The network is not too big, so it is easy to see that `importKerasLayers` inserts two placeholder layers instead of the GaussianNoise layers in the Keras network. With large networks, displaying all the layers and visually finding the placeholder layers might take more effort.

Find the placeholder layers that `importKerasLayers` inserts.

```
[placeholders,indices] = findPlaceholderLayers(layers)
```

```
placeholders =

  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras

indices =

     6
     7
```

The `PlaceholderLayer` object is an array that contains information about all the placeholder layers that `importKerasLayers` inserts in the imported network architecture. The indices correspond to the indices of layers in the `Layers` property of the network. You can replace these placeholder layers with your own Gaussian noise layer.

For more information on each placeholder layer, index into the object and display its properties.

```
gaussian1 = placeholders(1)
```

```
gaussian1 =

  PlaceholderLayer with properties:

    KerasConfiguration: [1×1 struct]
               Weights: []
                  Name: 'gaussian_noise_1'

  Show all properties
```

Display the Keras configuration of the first GaussianNoise layer. This information is useful for defining your own GaussianNoise layer.

```
gaussian1.KerasConfiguration
```

```
ans =

  struct with fields:

    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000
```

Display the Keras configuration of the second GaussianNoise layer.

```
gaussian2 = placeholders(2)
gaussian2.KerasConfiguration
```

```
gaussian2 =

  PlaceholderLayer with properties:

    KerasConfiguration: [1×1 struct]
               Weights: []
```

```
             NumInputs: 1
                  Name: 'gaussian_noise_2'

  Show all properties


ans =

  struct with fields:

    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Define a layer that adds random Gaussian noise to the data using the following code and save it with the name `GaussianNoiseLayer.m`.

```
classdef GaussianNoiseLayer < nnet.layer.Layer
    %GaussianNoiseLayer   A layer implementing the Keras layer
    %keras.layers.noise.GaussianNoise
    %
    %   layer = GaussianNoiseLayer(Name, Stddev) creates a layer with name
    %   Name that applies additive zero-centered Gaussian noise with
    %   standard deviation Stddev to the layer input. The noise is only
    %   applied during training, not during prediction.

    properties
        stddev      % Real scalar standard deviation.
    end

    methods
        function this = GaussianNoiseLayer(name, stddev)
            if ~(isstring(name) || ischar(name))
                error('First argument must be a string or character vector.');
            end
            if isempty(stddev) || ~(isnumeric(stddev) && isscalar(stddev) && isreal(std
                error('Second argument must be a real positive scalar.');
            end
            this.Name = name;
            this.Description = 'Apply additive zero-centered Gaussian noise during trai
            this.Type = 'Gaussian noise';
            this.stddev = stddev;
        end

        function Z = predict( this, X )
```

```
            % No noise during prediction
            Z = X;
        end

        function [Z,memory] = forward( this, X )
            % Add noise during training
            Z = X + randn(size(X))*this.stddev;
            memory = [];
        end

        function dLdX = backward( this, X, Z, dLdZ, memory )
            % Derivative of Z with respect to X doesn't depend on the noise
            dLdX = dLdZ;
        end
    end
end
```

Create Gaussian noise layers with the same configurations using the new function you defined.

```
GNlayer1 = GaussianNoiseLayer('new_gaussian_noise_1',1.5)
GNlayer2 = GaussianNoiseLayer('new_gaussian_noise_2',0.7)
```

```
GNlayer1 =

  GaussianNoiseLayer with properties:

    stddev: 1.5000
      Name: 'new_gaussian_noise_1'

  Show all properties


GNlayer2 =

  GaussianNoiseLayer with properties:

    stddev: 0.7000
      Name: 'new_gaussian_noise_2'

  Show all properties
```

Take a closer look at the connections to locate the placeholder layers.

```
layers.Connections
```

```
ans =

  15×2 table

           Source                    Destination
    _____________________    ____________________________________

    'input_1'                'conv2d_1'
    'input_1'                'conv2d_2'
    'conv2d_1'               'conv2d_1_relu'
    'conv2d_1_relu'          'gaussian_noise_1'
    'conv2d_2'               'conv2d_2_relu'
    'conv2d_2_relu'          'gaussian_noise_2'
    'gaussian_noise_1'       'max_pooling2d_1'
    'gaussian_noise_2'       'max_pooling2d_2'
    'max_pooling2d_1'        'flatten_1'
    'max_pooling2d_2'        'flatten_2'
    'flatten_1'              'concatenate_1/in1'
    'flatten_2'              'concatenate_1/in2'
    'concatenate_1'          'dense_1'
    'dense_1'                'activation_1_softmax'
    'activation_1_softmax'   'ClassificationLayer_activation_1'
```

Replace the placeholder for the first Gaussian Noise layer with the custom layer using the `replaceLayer` function.

```
layers = replaceLayer(layers,'gaussian_noise_1',GNlayer1);
new_layers.Connections
```

```
ans =

  15×2 table

           Source                    Destination
    _____________________    ____________________________________

    'input_1'                'conv2d_1'
    'input_1'                'conv2d_2'
    'conv2d_1'               'conv2d_1_relu'
    'conv2d_2'               'conv2d_2_relu'
    'conv2d_2_relu'          'gaussian_noise_2'
    'conv2d_2_relu'          'new_gaussian_noise_1'
    'gaussian_noise_2'       'max_pooling2d_2'
```

```
    'max_pooling2d_1'          'flatten_1'
    'max_pooling2d_2'          'flatten_2'
    'flatten_1'                'concatenate_1/in1'
    'flatten_2'                'concatenate_1/in2'
    'concatenate_1'            'dense_1'
    'dense_1'                  'activation_1_softmax'
    'activation_1_softmax'     'ClassificationLayer_activation_1'
    'new_gaussian_noise_1'     'max_pooling2d_1'
```

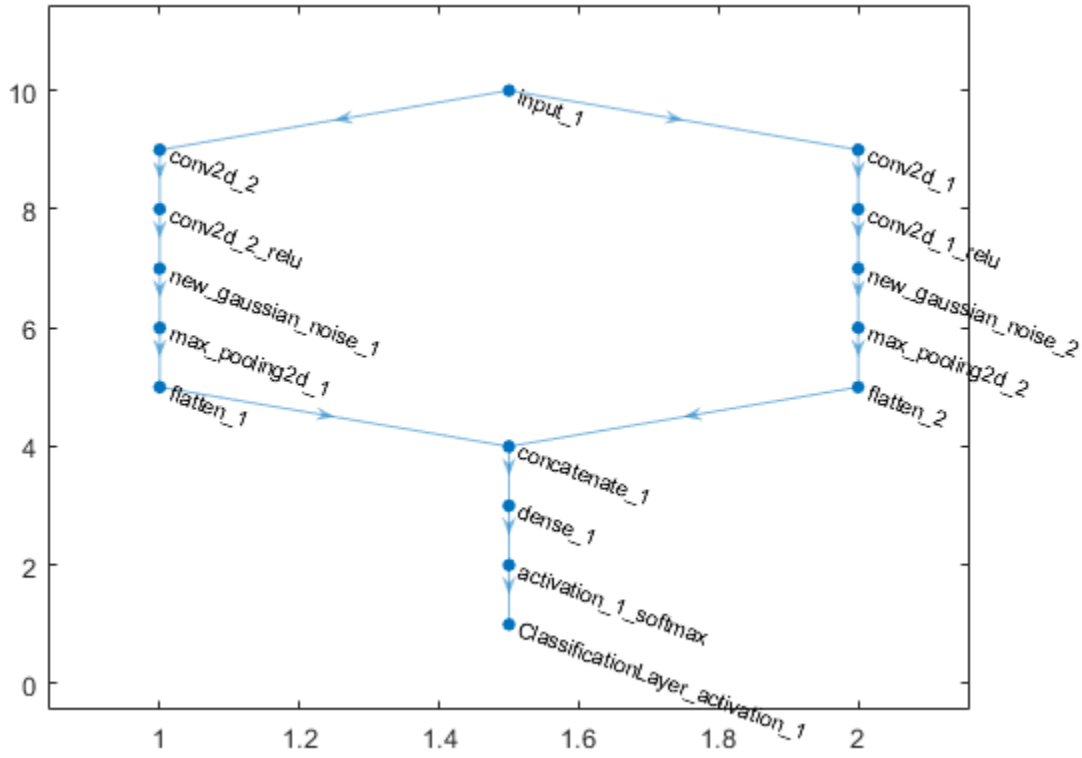
Replace the placeholder layer for the second GaussianNoise layer with the new layer in a similar way.

```
layers = replaceLayer(layers,'gaussian_noise_2',GNlayer2);
new_layers.Connections
```

```
ans =

  15×2 table

           Source                    Destination
    _____________________    ____________________________________

    'input_1'                'conv2d_1'
    'input_1'                'conv2d_2'
    'conv2d_1'               'conv2d_1_relu'
    'conv2d_1_relu'          'new_gaussian_noise_2'
    'conv2d_2'               'conv2d_2_relu'
    'conv2d_2_relu'          'new_gaussian_noise_1'
    'max_pooling2d_1'        'flatten_1'
    'max_pooling2d_2'        'flatten_2'
    'flatten_1'              'concatenate_1/in1'
    'flatten_2'              'concatenate_1/in2'
    'concatenate_1'          'dense_1'
    'dense_1'                'activation_1_softmax'
    'activation_1_softmax'   'ClassificationLayer_activation_1'
    'new_gaussian_noise_1'   'max_pooling2d_1'
    'new_gaussian_noise_2'   'max_pooling2d_2'
```

Plot the DAG network structure.

```
figure()
plot(new_layers)
```



- “Deep Learning in MATLAB”
- “Pretrained Convolutional Neural Networks”

Input Arguments

`modelfile` — Name of Keras model file
character vector

Name of the Keras model file containing the network architecture, and possibly the weights, specified as a character vector. 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. In this case, optionally you can supply the weights using the `'ImportWeights'` and `'WeightFile'` name-value pair arguments. If you supply weights, then the weights file must be in HDF5 format.

`modelfile` 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: `'digitsnet.h5'`

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `importKerasLayers(modelfile,'OutputLayerType','classification')` specifies `importKerasLayers` to put an output layer for a classification problem at the end of the Keras layers.

`OutputLayerType` — Type of output layer
`'classification'` | `'regression'`

Type of the output layer that `importKerasLayers` appends to the end of the imported network architecture when `modelfile` does not specify a loss function, specified as `'classification'` or `'regression'`.

Example: `'OutputLayerType','regression'`

Data Types: `char`

`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

Weight file name, from which to import weights when `modelfile` does not include weights, specified as a character vector. 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'`

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` can import the following Keras layer types. If the network contains any other type of layer, then the software inserts a place holder layer in place of the unsupported layer. To learn the names and indices of the unsupported layers in the network, call the `findPlaceholderLayers` function on the returned object.

Supported Keras Layers

Keras Layer	NNET Layer Function
Add	`additionLayer`

Activation, with activation names:
• relu
• linear
• softmax
• sigmoid
• tanh
	`reluLayer`
	`softmaxLayer`
	`nnet.keras.layer.SigmoidLayer`
	`nnet.keras.layer.TanhLayer`

AveragePooling2D	`averagePooling2dLayer`
BatchNormalization	`BatchNormalizationLayer`
Concatenate	`depthConcatenationLayer`
Conv2D	`convolution2dLayer`
Conv2DTranspose	`transposedConv2dLayer`
Dense	`fullyConnectedLayer`
Dropout	`dropoutLayer`
Flatten	`nnet.keras.layer.FlattenCStyleLayer`
GlobalAveragePooling2D	`nnet.keras.layer.GlobalAveragePooling2DLayer`
Input	`imageInputLayer`
LeakyReLU	`leakyReluLayer`
MaxPooling2D	`maxPooling2dLayer`
ZeroPadding2D	`nnet.keras.layer.ZeroPadding2DLayer`

- You can replace a placeholder layer with a new layer that you wrote.
  - If the network is a series network, then replace the layer by assigning the layer to the location of the placeholder layer. For example, `layer(2) = newlayer;`
  - If the network is a DAG network, then replace the layer by first removing the placeholder layer from the network, and then inserting the new layer and

redefining the connections. Also, you can conveniently use the `replaceLayer` function. For an example, see “Find and Replace Placeholder Layers with Custom Layers” on page 1-1107.

References

[1] *Keras: The Python Deep Learning library*. https://keras.io

See Also

`alexnet` | `findPlaceholderLayers` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `importKerasNetwork` | `replaceLayer` | `resnet50` | `vgg16` | `vgg19`

Topics

“Deep Learning in MATLAB”
“Pretrained Convolutional Neural Networks”

Introduced in R2017b

findPlaceholderLayers

Find placeholder layers in `Layer` array or `LayerGraph` imported using `importKerasLayers`

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` function. Placeholder layers are the layers `importKerasLayers` inserts in place of the Keras layers that are not yet supported by the Neural Network Toolbox.

Note This function requires Neural Network Toolbox Importer *for TensorFlow-Keras Models* support package. If this support package is not installed, type `importKerasLayer` or `importKerasNetwork` at the Command Line for a download link.

`[placeholderlayers,indices] = findPlaceholderLayers(importedLayers)` also returns the indices of the placeholder layers.

Examples

Import Keras Network Layers and Find Placeholder Layers

Specify the file to import the layers from.

```
modelfile = 'digitsDAGnetwithnoise.h5';
```

Import the network layers.

```
importedLayers = importKerasLayers(modelfile)
```

```
Warning: Unable to import some Keras layers, because they are not yet supported by the
Toolbox. They have been replaced by placeholder layers. To find these layers, call the
findPlaceholderLayers on the returned object.
> In nnet.internal.cnn.keras.importKerasLayers (line 26)
  In importKerasLayers (line 102)

importedLayers =

  LayerGraph with properties:

         Layers: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]
```

This Directed Acyclic Graph (DAG) network includes some layer types that are not yet supported by the Neural Network Toolbox. The `importKerasLayers` function places each layer with a placeholder layer and returns a warning message.

Display the network layers.

```
importedLayers.Layers
```

```
ans =

  15x1 Layer array with layers:

     1   'input_1'                           Image Input             28x28x1 images
     2   'conv2d_1'                          Convolution             20 7x7 convolutio
     3   'conv2d_1_relu'                     ReLU                    ReLU
     4   'conv2d_2'                          Convolution             20 3x3 convolutio
     5   'conv2d_2_relu'                     ReLU                    ReLU
     6   'gaussian_noise_1'                  PLACEHOLDER LAYER       Placeholder for
     7   'gaussian_noise_2'                  PLACEHOLDER LAYER       Placeholder for
     8   'max_pooling2d_1'                   Max Pooling             2x2 max pooling w
     9   'max_pooling2d_2'                   Max Pooling             2x2 max pooling w
    10   'flatten_1'                         Flatten C-style         Flatten activatio
    11   'flatten_2'                         Flatten C-style         Flatten activatio
    12   'concatenate_1'                     Depth concatenation     Depth concatenati
    13   'dense_1'                           Fully Connected         10 fully connecte
    14   'activation_1'                      Softmax                 softmax
    15   'ClassificationLayer_activation_1'  Classification Output   crossentropyex
```

`importKerasLayers` inserts placeholder layers instead of the GaussianNoise layers in the Keras network. The network is not too big, so it is easy to see the placeholder layers. With large networks, displaying all the layers and visually finding the placeholder layers might take more effort.

Find the placeholder layers and indices.

```
[placeholderlayers,indices] = findPlaceholderLayers(importedLayers)
```

```
placeholderlayers =

  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras

indices =

     6
     7
```

The `PlaceholderLayer` object is an array that contains information about all the placeholder layers that `importKerasLayers` inserts in the imported network architecture. The indices correspond to the indices of layers in the `Layers` property of the network architecture.

For more information on each placeholder layer, index into the object and display its properties.

```
gaussian1 = placeholderlayers(1)
```

```
gaussian1 =

  PlaceholderLayer with properties:

    KerasConfiguration: [1×1 struct]
               Weights: []
                  Name: 'gaussian_noise_1'

  Show all properties
```

You can replace these placeholder layers with your own Gaussian noise layer. Display the Keras configuration of the Gaussian noise layer. This information is useful if you choose to define your own layer.

```
gaussian1.KerasConfiguration

ans =

  struct with fields:

    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000
```

For an example of writing your own layer and replacing a placeholder layer, see “Find and Replace Placeholder Layers with Custom Layers” on page 1-1107.

Input Arguments

`importedLayers` — Network architecture imported by `importKerasLayers`

`Layer` array | `LayerGraph`

Network architecture imported by `importKerasLayers`, specified as a `Layer` array when Keras network is of type 'Sequential' or specified as a `LayerGraph` object when Keras network is of type 'Model'.

Output Arguments

`placeholderlayers` — All placeholder layers in network architecture

array of `PlaceholderLayer` objects

All placeholder layers in the network architecture that `importKerasLayers` imports, returned as an array of `PlaceholderLayer` objects.

`indices` — Indices of placeholder layers

vector

Indices of placeholder layers in the `Layers` property of `importedLayers`, returned as a vector.

If you remove a layer from or add a layer to a `Layer` array or `LayerGraph`, then the indices of the other layers in the object might change. You must call `findPlaceholderLayers` function again to find the updated indices of the rest of the placeholder layers.

References

[1] *Keras: The Python Deep Learning library*. https://keras.io

See Also

`PlaceholderLayer` | `importKerasLayers` | `replaceLayer`

Introduced in R2017b

PlaceholderLayer

Layer to replace an unsupported Keras layer

Description

`PlaceholderLayer` is a layer that `importKerasLayers` inserts into a `Layer` array or a `LayerGraph` in place of an unsupported Keras layer.

Creation

Importing layers from a Keras network that has layers that are not yet supported by Neural Network Toolbox creates the instances of a `PlaceholderLayer` object.

Properties

`KerasConfiguration` — Keras configuration of layer
structure

Keras configuration of a layer, returned as a structure. The fields of the structure depend on the layer type.

Data Types: `struct`

`Weights` — Weights imported from Keras layer
structure

Weights imported from a Keras layer, if any, returned as a structure. The fields of the structure depend on the layer type.

Data Types: `struct`

Examples

Find and Explore Placeholder Layers

Specify the Keras network file to import layers from.

```
modelfile = 'digitsDAGnetwithnoise.h5';
```

Import network architecture.

```
netlayers = importKerasLayers(modelfile)
```

```
Warning: Unable to import some Keras layers, because they are not yet supported by the
Toolbox. They have been replaced by placeholder layers. To find these layers, call the
findPlaceholderLayers on the returned object.
> In nnet.internal.cnn.keras.importKerasLayers (line 26)
  In importKerasLayers (line 102)

importedLayers =

  LayerGraph with properties:

         Layers: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]
```

`digitsDAGnetwithnoise.h5` contains a Directed Acyclic Graph (DAG) network. The network includes some layer types that are not yet supported by the Neural Network Toolbox. The `importKerasLayers` function replaces these layers with a placeholder layer and returns a warning message.

Take a closer look at the layers.

```
netlayers.Layers
```

```
ans =

  15x1 Layer array with layers:

     1   'input_1'                          Image Input             28x28x1 images
     2   'conv2d_1'                         Convolution             20 7x7 convolutio
     3   'conv2d_1_relu'                    ReLU                    ReLU
     4   'conv2d_2'                         Convolution             20 3x3 convolutio
     5   'conv2d_2_relu'                    ReLU                    ReLU
     6   'gaussian_noise_1'                 PLACEHOLDER LAYER       Placeholder for
```

```
     7   'gaussian_noise_2'                  PLACEHOLDER LAYER       Placeholder for
     8   'max_pooling2d_1'                   Max Pooling             2x2 max pooling w
     9   'max_pooling2d_2'                   Max Pooling             2x2 max pooling w
    10   'flatten_1'                         Flatten C-style         Flatten activatio
    11   'flatten_2'                         Flatten C-style         Flatten activatio
    12   'concatenate_1'                     Depth concatenation     Depth concatenati
    13   'dense_1'                           Fully Connected         10 fully connecte
    14   'activation_1_softmax'              Softmax                 softmax
    15   'ClassificationLayer_activation_1'  Classification Output   crossentropyex
```

Two placeholder layers replace the GaussianNoise layers in the Keras network.

Find the placeholder layers.

```
placeholders = findPlaceholderLayers(netlayers)
```

```
placeholders =

  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras
```

`placeholders` is a `PlaceholderLayer` object that contains the two placeholder layers `importKerasLayers` inserted in place of the GaussianNoise layer of the Keras network.

For more information on each placeholder layer, index into the object and display its properties.

```
gaussian1 = placeholders(1)
gaussian2 = placeholders(2)
```

```
gaussian1 =

  PlaceholderLayer with properties:

    KerasConfiguration: [1×1 struct]
               Weights: []
                  Name: 'gaussian_noise_1'

  Show all properties


gaussian2 =
```

```
  PlaceholderLayer with properties:

    KerasConfiguration: [1×1 struct]
               Weights: []
                  Name: 'gaussian_noise_2'

  Show all properties
```

Display the Keras configuration of each 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
```

See Also

`findPlaceholderLayers` | `importKerasLayers`

Introduced in R2017b

checkLayer

Check validity of custom layer

Syntax

```
checkLayer(layer,validInputSize)
checkLayer(layer,validInputSize,'ObservationDimension',dim)
```

Description

`checkLayer(layer,validInputSize)` checks the validity of a custom layer using generated data of size `validInputSize`. Specify `validInputSize` to be the typical size of a single observation input to the layer.

`checkLayer(layer,validInputSize,'ObservationDimension',dim)` specifies the dimension of the data which 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 layer validity of the example layer `examplePreluLayer`. To use this layer, add the example folder to the path.

```
exampleFolder = genpath(fullfile(matlabroot,'examples','nnet'));
addpath(exampleFolder)
```

Create an instance of the layer and check that it is valid using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects inputs of size *h*-by-*w*-by-*c*, where *h*, *w*, and *c* are the height, width, and number of channels of the previous layer output, respectively.

Specify `validInputSize` to be the size of a typical 3-D array input.

```
layer = examplePreluLayer(20);
validInputSize = [5 5 20];
checkLayer(layer,validInputSize)
```

```
Running nnet.checklayer.TestCase
....
================================================================================
nnet.checklayer.TestCase/predictDoesNotError(Observations=multiple) was filtered.
    Test Diagnostic: Test skipped. To enable checks with multiple observations, specify
    For layers used in convolutional neural networks, set 'ObservationDimension' to be
    For layers used in recurrent neural networks, set 'ObservationDimension' to be 2.
Details
================================================================================
....
================================================================================
nnet.checklayer.TestCase/backwardDoesNotError(Observations=multiple) was filtered.
    Test Diagnostic: Test skipped. To enable checks with multiple observations, specify
    For layers used in convolutional neural networks, set 'ObservationDimension' to be
    For layers used in recurrent neural networks, set 'ObservationDimension' to be 2.
Details
================================================================================
..

================================================================================
nnet.checklayer.TestCase/backwardIsConsistentInSize(Observations=multiple) was filtered
    Test Diagnostic: Test skipped. To enable checks with multiple observations, specify
    For layers used in convolutional neural networks, set 'ObservationDimension' to be
    For layers used in recurrent neural networks, set 'ObservationDimension' to be 2.
Details
================================================================================
..........
.
Done nnet.checklayer.TestCase
__________

Failure Summary:

     Name                                                                      Failed
    ================================================================================
     nnet.checklayer.TestCase/predictDoesNotError(Observations=multiple)
    --------------------------------------------------------------------------------
     nnet.checklayer.TestCase/backwardDoesNotError(Observations=multiple)
    --------------------------------------------------------------------------------
```

```
     nnet.checklayer.TestCase/backwardIsConsistentInSize(Observations=multiple)

Test Summary:
     18 Passed, 0 Failed, 3 Incomplete.
     Time elapsed: 7.3724 seconds.
```

The results show no failed tests and three skipped tests. The skipped tests check that the layers work with multiple observations. To check that the layer is valid for multiple observations, use the `'ObservationsDimension'` option.

Check Multiple Observations

By default, built-in layers output 4-D arrays with size *h*-by-*w*-by-*c*-by-*N*, except for LSTM layers and sequence input layers, which output 3-D arrays of size *D*-by-*N*-by-*S*.

These dimensions correspond to the following:

- *h* – Height of the output
- *w* – Width of the output
- *c* – Number of channels in the output
- *N* – Mini-batch size
- *D* – Feature dimension of sequence
- *S* – Sequence length

The layer `examplePreluLayer` expects 4-D array inputs from the previous layer, where the 4th dimension indexes the observations. Specify the typical size of an observation (a 3-D array) and set `'ObservationDimension'` to 4.

```
layer = examplePreluLayer(20);
validInputSize = [5 5 20];
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

```
Running nnet.checklayer.TestCase
..........
..........
.
Done nnet.checklayer.TestCase
__________

Test Summary:
     21 Passed, 0 Failed, 0 Incomplete.
     Time elapsed: 0.20383 seconds.
```

Here, the function does not detect any issues with the layer.

To remove the example folder, use `rmpath`.

```
rmpath(exampleFolder)
```

Input Arguments

`layer` — Input layer

`nnet.layer.Layer` object

Input layer, specified as an `nnet.layer.Layer` object. For an example showing how to define your own layer, see "Define a Custom Deep Learning Layer with Learnable Parameters".

The `checkLayer` function does not support built-in layers.

`validInputSize` — Valid input size

vector of positive integers

Valid input size of the layer, specified as a vector of positive integers. Specify `validInputSize` to be the typical size of a single observation input to the layer.

Example: `[5 5 10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`dim` — Observation dimension

positive integer

Observation dimension, specified as a positive integer. The observation dimension depends on the output of the previous layer. Specify `dim` to be the dimension which indexes the observations in the mini-batch.

Built-in layers output 4-D arrays with size *h*-by-*w*-by-*c*-by-*N*, except for LSTM layers and sequence input layers, which output 3-D arrays of size *D*-by-*N*-by-*S*.

Fully connected, ReLU, dropout, and softmax layers also accept 3-D inputs. When these layers get inputs of this shape, then they output 3-D arrays of size *D*-by-*N*-by-*S*.

These dimensions correspond to the following:

- *h* – Height of the output
- *w* – Width of the output
- *c* – Number of channels in the output
- *N* – Number of observations (mini-batch size)
- *D* – Feature dimension of sequence
- *S* – Sequence length

For example, if `layer` is a convolutional layer that accepts input of size *h*-by-*w*-by-*c*-by-*N*, where *N* is the number of observations, then set `dim` to 4. If `layer` is a recurrent layer that accepts input of size *D*-by-*N*-by-*S*, where *N* is the number of observations, then set `dim` to 2.

Example: 4

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Algorithms

List of Tests

The `checkLayer` function checks the validity of a custom layer by performing a series of tests, described in this table.

Test	Description
`predictSignatureIsWellDefined`	The signature for `predict` has exactly two input arguments and one output argument.
`forwardSignatureIsWellDefined`	The signature for `forward` has exactly two input arguments and two output arguments. If you have not implemented `forward`, then `checkLayer` does not run this test.

Test	Description
`backwardSignatureIsWellDefined`	The signature for `backward` has exactly five input arguments and 1 + *n* output arguments, where *n* is the number of learnable parameters.
`predictDoesNotError`	`predict` does not throw an error.
`forwardDoesNotError`	`forward` does not throw an error. If you have not implemented `forward`, then `checkLayer` does not run this test.
`backwardDoesNotError`	`backward` does not throw an error.
`backwardIsConsistentInSize`	The output of `backward` is consistent in size: `dLdX` must be the same size as `X`, and each `dLdW` must be the same size as `W`.
`predictIsConsistentInType`	The output of `predict` is consistent in type: `Z` must be the same type as `X`. This test also checks for GPU compatibility. To execute the layer functions on a GPU, they must support inputs and outputs of type `gpuArray` with the underlying data type `single`.
`forwardIsConsistentInType`	The output of `forward` is consistent in type: `Z` must be the same type as `X`. If you have not implemented `forward`, then `checkLayer` does not run this test. This test also checks for GPU compatibility. To execute the layer functions on a GPU, they must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

Test	Description
`backwardIsConsistentInType`	The output of `backward` is consistent in type: `dLdX` must be the same type as `X`, and each `dLdW` must be the same type as `W`. This test also checks for GPU compatibility. To execute the layer functions on a GPU, they must support inputs and outputs of type `gpuArray` with the underlying data type `single`.
`gradientsAreNumericallyCorrect`	The gradients computed in `backward` are consistent with the numerical gradients.

For more information on the tests used by `checkLayer`, see “Check Custom Layer Validity”.

See Also

`trainNetwork` | `trainingOptions`

Topics

“Check Custom Layer Validity”
“Deep Learning in MATLAB”
“Define Custom Deep Learning Layers”
“Define a Custom Deep Learning Layer with Learnable Parameters”

Introduced in R2018a

replaceLayer

Replace layer in layer graph

Syntax

```
newlgraph = replaceLayer(lgraph,layerName,larray)
```

Description

`newlgraph = replaceLayer(lgraph,layerName,larray)` replaces the layer `layerName` in the layer graph `lgraph` with the layer array `larray`.

`replaceLayer` connects the layers in `larray` sequentially and it connects `larray` into the layergraph `lgraph`.

Note This function requires Neural Network Toolbox Importer *for TensorFlow-Keras Models* support package. If this support package is not installed, type `importKerasLayer` or `importKerasNetwork` at the Command Line for a download link.

Examples

Replace Layer in DAG Network with Layer Array

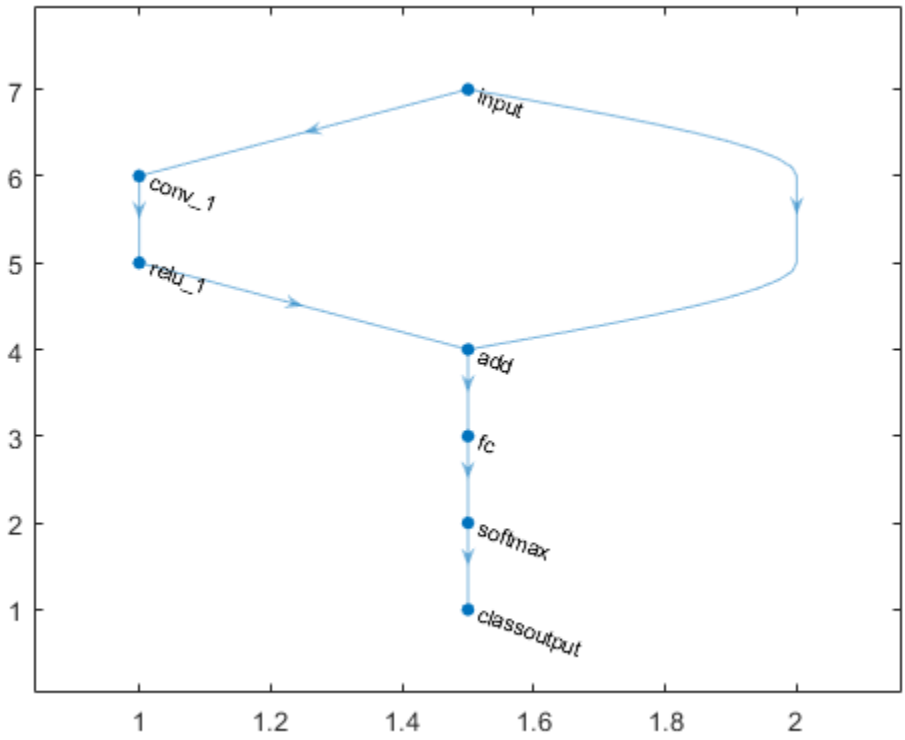
Define a Directed Acyclic Graph network architecture.

```
layers = [
            imageInputLayer([28 28 1],'Name','input')
            convolution2dLayer(5,20,'Name','conv_1')
            reluLayer('Name','relu_1')
            additionLayer(2,'Name','add')
            fullyConnectedLayer(10,'Name','fc')
            softmaxLayer('Name','softmax')
```

```
            classificationLayer('Name','classoutput')];
lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'input','add/in2');
```

Plot the network architecture.

```
figure
plot(lgraph)
```



Display the network connections.

```
lgraph.Connections
```

```
ans =
```

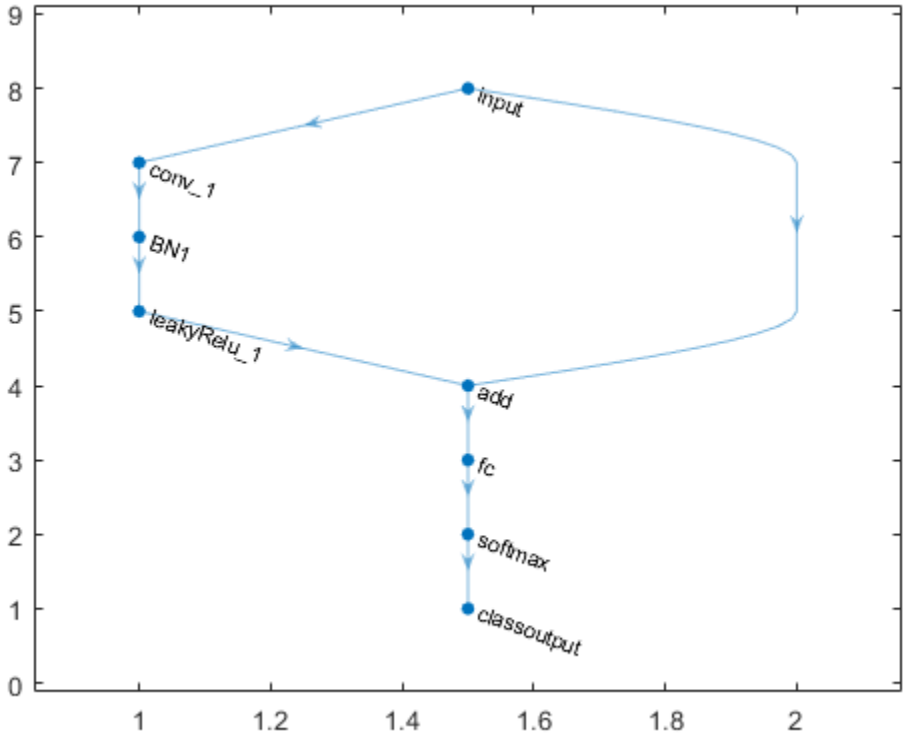
```
  7×2 table

     Source       Destination
    _________    _____________

    'input'      'conv_1'
    'input'      'add/in2'
    'conv_1'     'relu_1'
    'relu_1'     'add/in1'
    'add'        'fc'
    'fc'         'softmax'
    'softmax'    'classoutput'
```

Replace the ReLU layer in the network with a batch normalization layer followed by a leaky ReLU layer.

```
larray = [batchNormalizationLayer('Name','BN1')
          leakyReluLayer('Name','leakyRelu_1','Scale',.02)];
newlgraph = replaceLayer(lgraph,'relu_1',larray);
```

Plot the new network architecture.

```
figure
plot(newlgraph)
```



Display the updated network connections.

```
newlgraph.Connections
```

```
ans =

  8×2 table

      Source          Destination
    _____________    _____________

    'input'          'conv_1'
    'input'          'add/in2'
    'conv_1'         'BN1'
```

```
    'add'            'fc'
    'fc'             'softmax'
    'softmax'        'classoutput'
    'BN1'            'leakyRelu_1'
    'leakyRelu_1'    'add/in1'
```

Input Arguments

`lgraph` — Directed Acyclic Graph network architecture
`LayerGraph` object

Directed Acyclic Graph network architecture, specified as a `LayerGraph` object.

`layerName` — Name of the layer to replace
string | character vector

Name of the layer to replace in `lgraph`, specified as a string or a character vector.

`larray` — Array of layers
`Layer` array object

Array of layers, specified as a `Layer` array object.

Output Arguments

`newlgraph` — Updated DAG network architecture
`LayerGraph` object

Updated DAG network, returned as a `LayerGraph` object.

See Also

`PlaceholderLayer` | `connectLayers` | `findPlaceholderLayers` | `layerGraph`

Introduced in R2018a