Neural Network Toolbox™

Reference

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth

MATLAB®

R2015b

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)

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.

More About

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

`sim` | `init` | `train` | `revert`

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

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

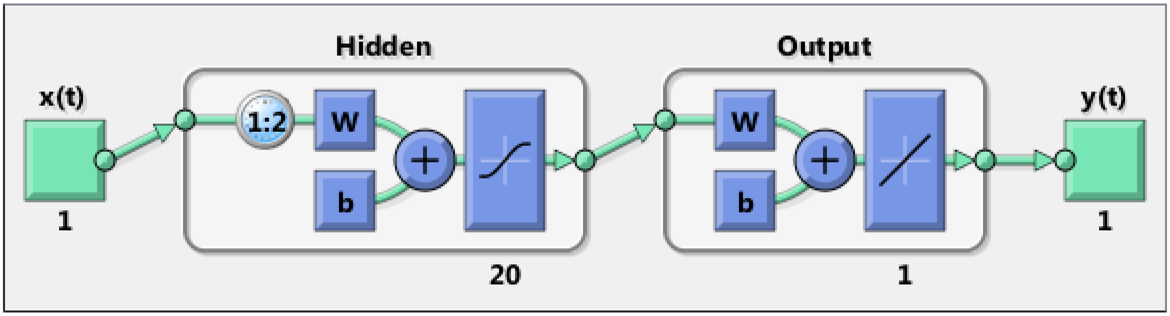
and returns the network with input delay connections increased, and output feedback delays decreased, by the specified number of delays `n`. The result is a network that behaves identically, except that outputs are produced `n` timesteps later.

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

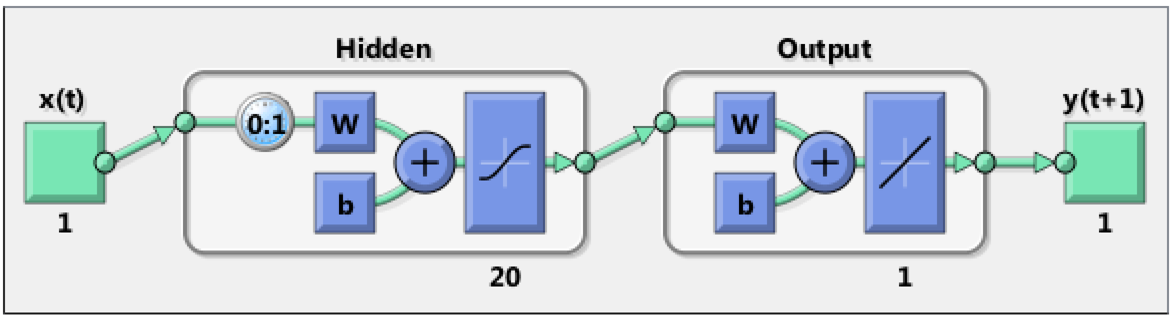
Examples

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.

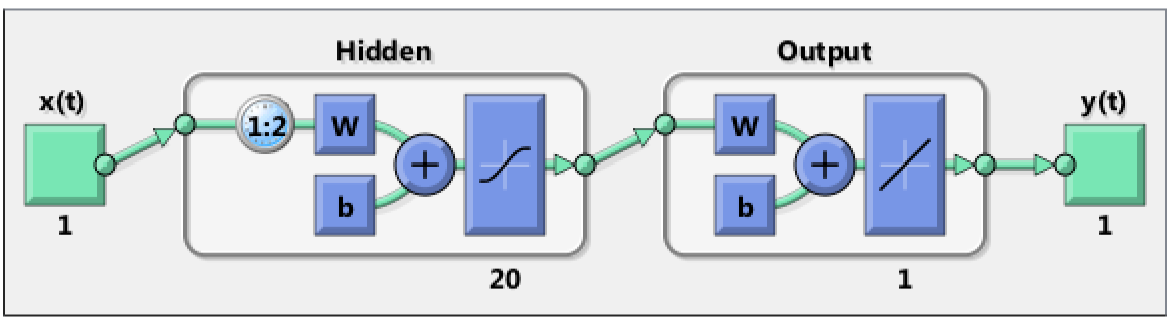
```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)
```




```
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`

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`.

More About

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

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

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'`)

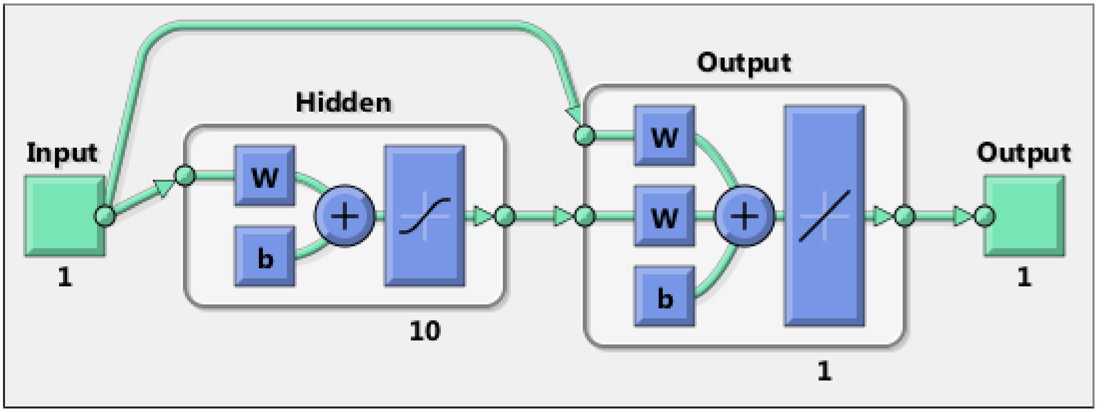
and returns a new cascade-forward neural network.

Examples

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
```



More About

- "Create, Configure, and Initialize Multilayer Neural Networks"
- "Neural Network Object Properties"
- "Neural Network Subobject Properties"

See Also

`feedforwardnet` | `network`

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

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

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

`nndata` | `numsamples` | `getsamples` | `setsamples` | `catelements` | `catsignals` | `cattimesteps`

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

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

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

`nndata` | `numtimesteps` | `gettimesteps` | `settimesteps` | `catelements` | `catsignals` | `catsamples`

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`

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);
Ycloesed = 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`

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.

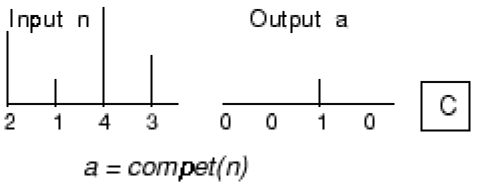
Examples

```
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
```

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

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,

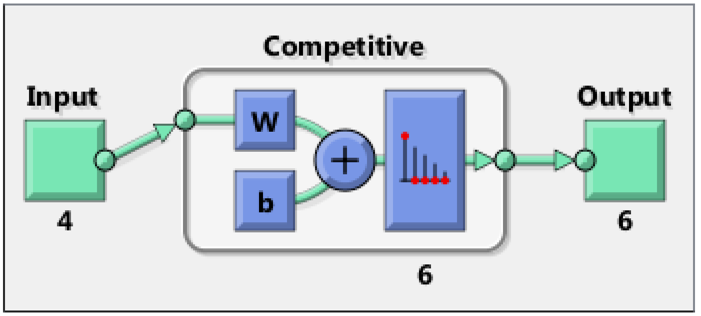
`numClasses`	Number of classes to classify inputs (default = 5)
`kohonenLR`	Learning rate for Kohonen weights (default = 0.01)
`conscienceLR`	Learning rate for conscience bias (default = 0.001)

and returns a competitive layer with `numClasses` neurons.

Examples

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

`selforgmap` | `lvqnet` | `patternnet`

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

seq2con | concur

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

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

isconfigured | init | train | unconfigure

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 ith 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

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`.

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 =

    0.0267
```

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

disp

Neural network properties

Syntax

```
disp(net)
```

To Get Help

Type `help network/disp`.

Description

`disp(net)` displays a network's properties.

Examples

Here a perceptron is created and displayed.

```
net = newp([-1 1; 0 2],3);
disp(net)
```

See Also

`display` | `sim` | `init` | `train` | `adapt`

display

Name and properties of neural network variables

Syntax

```
display(net)
```

To Get Help

Type `help network/display`.

Description

`display(net)` displays a network variable's name and properties.

Examples

Here a perceptron variable is defined and displayed.

```
net = newp([-1 1; 0 2],3);
display(net)
```

`display` is automatically called as follows:

```
net
```

See Also

`disp` | `sim` | `init` | `train` | `adapt`

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.

More About

Algorithms

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

```
d = sum((x-y).^2).^0.5
```

See Also

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

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'`)

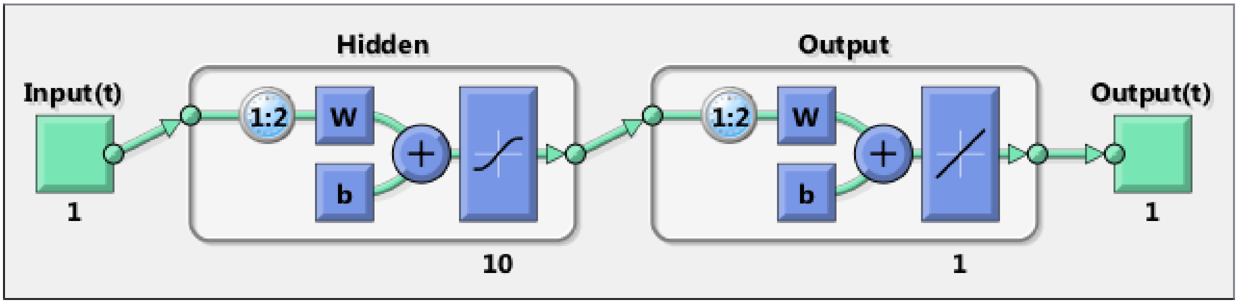
and returns a distributed delay neural network.

Examples

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

preparets | removedelay | timedelaynet | narnet | narxnet

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`

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`

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

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

dividetrain

Assign all targets to training set

Syntax

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

Description

`[trainInd,valInd,testInd] =`
`dividetrain(Q,trainRatio,valRatio,testRatio)` assigns all targets to the training set and no targets to either 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(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 | divideind | divideint | dividerand

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

sim | dist | feedforwardnet | negdist | normprod

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`

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`

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

preparets | removedelay | timedelaynet | layrecnet | narnet | narxnet

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

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

`nndata` | `catsamples` | `preparets`

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.

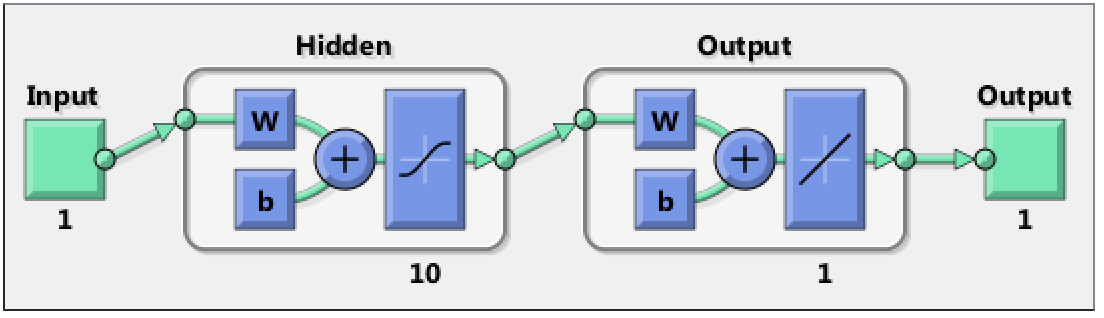
Examples

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
```



More About

- "Neural Network Object Properties"
- "Neural Network Subobject Properties"

See Also

fitnet | network | patternnet | cascadeforwardnet

fitnet

Function fitting neural network

Syntax

```
fitnet(hiddenSizes,trainFcn)
```

Description

Fitting networks are feedforward neural networks (`feedforwardnet`) used to fit an input-output relationship.

`fitnet(hiddenSizes,trainFcn)` takes these arguments,

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

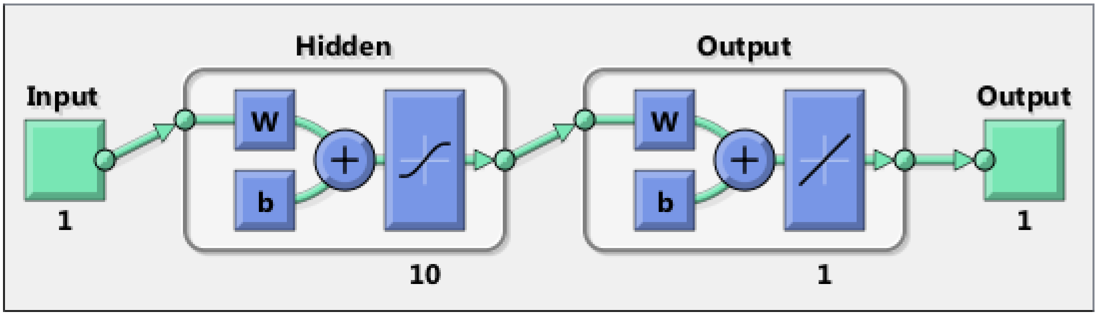
and returns a fitting neural network.

Examples

Here a fitting neural network is used to solve a simple problem.

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

```
perf =

    1.4639e-04
```



More About

- “Fit Data with a Neural Network”
- “Neural Network Object Properties”
- “Neural Network Subobject Properties”

See Also

`feedforwardnet` | `network` | `nftool`

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

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

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` | `setwb` | `separatewb`

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

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

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

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

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


ans =

    11    12    13
    24    25    26


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


ans =

    [2]    [5]
    [8]    [6]


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


ans =
```

```
[11]    [12]    [13]    [14]
[21]    [22]    [23]    [24]
[31]    [32]    [33]    [34]
```

See Also

gsubtract | gdivide | gnegate | gsqrt | gmultiply

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

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

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


ans =

    0.1000    0.2000    0.3000
    0.2000    0.2500    0.3000


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


ans =

    [      1]    [0.6667]
    [0.6000]    [      2]


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


ans =
```

```
[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` | `gsubtract` | `gnegate` | `gsqrt` | `gmultiply`

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)
```

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] = house_dataset;
houseNet = feedforwardnet(10);
houseNet = train(houseNet,x,t);
```

```
y = houseNet(x);
```

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

```
genFunction(houseNet,'houseFcn');
y2 = houseFcn(x);
accuracy2 = max(abs(y-y2))
mcc -W lib:libHouse -T link:lib houseFcn
```

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(houseNet,'houseFcn','MatrixOnly','yes');
y3 = houseFcn(x);
accuracy3 = max(abs(y-y3))

x1Type = coder.typeof(double(0),[13 Inf]); % Coder type of input 1
codegen houseFcn.m -config:mex -o houseCodeGen -args {x1Type}
y4 = houseCodeGen(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`

More About

- "Deploy Neural Network Functions"

See Also

`gensim`

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

`nndata` | `numelements` | `setelements` | `catelements` | `getsamples` | `gettimesteps` | `getsignals`

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

`nndata` | `numsamples` | `setsamples` | `catsamples` | `getelements` | `gettimesteps` | `getsignals`

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

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

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` | `setsiminit` | `nndata2sim` | `sim2nndata`

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

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

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

`setwb` | `formwb` | `separatewb`

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

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

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

```
ans =

    10    20    30
    80   100   120
```

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

```
ans =

    [ 1]    [6]
    [15]    [8]
```

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

```
ans =
```

```
[10]    [20]    [30]    [ 40]
[20]    [40]    [60]    [ 80]
[30]    [60]    [90]    [120]
```

See Also

gadd | gsubtract | gdivide | gnegate | gsqrt

gnegate

Generalized negation

Syntax

`gnegate(x)`

Description

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

Examples

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 =

    -1    -2
    -3    -4


ans =

    -1     3
     5    -2
```

See Also

`gadd` | `gsubtract` | `gsqrt` | `gdivide` | `gmultiply`

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

gridtop

Grid layer topology function

Syntax

```
gridtop(dim1,dim2,...,dimN)
```

Description

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

`gridtop(dim1,dim2,...,dimN)` takes `N` arguments,

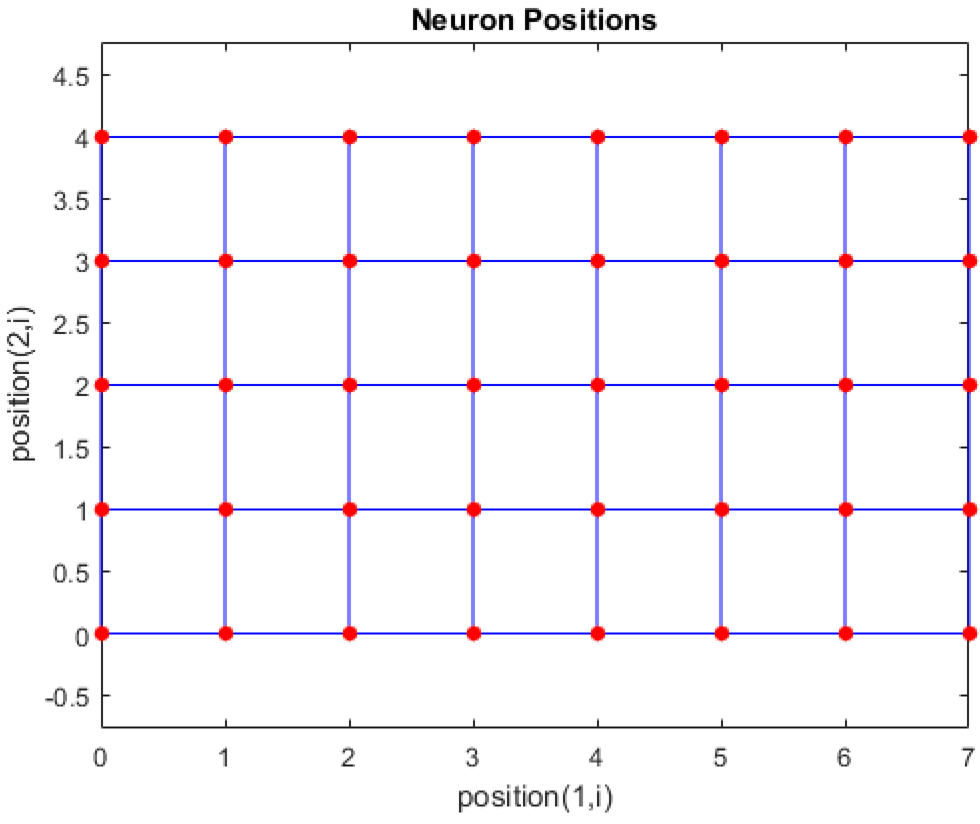
`dimi`	Length of layer in dimension `i`

and returns an `N`-by-`S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

Examples

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

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

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

```
gsqrt({1 2; 3 4})
```

```
ans =

    [     1]    [1.4142]
    [1.7321]    [     2]
```

See Also

`gadd` | `gsubtract` | `gnegate` | `gdivide` | `gmultiply`

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

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

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

```
ans =

    -9    -8    -7
   -16   -15   -14
```

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

```
ans =

    [ 0]    [-1]
    [-2]    [ 2]
```

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

```
ans =
```

```
[  -9]     [  -8]     [  -7]     [  -6]
[-19]     [-18]     [-17]     [-16]
[-29]     [-28]     [-27]     [-26]
```

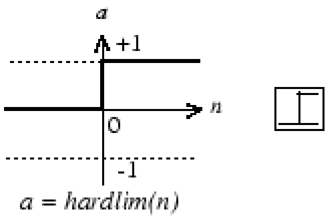
See Also

gadd | gmultiply | gdivide | gnegate | gsqrt

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';
```

More About

Algorithms

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

0 otherwise

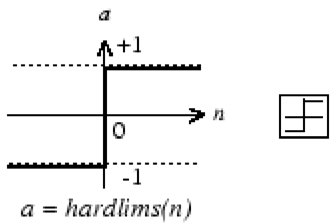
See Also

sim | hardlims

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';
```

More About

Algorithms

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

See Also

sim | hardlim

hextop

Hexagonal layer topology function

Syntax

```
hextop(dim1,dim2,...,dimN)
```

Description

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

`hextop(dim1,dim2,...,dimN)` takes `N` arguments,

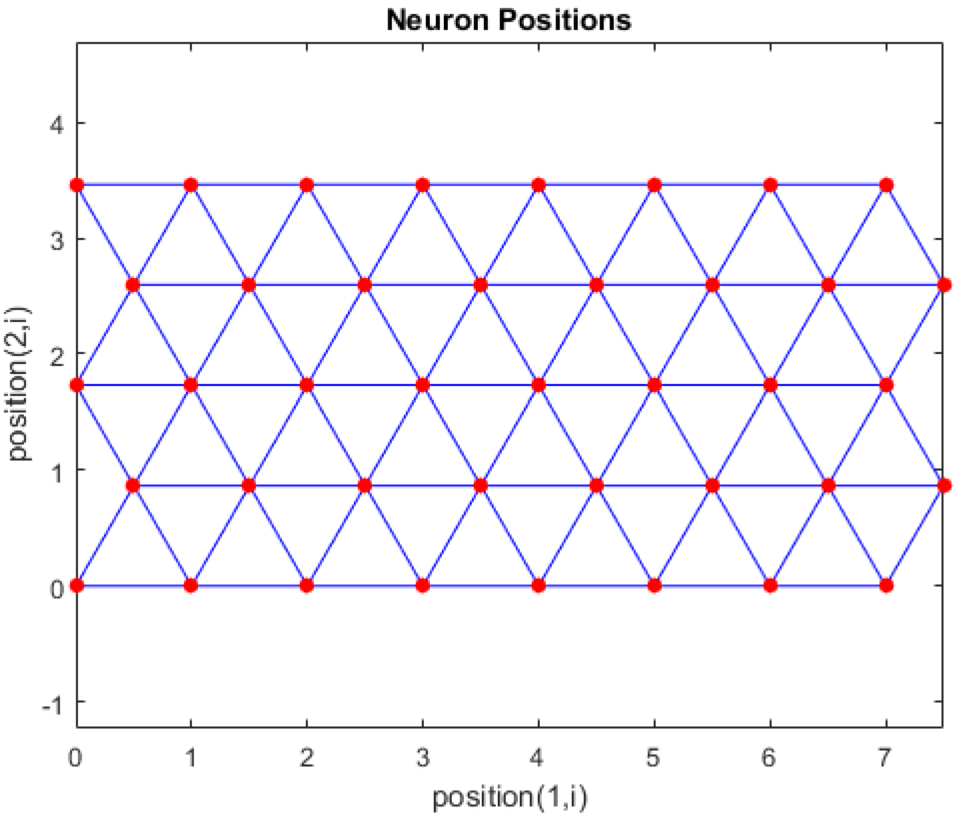
`dimi`	Length of layer in dimension `i`

and returns an `N`-by-`S` matrix of `N` coordinate vectors where `S` is the product of `dim1*dim2*...*dimN`.

Examples

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

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

vec2ind | sub2ind | ind2sub

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.

More About

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

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

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`.

More About

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`

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` string:

`'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`.

More About

Algorithms

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

See Also

`cascadeforwardnet` | `init` | `initnw` | `initwb` | `feedforwardnet`

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

`lvqnet` | `init`

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`.

More About

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` | `init` | `initlay` | `initwb` | `feedforwardnet`

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`.

More About

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`

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])
```

Network Use

You can create a standard network that uses `initzero` to initialize its weights by calling `newp` or `newlin`.

To prepare the weights and the bias of layer `i` of a custom network to be initialized with `midpoint`,

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 `'initzero'`.
4. Set each `net.layerWeights{i,j}.initFcn` to `'initzero'`.
5. Set each `net.biases{i}.initFcn` to `'initzero'`.

To initialize the network, call `init`.

See `help newp` and `help newlin` for initialization examples.

See Also

`initwb` | `initlay` | `init`

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`

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'`)

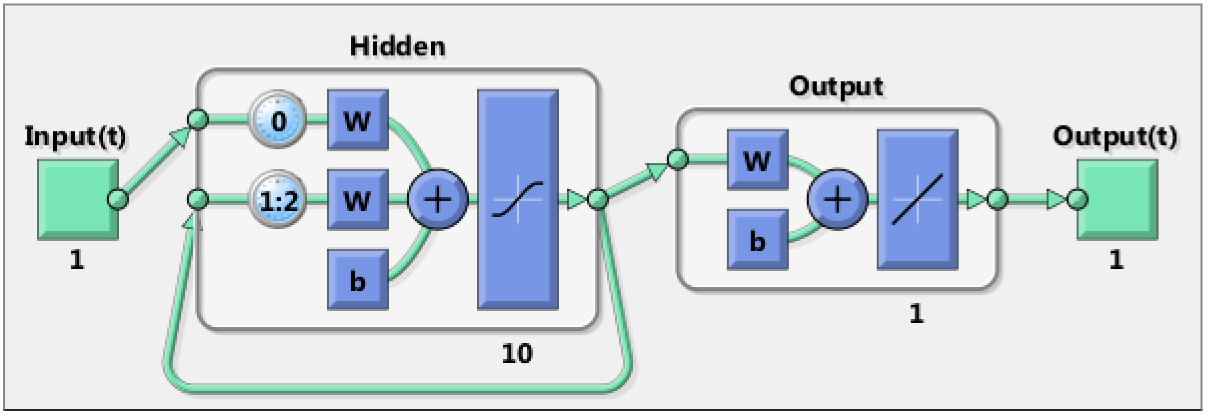
and returns a layer recurrent neural network.

Examples

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

preparets | removedelay | distdelaynet | timedelaynet | narnet | narxnet

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* string:

'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`).

More About

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

`learnk` | `learnos` | `adapt` | `train`

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* string:

'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,[])
```

Network Use

You can create a standard network that uses `learngd` with `newff`, `newcf`, or `newelm`. To prepare the weights and the bias of layer `i` of a custom network to adapt with `learngd`,

1. Set `net.adaptFcn` to `'trains'`. `net.adaptParam` automatically becomes `trains`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` to `'learngd'`. Set each `net.layerWeights{i,j}.learnFcn` to `'learngd'`. Set

net.biases{i}.learnFcn to 'learngd'. Each weight and bias learning parameter property is automatically set to learngd's default parameters.

To allow the network to adapt,

1 Set net.adaptParam properties to desired values.
2 Call adapt with the network.

See help newff or help newcf for examples.

More About

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

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` string:

`'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.

Network Use

You can create a standard network that uses `learngdm` with `newff`, `newcf`, or `newelm`.

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

1 Set `net.adaptFcn` to `'trains'`. `net.adaptParam` automatically becomes `trains`'s default parameters.

2 Set each `net.inputWeights{i,j}.learnFcn` to `'learngdm'`.
  Set each `net.layerWeights{i,j}.learnFcn` to `'learngdm'`. Set `net.biases{i}.learnFcn` to `'learngdm'`. Each weight and bias learning parameter property is automatically set to `learngdm`'s default parameters.

To allow the network to adapt,

1 Set `net.adaptParam` properties to desired values.
2 Call `adapt` with the network.

See `help newff` or `help newcf` for examples.

More About

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`

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` string:

`'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`).

More About

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

`learnhd` | `adapt` | `train`

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`* string:

`'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`).

More About

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

`learnh` | `adapt` | `train`

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* string:

`'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`).

More About

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

`learnk` | `learnos` | `adapt` | `train`

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* string:

'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`).

More About

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

`learnis` | `learnos` | `adapt` | `train`

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* string:

'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`).

More About

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

`learnlv2` | `adapt` | `train`

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* string:

'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`).

More About

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

`learnlv1` | `adapt` | `train`

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` string:

`'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`).

More About

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

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

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` string:

'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,[],[],[],[],[])
```

Network Use

You can create a standard network that uses learnp with newp.

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

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 'learnp'.
4 Set each net.layerWeights{i,j}.learnFcn to 'learnp'.
5 Set net.biases{i}.learnFcn to 'learnp'. (Each weight and bias learning parameter property automatically becomes the empty matrix, because learnp has no learning 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`).

See `help newp` for adaption and training examples.

More About

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`

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` string:

'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,[],[],[],[],[])
```

Network Use

You can create a standard network that uses learnpn with newp.

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

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 'learnpn'.
4. Set each net.layerWeights{i,j}.learnFcn to 'learnpn'.
5. Set net.biases{i}.learnFcn to 'learnpn'. (Each weight and bias learning parameter property automatically becomes the empty matrix, because learnpn has no learning 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`).

See `help newp` for adaption and training examples.

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.

More About

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`

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`* string:

'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)
```

Network Use

You can create a standard network that uses `learnsom` with `newsom`.

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 `'learnsom'`.
4. Set each `net.layerWeights{i,j}.learnFcn` to `'learnsom'`.
5. Set `net.biases{i}.learnFcn` to `'learnsom'`. (Each weight learning parameter property is automatically set to `learnsom`'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`).

More About

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

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* string:

`'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`).

More About

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`

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* string:

'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`).

More About

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`

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.

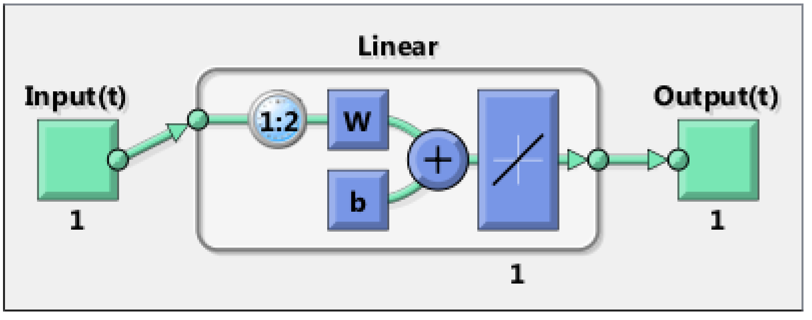
If the learning rate is too small, learning will happen very slowly. However, a greater danger is that it may be too large and learning will become unstable resulting in large changes to weight vectors and errors increasing instead of decreasing. If a data set is available which characterizes the relationship the layer is to learn, the maximum stable learning rate can be calculated with `maxlinlr`.

Examples

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

preparets | removedelay | timedelaynet | narnet | narxnet

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`.

More About

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
```

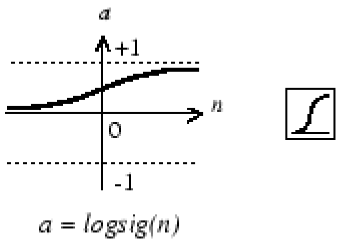
See Also

dist | mandist | selforgmap | sim

logsig

Log-sigmoid transfer function

Graph and Symbol



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* string:

`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';
```

More About

Algorithms

```
logsig(n) = 1 / (1 + exp(-n))
```

See Also

`sim` | `tansig`

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.

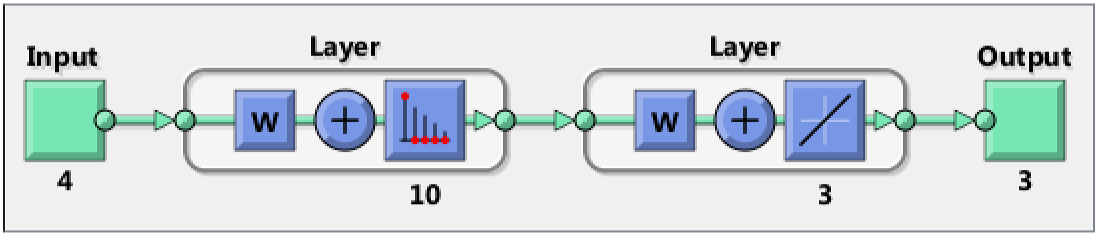
The other option for the `lvq` learning function is `learnlv2`.

Examples

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

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

`lvqnet` | `initlvq`

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`* string:

`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

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.

More About

Algorithms

The Manhattan distance `D` between two vectors `X` and `Y` is

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

See Also

`dist` | `linkdist` | `sim`

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

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.

More About

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`

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

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.

More About

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`

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`

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`

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`

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])
```

Network Use

You can create a standard network that uses `midpoint` to initialize weights by calling `newc`.

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

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 `'midpoint'`. Set each `net.layerWeights{i,j}.initFcn` to `'midpoint'`.

To initialize the network, call `init`.

See Also

`initwb` | `initlay` | `init`

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 mth matrix is an Ri-by-2 matrix of the minimum and maximum values of elements for the matrix on the ith 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)
```

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

Here a two-layer feedforward network is created and trained to predict median house prices using the `mse` performance function and a regularization value of 0.01, which is the default performance function for `feedforwardnet`.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net.performFcn = 'mse';  % Redundant, MSE is default
net.performParam.regularization = 0.01;
net = train(net,x,t);
y = net(x);
perf  = perform(net,t,y);
```

Alternately, you can call this function directly.

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

See Also
`mae`

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'`)

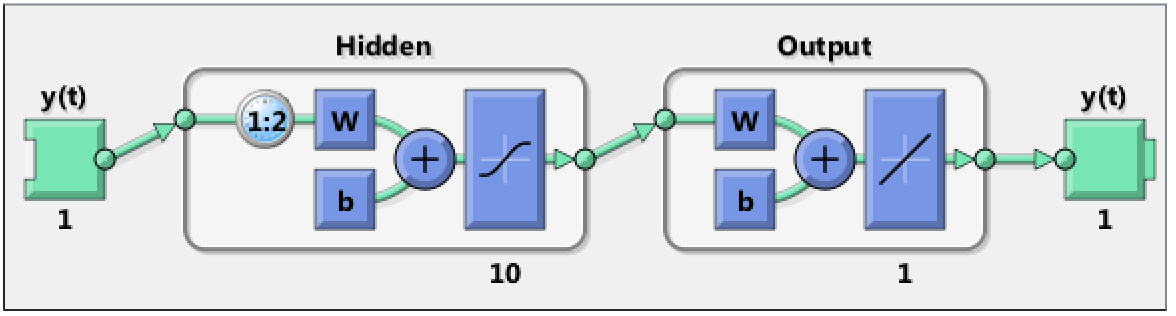
and returns a NAR neural network.

Examples

Here a NAR network is used to solve a simple time series problem.

```
T = simplenar_dataset;
net = narnet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,{},{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi);
perf = perform(net,Ts,Y)
```

```
perf =

   1.0100e-09
```



See Also

preparets | removedelay | timedelaynet | narnet | narxnet

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'`)

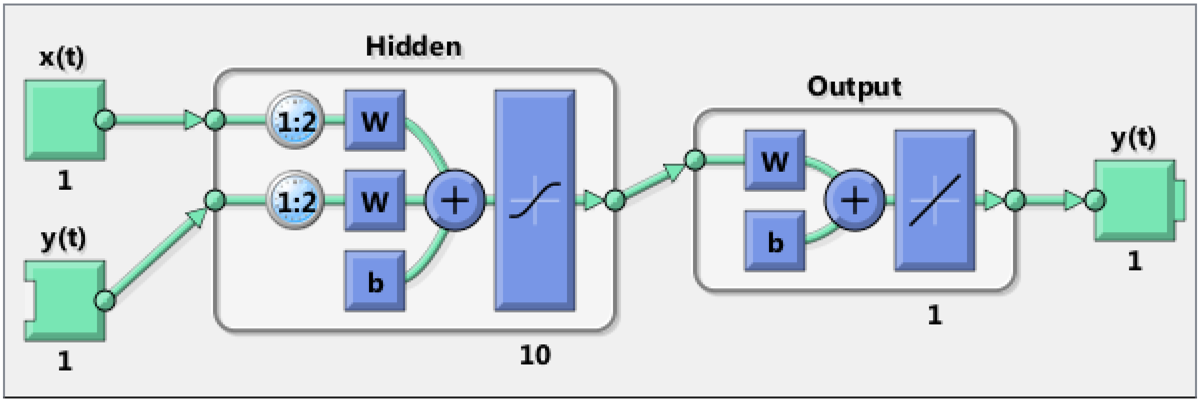
and returns a NARX neural network.

Examples

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

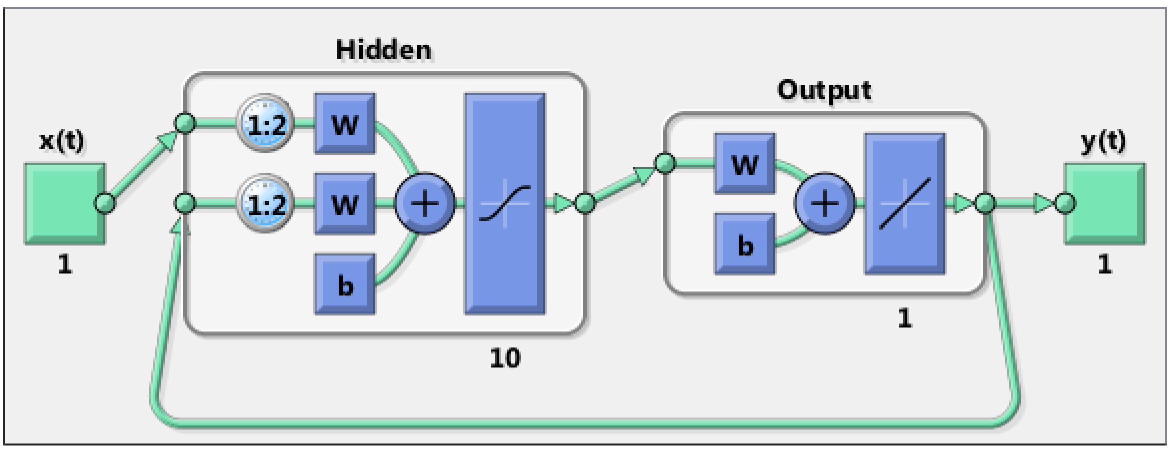
```
[X,T] = simpleseries_dataset;
net = narxnet(1:2,1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai);
```

```
perf = perform(net,Ts,Y)

perf =

    0.0192
```



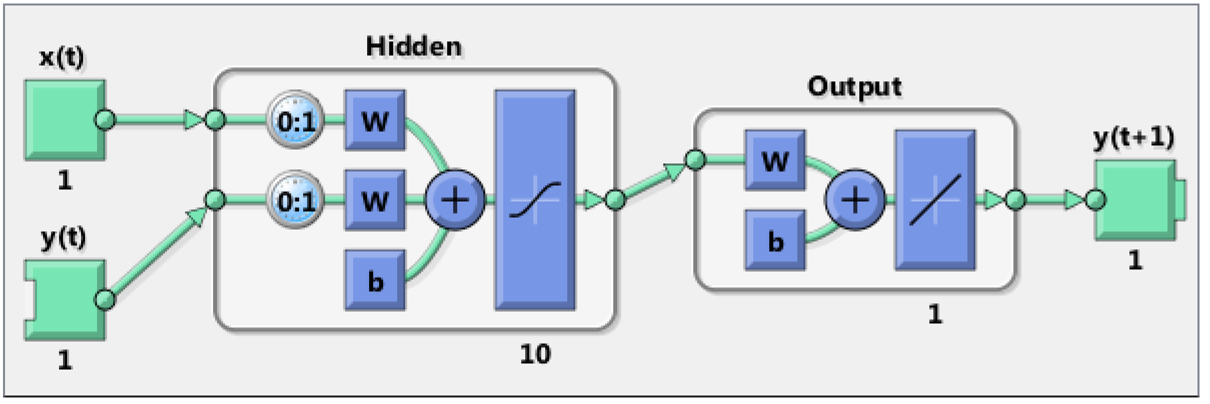
Here the NARX network is simulated in closed loop form.

```
netc = closeloop(net);
view(netc)
[Xs,Xi,Ai,Ts] = preparets(netc,X,{},T);
y = netc(Xs,Xi,Ai);
```



Here the NARX network is used to predict the next output, a timestep ahead of when it will actually appear.

```
netp = removedelay(net);
view(netp)
[Xs,Xi,Ai,Ts] = preparets(netp,X,{},T);
y = netp(Xs,Xi,Ai);
```



See Also

closeloop | narnet | openloop | preparets | removedelay | timedelaynet

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”.

More About

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`

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`.

More About

Algorithms

`negdist` returns the negative Euclidean distance:

```
z = -sqrt(sum(w-p)^2)
```

See Also

`competlayer` | `sim` | `dist` | `dotprod` | `selforgmap`

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

`tansig` | `logsig`

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

`sim` | `netsum` | `concur`

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 layer i of a network.

```
net.layers(i).netFcn = 'compet';
```

Use feedforwardnet or cascadeforwardnet to create a standard network that uses netsum.

See Also

cascadeforwardnet | netprod | netinv | feedforwardnet

network

Create custom neural network

Syntax

```
net = network
net =
network(numInputs,numLayers,biasConnect,inputConnect,layerConnect,outputConnec
```

To Get Help

Type `help network/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,outputConnec`
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 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)
```

More About

- "Neural Network Object Properties"
- "Neural Network Subobject Properties"

See Also

sim

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

sim | newrb | newrbe | newpnn

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,:)
```

More About

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`

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)
```

More About

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

`sim` | `ind2vec` | `vec2ind` | `newrb` | `newrbe` | `newgrnn`

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)
```

More About

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

`sim` | `newrbe` | `newgrnn` | `newpnn`

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)
```

More About

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

sim | newrb | newgrnn | newpnn

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 Neural Network”.

More About

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`

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`

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`

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

nnsize | tonndata | fromnndata | nndata2sim | sim2nndata

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

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` | `sim2nndata` | `nnsize`

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`

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`

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`

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.

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

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`

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.

More About

Algorithms

`normprod` returns the dot product normalized by the sum of the input vector elements.

```
z = w*p/sum(p)
```

See Also

`dotprod`

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`

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 Neural Network".

More About

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`

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 "Neural Network Time Series Prediction and Modeling".

`ntstool('close')` closes the tool.

See Also

`nctool` | `nftool` | `nprtool`

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

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.

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

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, `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`

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

`nndata` | `nnsize` | `getelements` | `setelements` | `catelements` | `numsamples` | `numsignals` | `numtimesteps`

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

`numnan` | `nndata` | `nnsize`

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

numnan | nndata | nnsize

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

`nndata` | `nnsize` | `getsamples` | `setsamples` | `catsamples` | `numelements` | `numsignals` | `numtimesteps`

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

`nndata` | `nnsize` | `getsignals` | `setsignals` | `catsignals` | `numelements` | `numsamples` | `numtimesteps`

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

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

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`

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.

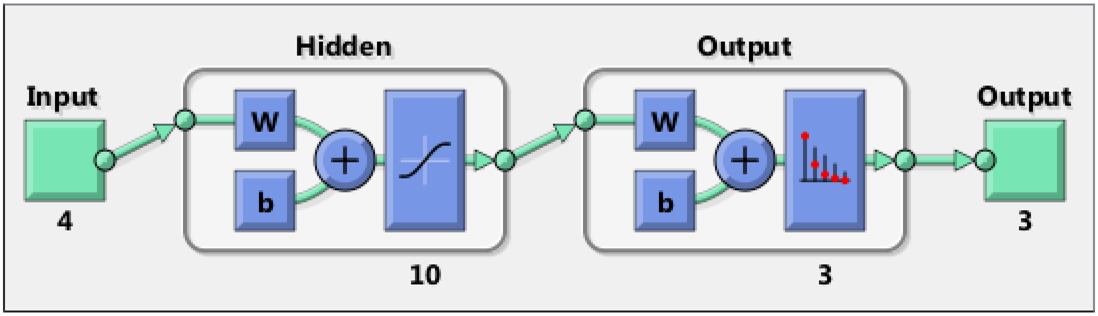
Examples

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);
```



More About

- "Classify Patterns with a Neural Network"
- "Neural Network Object Properties"
- "Neural Network Subobject Properties"

See Also

competlayer | lvqnet | network | nprtool | selforgmap

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.

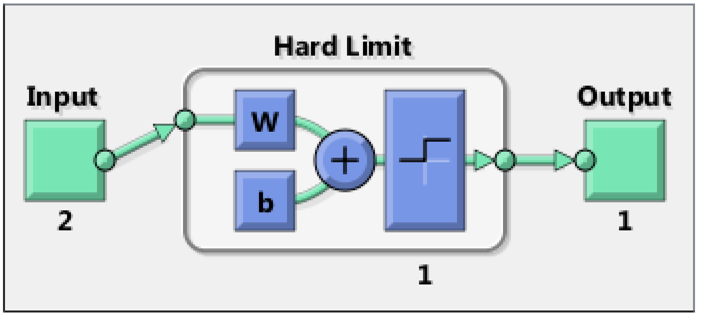
In addition to the default hard limit transfer function, perceptrons can be created with the `hardlims` transfer function. The other option for the perceptron learning rule is `learnpn`.

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

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

preparets | removedelay | patternnet | timedelaynet | narnet | narxnet

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 dimension be 1. This gives the flexibilty 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

train | configure | init

plotconfusion

Plot classification confusion matrix

Syntax

```
plotconfusion(targets,outputs)
plotconfusion(targets1,outputs1,'name1',targets2,outputs2,'name2',...)
```

Description

`plotconfusion(targets,outputs)` takes target and output data and generates a confusion plot. The target data are ground truth labels in 1-of-N form (in each column, a single element is 1 to indicate the correct class, and all other elements are 0). The output data are the outputs from a neural network that performs classification. They can either be in 1-of-N form, or may also be probabilities where each column sums to 1.

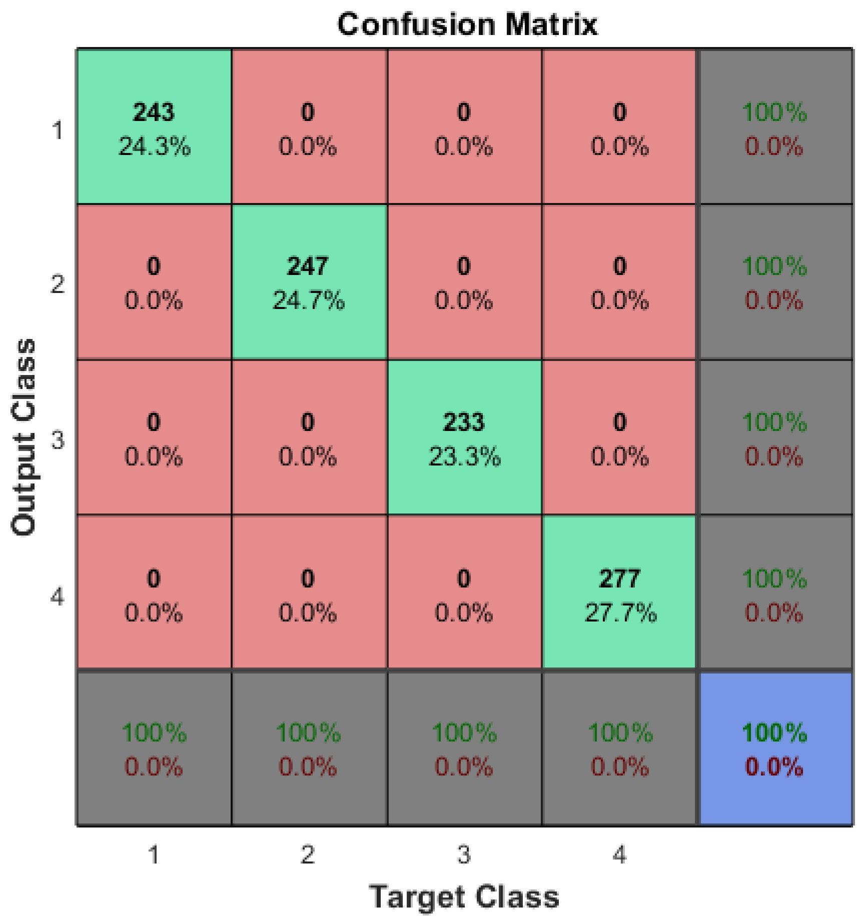
`plotconfusion(targets1,outputs1,'name1',targets2,outputs2,'name2',...)` generates several confusion plots in one figure, and prefixes the character strings specified by the `'name'` arguments to the titles of the appropriate plots.

On the confusion matrix plot, the rows show the predicted class, and the columns show the true class. The diagonal cells show where the true class and predicted class match. The off diagonal cells show instances where the classifier has made mistakes. The column on the right hand side of the plot shows the accuracy for each predicted class, while the row at the bottom of the plot shows the accuracy for each true class. The cell in the bottom right of the plot shows the overall accuracy.

Examples

This example shows how to train a pattern recognition network and plot its accuracy.

```
[x,t] = simpleclass_dataset;
net = patternnet(10);
net = train(net,x,t);
y = net(x);
plotconfusion(t,y)
```


Confusion Matrix
243
24.3%
0
0.0%
0
0.0%
0
0.0%
100%
0.0%
0
0.0%
247
24.7%
0
0.0%
0
0.0%
100%
0.0%
0
0.0%
0
0.0%
233
23.3%
0
0.0%
100%
0.0%
0
0.0%
0
0.0%
0
0.0%
277
27.7%
100%
0.0%
100%
0.0%
100%
0.0%
100%
0.0%
100%
0.0%
100%
0.0%
1
2
3
4
Output Class
Target Class

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`

ploterrcorr

Plot autocorrelation of error time series

Syntax

```
ploterrcorr(error)
ploterrcorr(errors,'outputIndex',outIdx)
```

Description

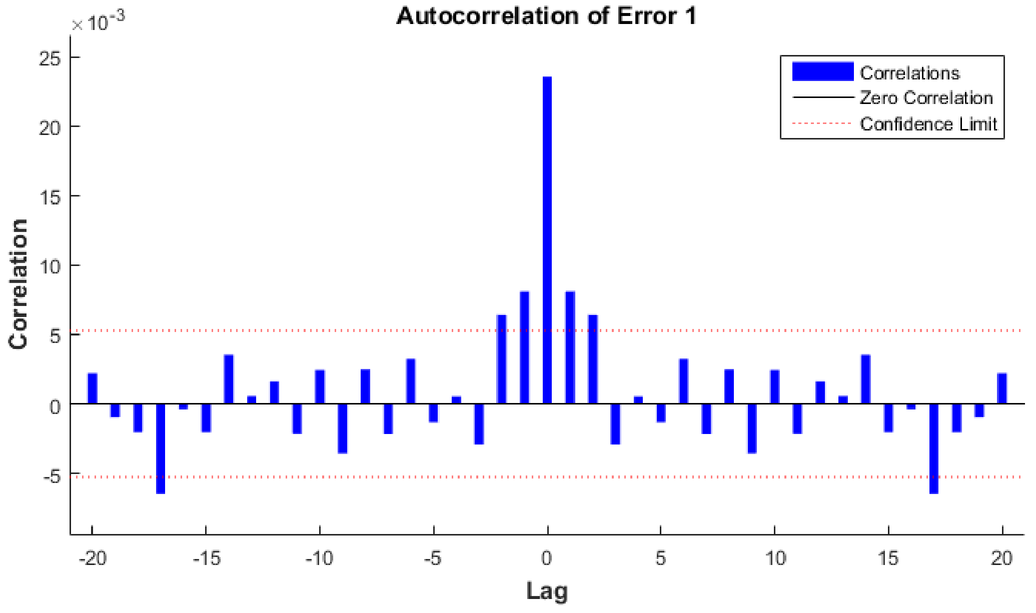
`ploterrcorr(error)` takes an error time series and plots the autocorrelation of errors across varying lags.

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

Examples

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`

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`.

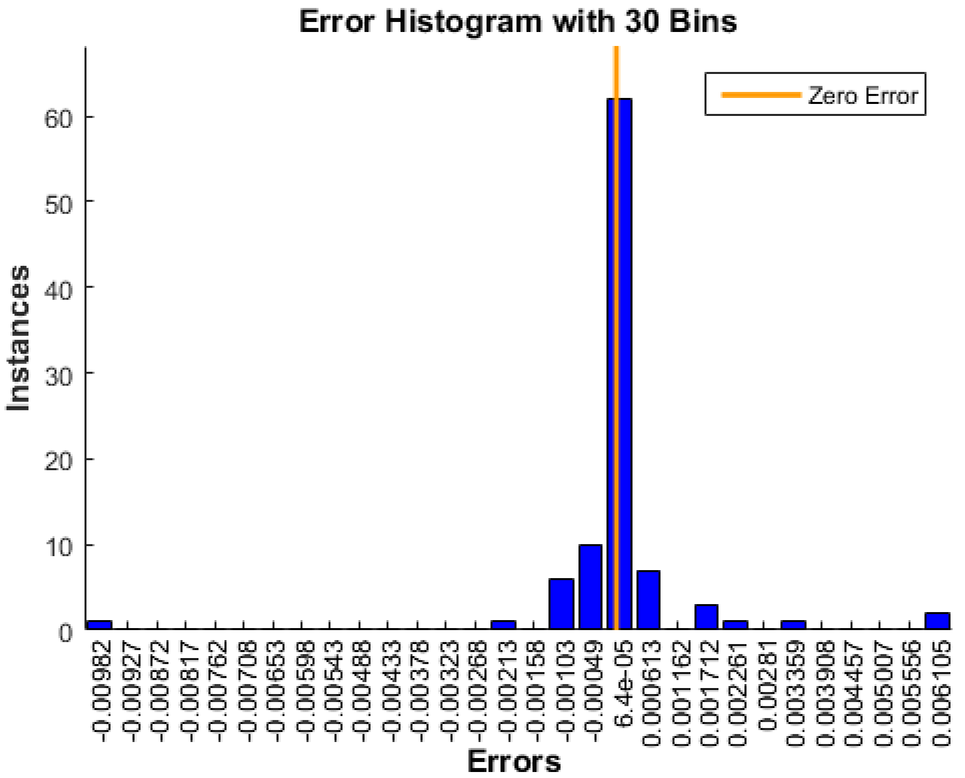
`ploterrhist(e1,'name1',e2,'name2',...)` takes any number of errors and names and plots each pair.

`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

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`

plotes

Plot error surface of single-input neuron

Syntax

```
plotes(WV,BV,ES,V)
```

Description

`plotes(WV,BV,ES,V)` takes these arguments,

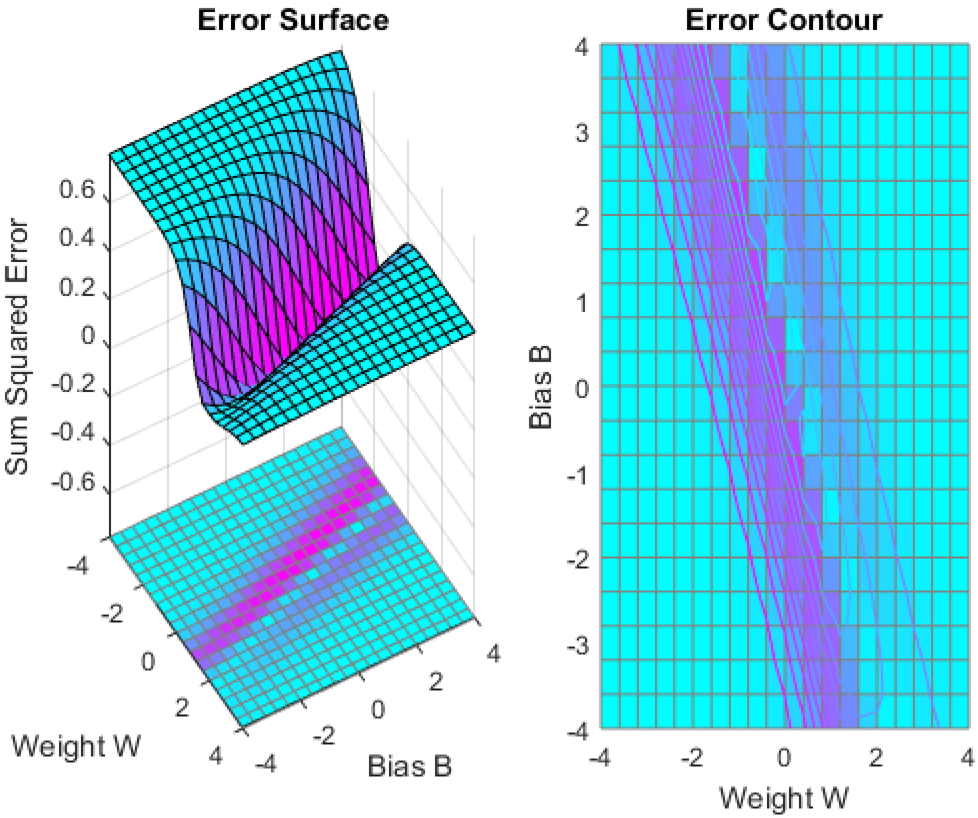
WV	1-by-N row vector of values of W
BV	1-by-M row vector of values of B
ES	M-by-N matrix of error vectors
V	View (default = [-37.5, 30])

and plots the error surface with a contour underneath.

Calculate the error surface ES with `errsurf`.

Examples

```
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`

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 `inputs`.

The plot appears only for networks with one input.

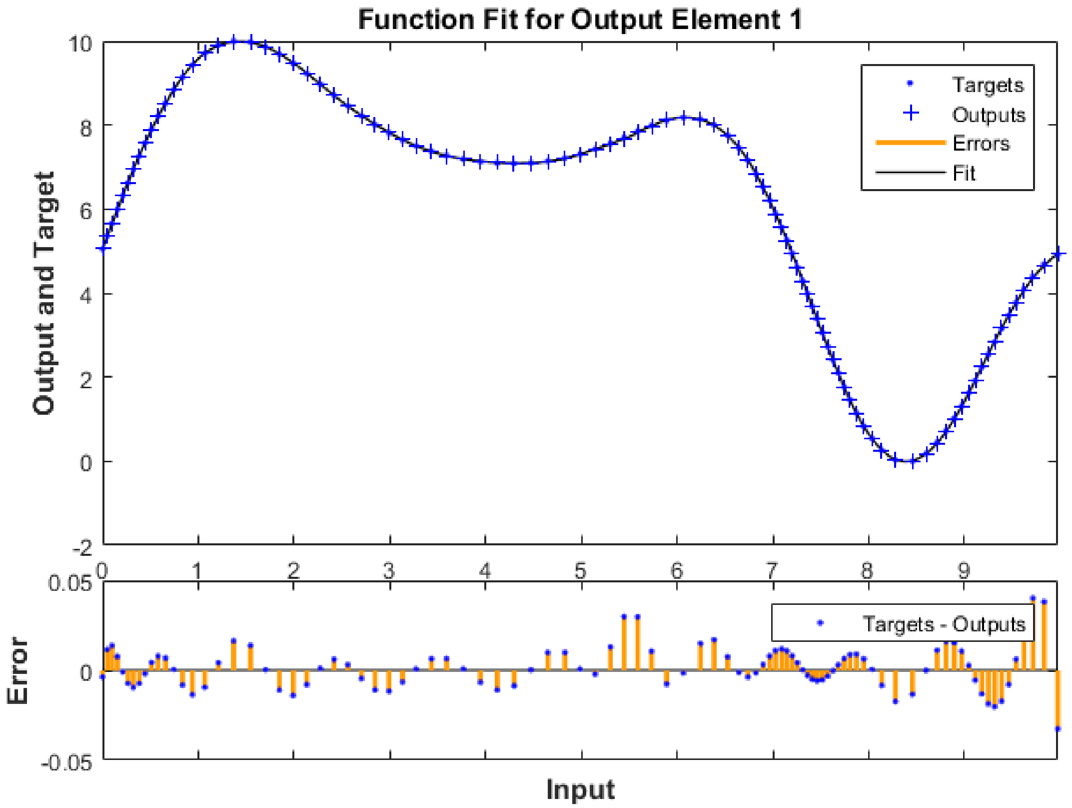
Only the first output/targets appear if the network has more than one output.

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

Examples

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`

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.

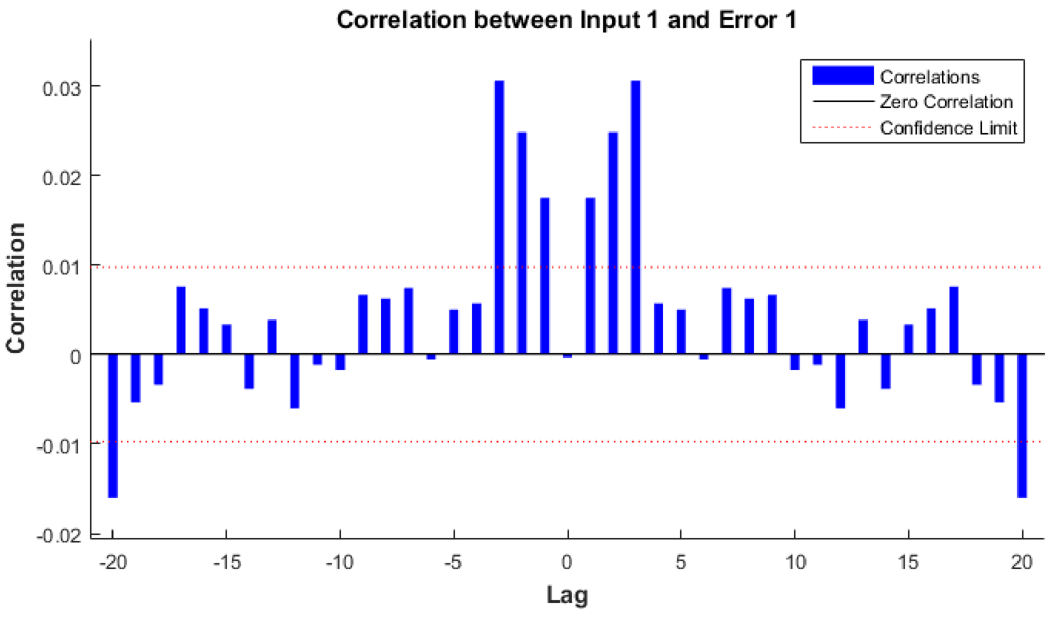
`plotinerrcorr(...,'inputIndex',inputIndex)` optionally defines which input element is being correlated and plotted. The default is 1.

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

Examples

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` | `plotresponse` | `ploterrhist`

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

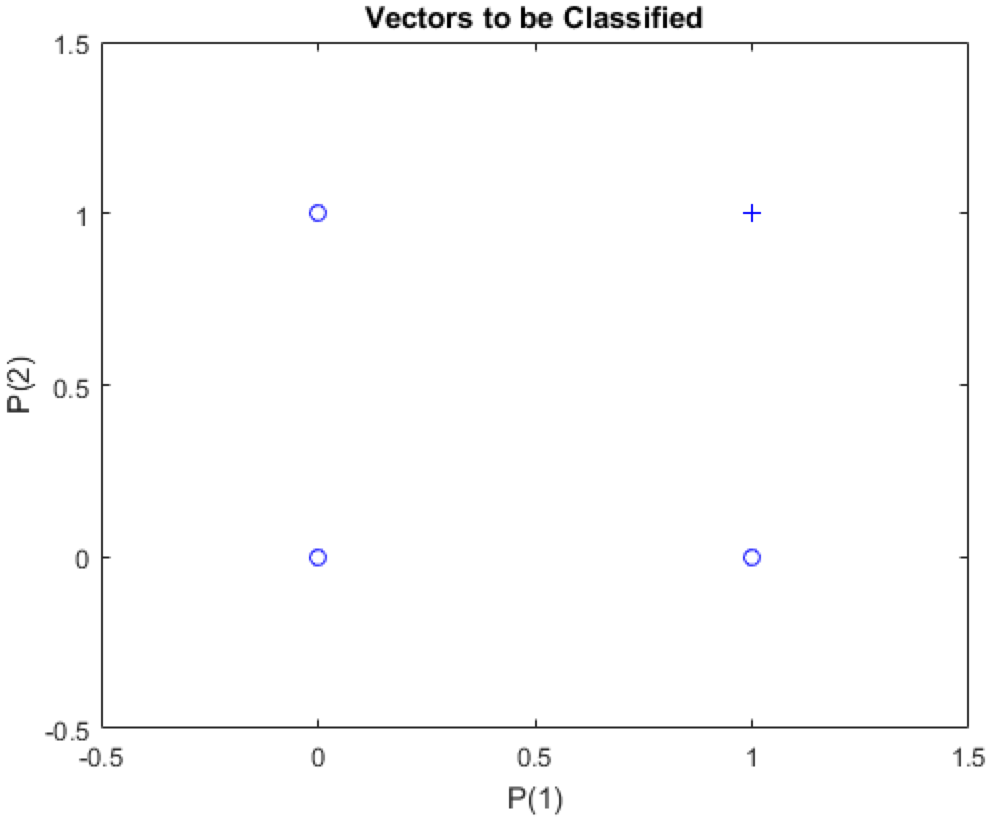
and deletes the last line before plotting the new one.

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

Examples

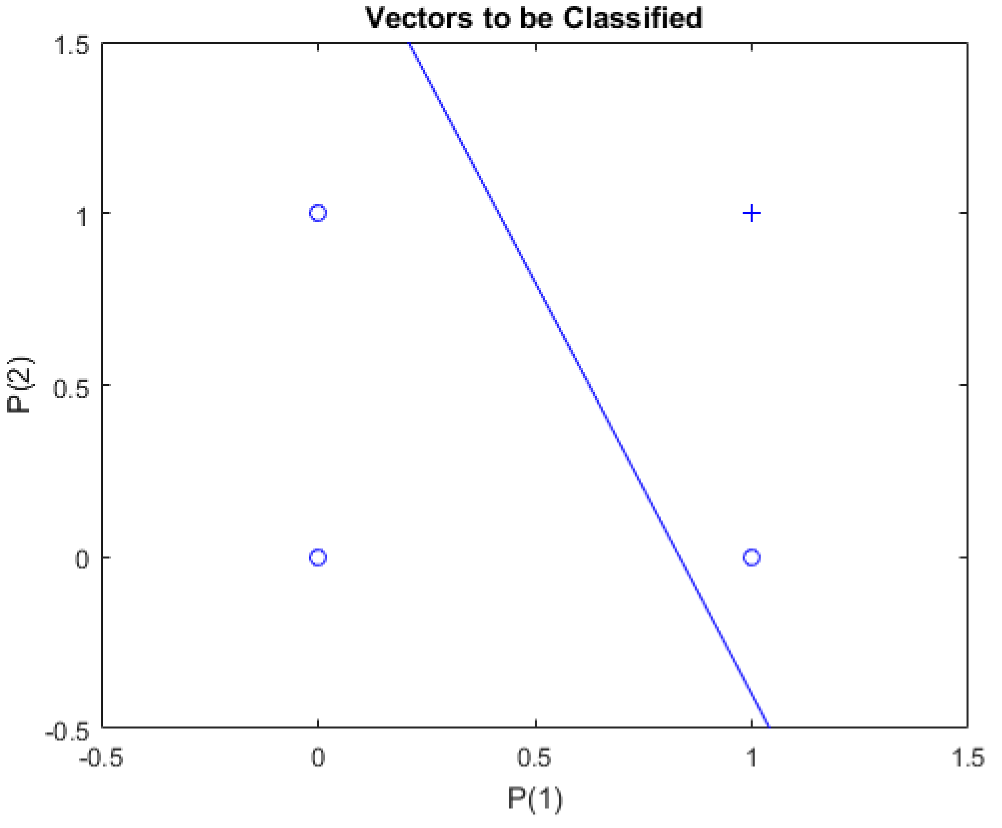
The code below defines and plots the inputs and targets for a perceptron:

```
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`

plotperform

Plot network performance

Syntax

```
plotperform(TR)
```

Description

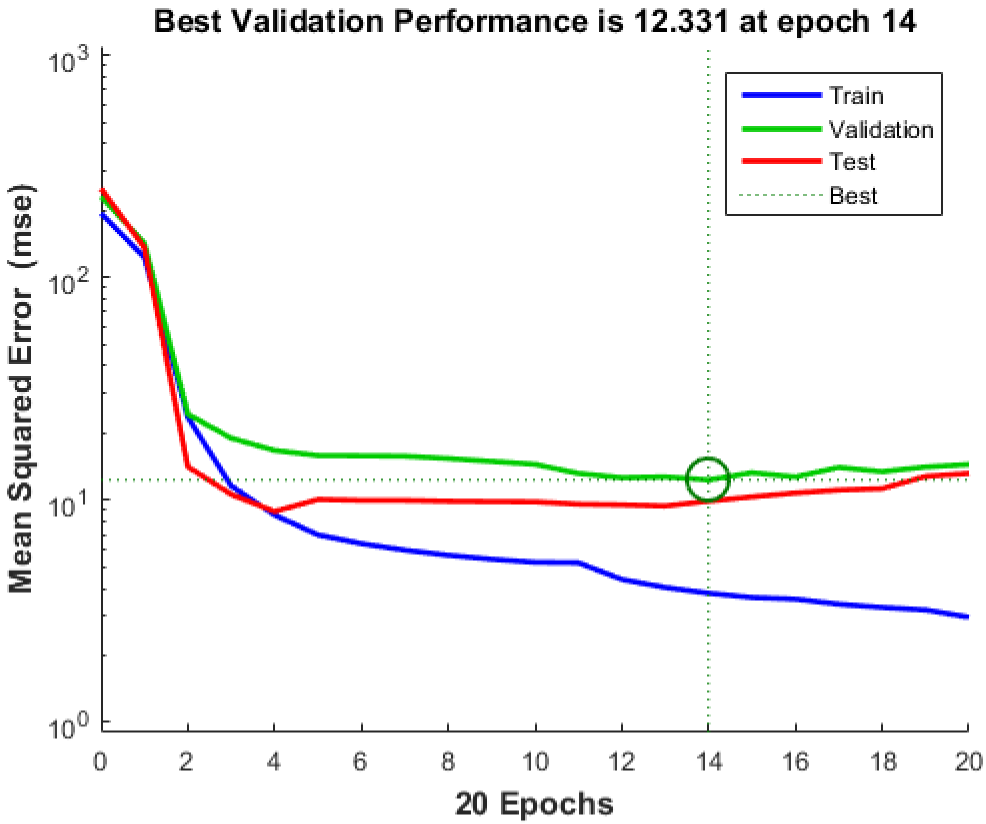
`plotperform(TR)` plots error vs. epoch for the training, validation, and test performances of the training record `TR` returned by the function `train`.

Examples

Plot Performances

This example shows how to use `plotperform` to obtain a plot of training record error values against the number of training epochs.

```
[x,t] = house_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`

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,

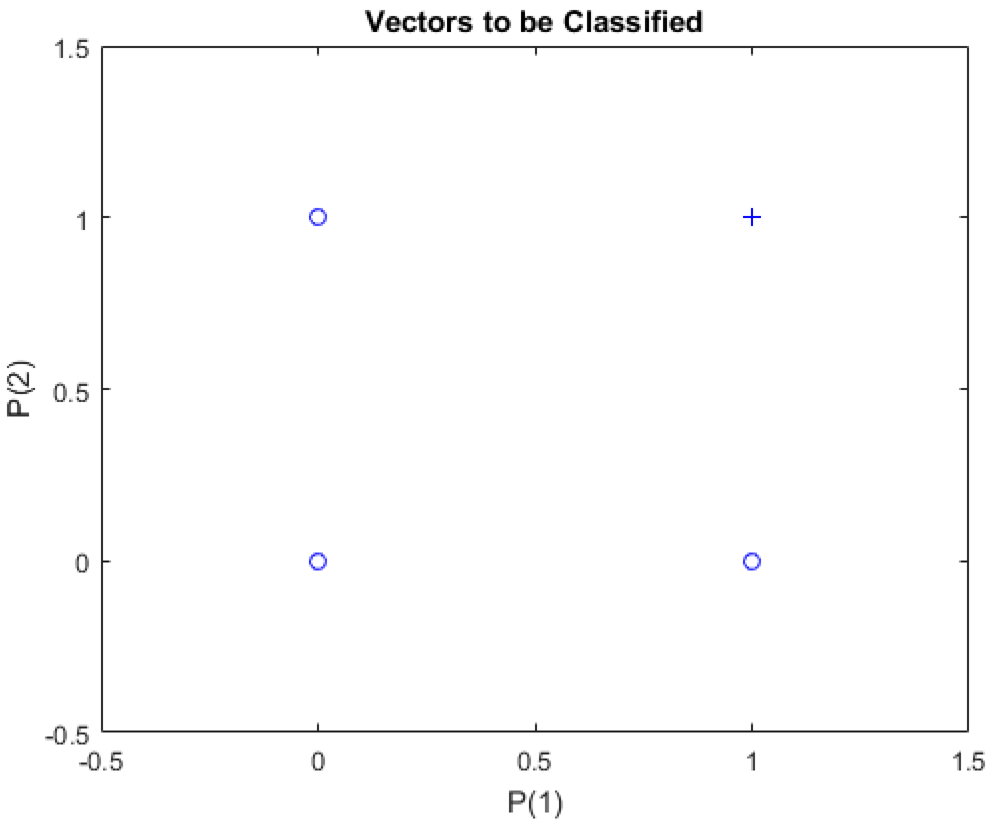
V	Graph limits = `[x_min x_max y_min y_max]`

and plots the column vectors with limits set by V.

Examples

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`

plotregression

Plot linear regression

Syntax

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

Description

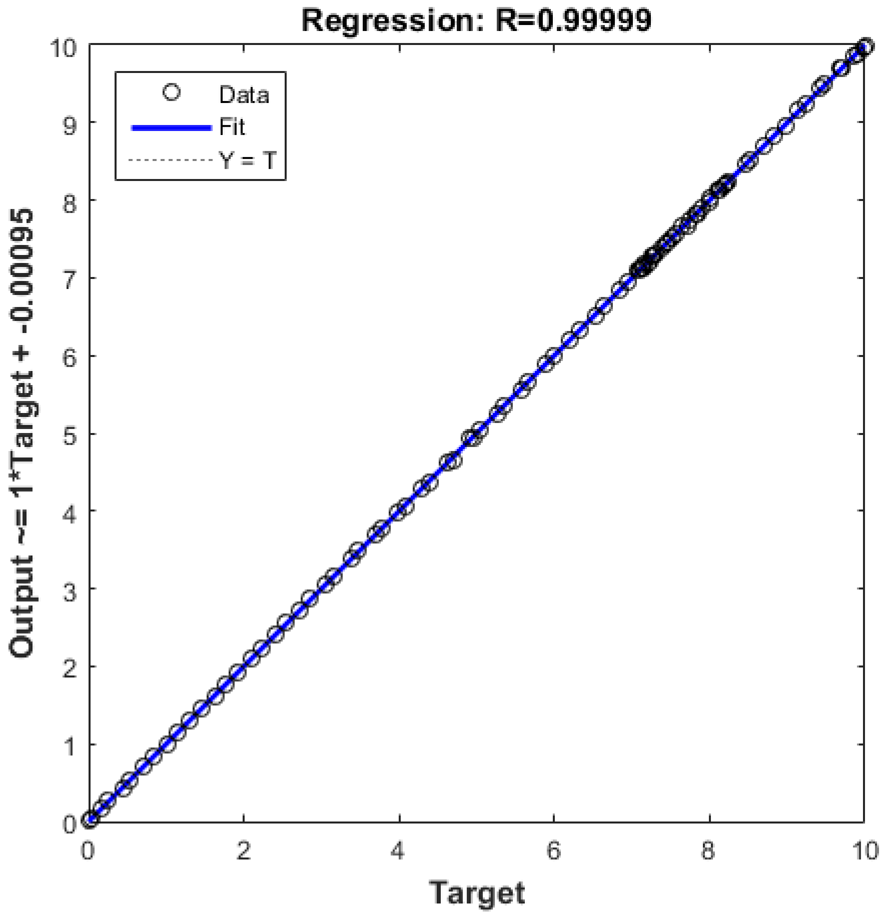
`plotregression(targets,outputs)` plots the linear regression of `targets` relative to `outputs`.

`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`

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.

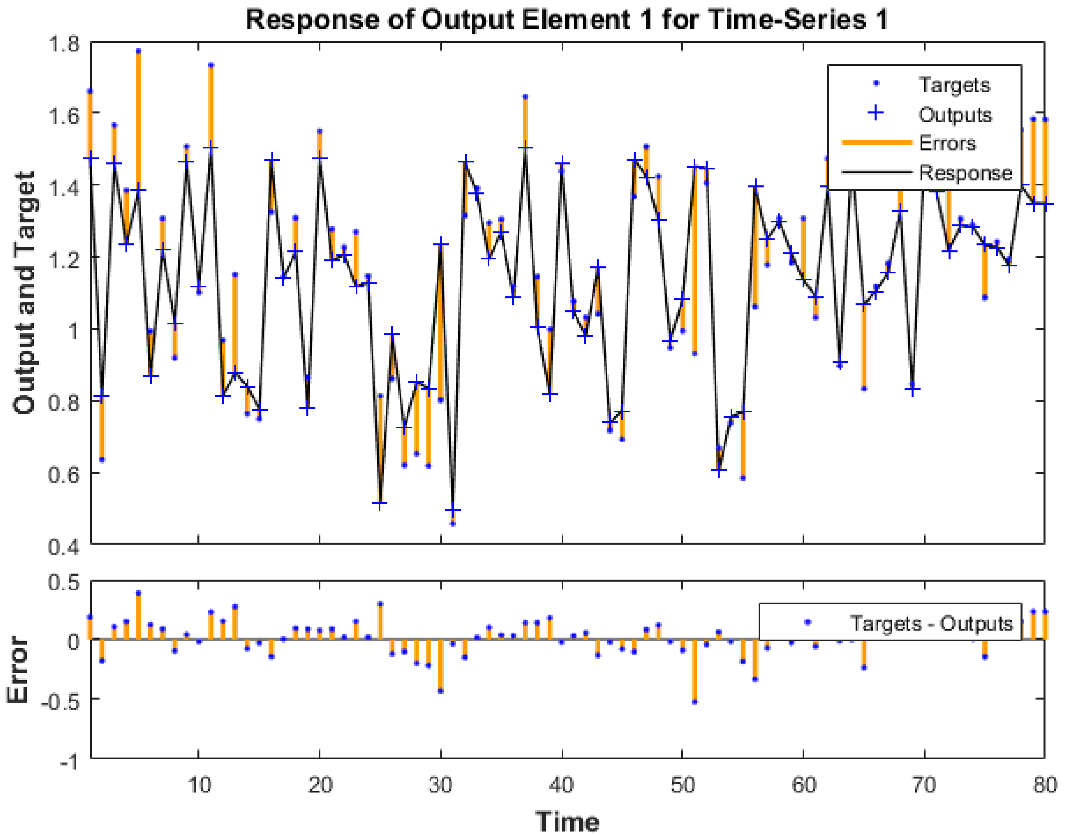
`plotresponse(t1,'name',t2,'name2',...,y)` takes multiple target/name pairs, typically defining training, validation and testing targets, and the output. It plots the responses with colors indicating the different target sets.

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

Examples

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` | `plotinerrcorr` | `ploterrhist`

plotroc

Plot receiver operating characteristic

Syntax

```
plotroc(targets,outputs)
plotroc(targets1,outputs2,'name1',...)
```

Description

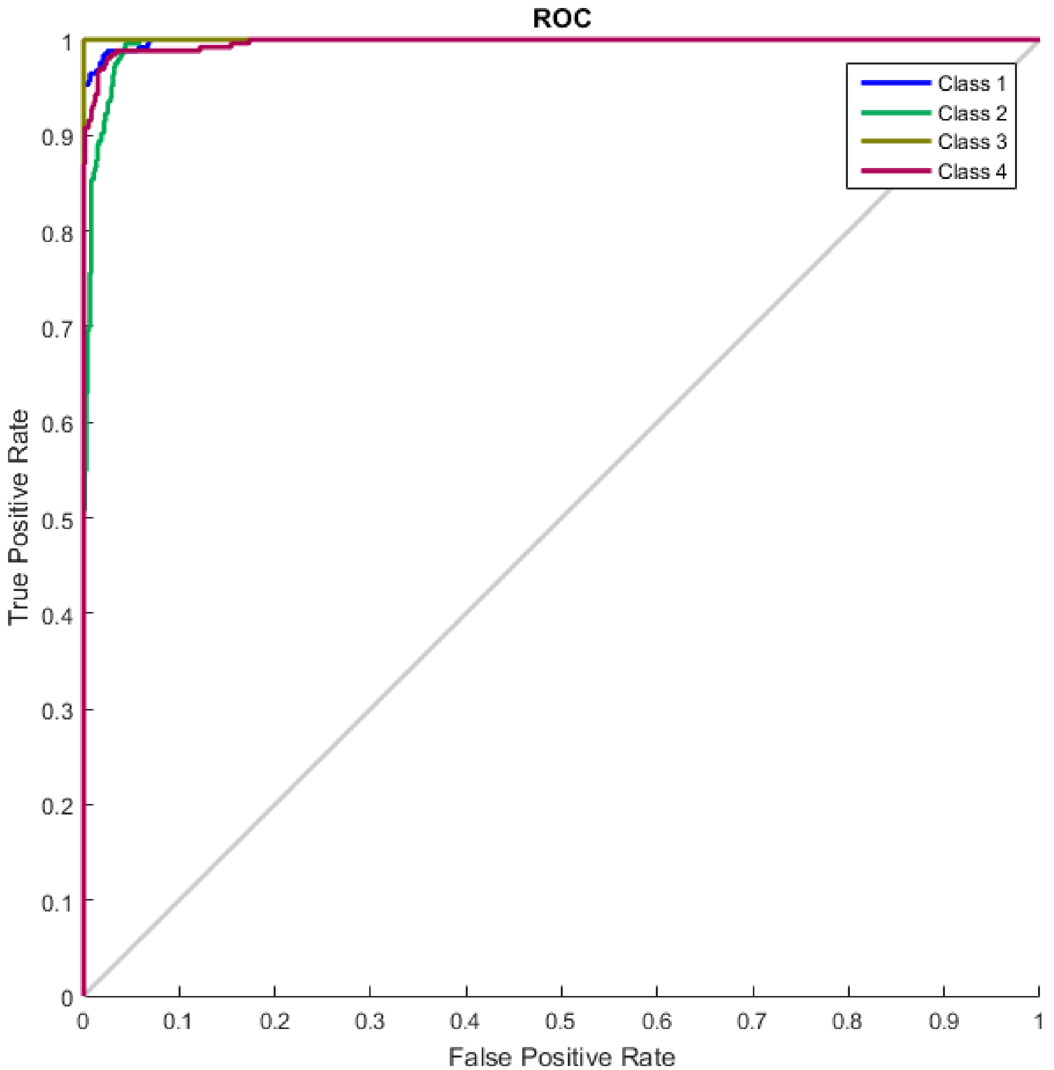
`plotroc(targets,outputs)` plots the receiver operating characteristic for each output class. The more each curve hugs the left and top edges of the plot, the better the classification.

`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
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Class 1
Class 2
Class 3
Class 4

See Also

`roc`

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,

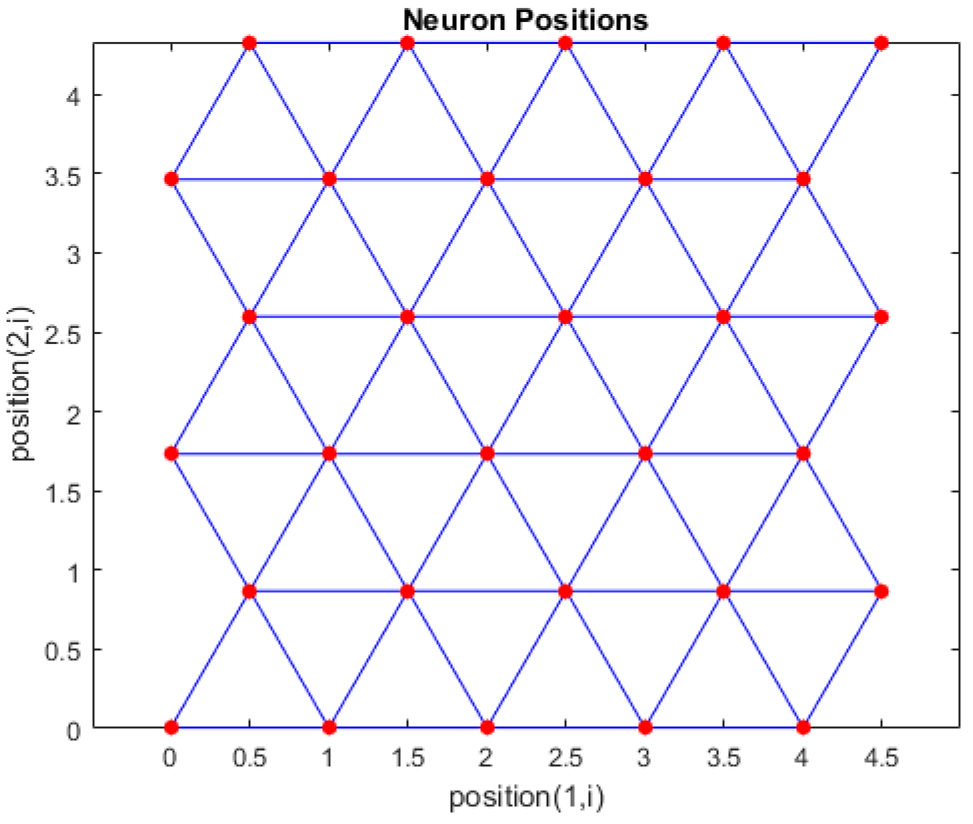
W	S-by-R weight matrix
D	S-by-S distance matrix
ND	Neighborhood distance (default = 1)

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

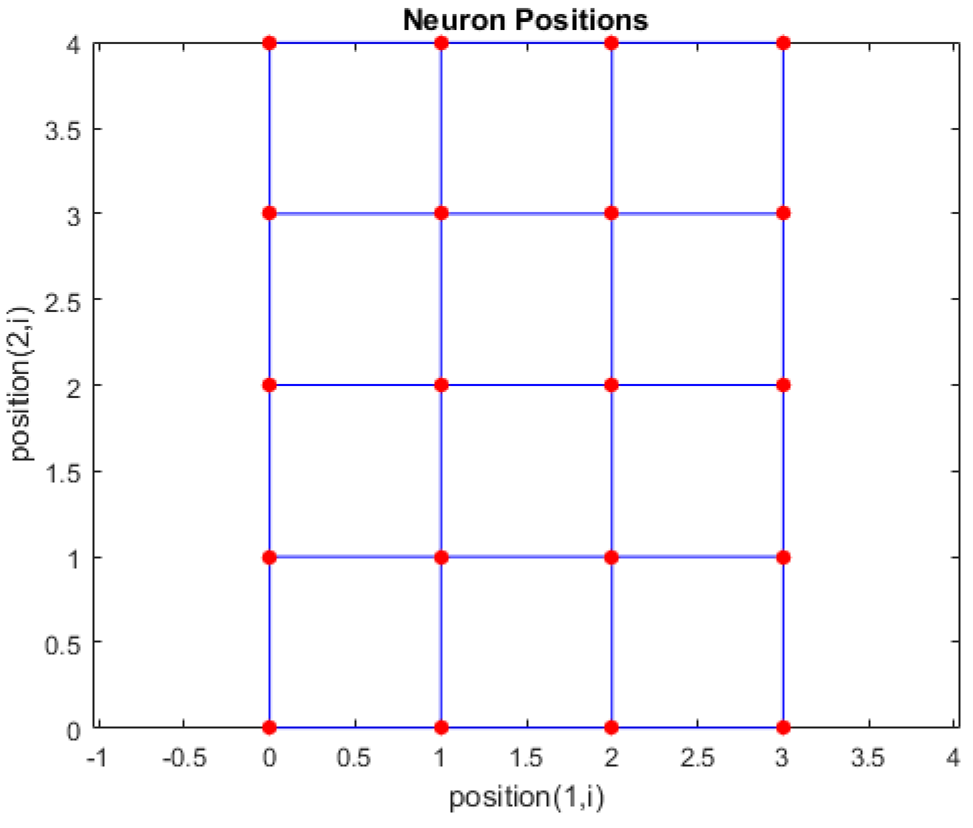
Examples

These examples generate plots of various layer topologies.

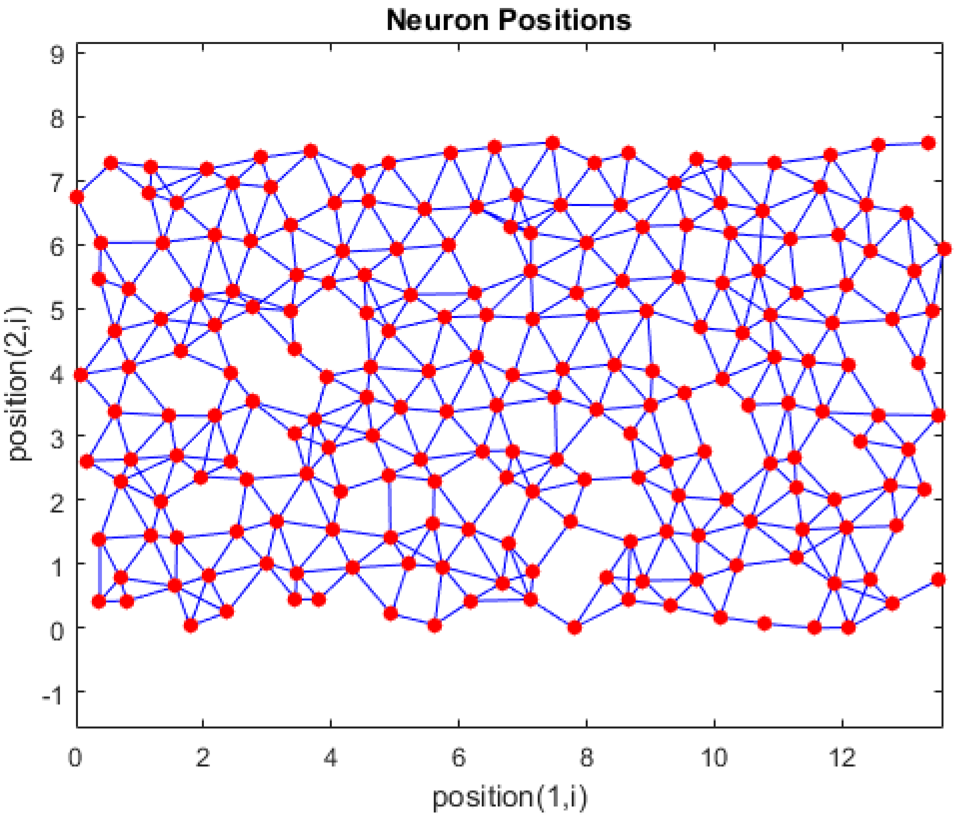
```
pos = hextop(5,6);
plotsom(pos)
```



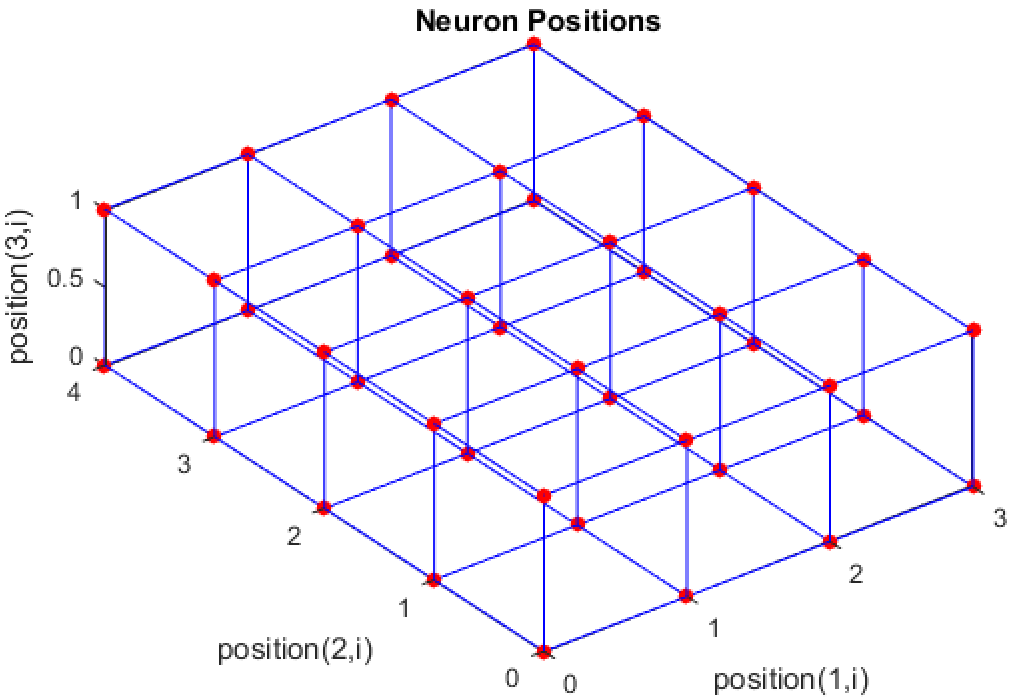

```
pos = gridtop(4,5);
plotsom(pos)
```



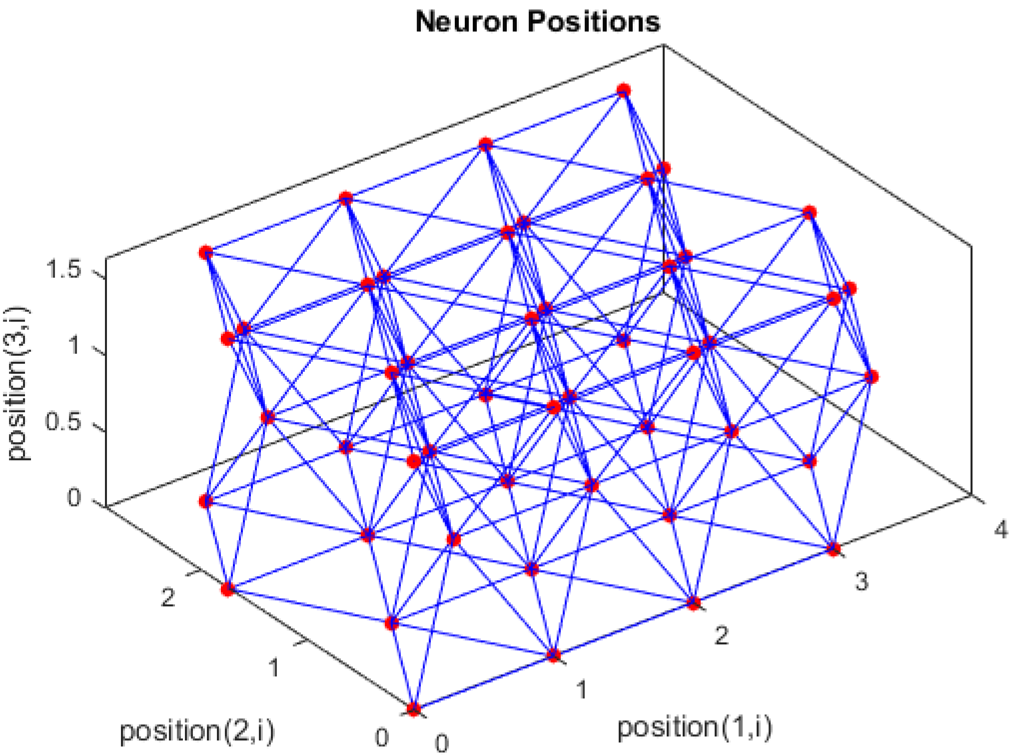

```
pos = randtop(18,12);
plotsom(pos)
```




```
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`

plotsomhits

Plot self-organizing map sample hits

Syntax

```
plotsomhits(net,inputs)
```

Description

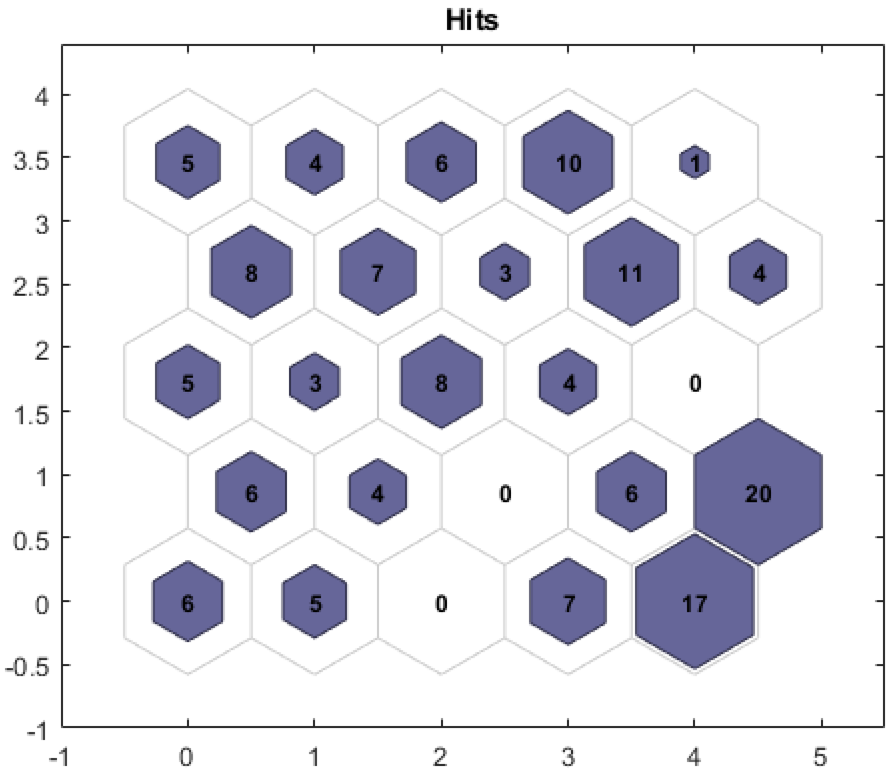
`plotsomhits(net,inputs)` plots a `SOM` layer, with each neuron showing the number of input vectors that it classifies. The relative number of vectors for each neuron is shown via the size of a colored patch.

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`

plotsomnc

Plot self-organizing map neighbor connections

Syntax

```
plotsomnc(net)
```

Description

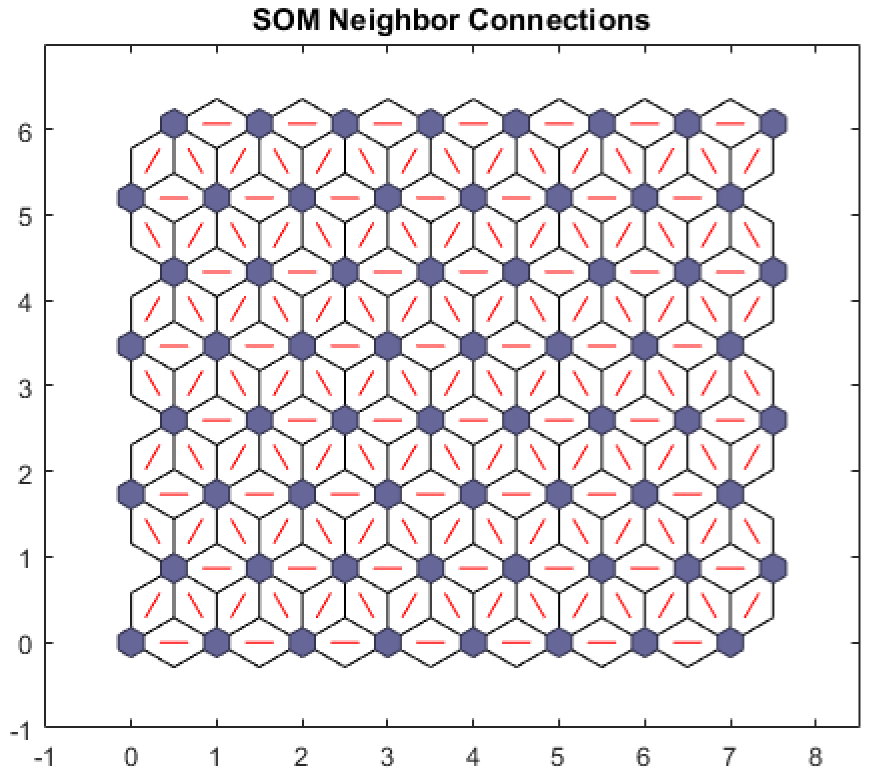
`plotsomnc(net)` plots a `SOM` layer showing neurons as gray-blue patches and their direct neighbor relations with red lines.

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

`plotsomnd` | `plotsomplanes` | `plotsomhits`

plotsomnd

Plot self-organizing map neighbor distances

Syntax

```
plotsomnd(net)
```

Description

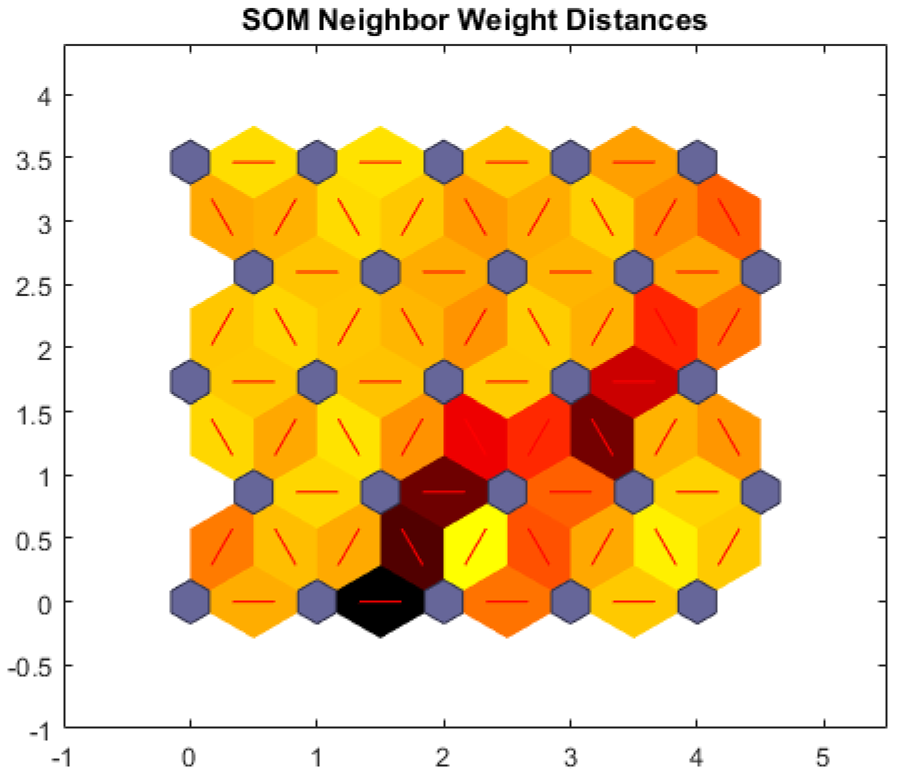
`plotsomnd(net)` plots a `SOM` layer showing neurons as gray-blue patches and their direct neighbor relations with red lines. The neighbor patches are colored from black to yellow to show how close each neuron's weight vector is to its neighbors.

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`

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.

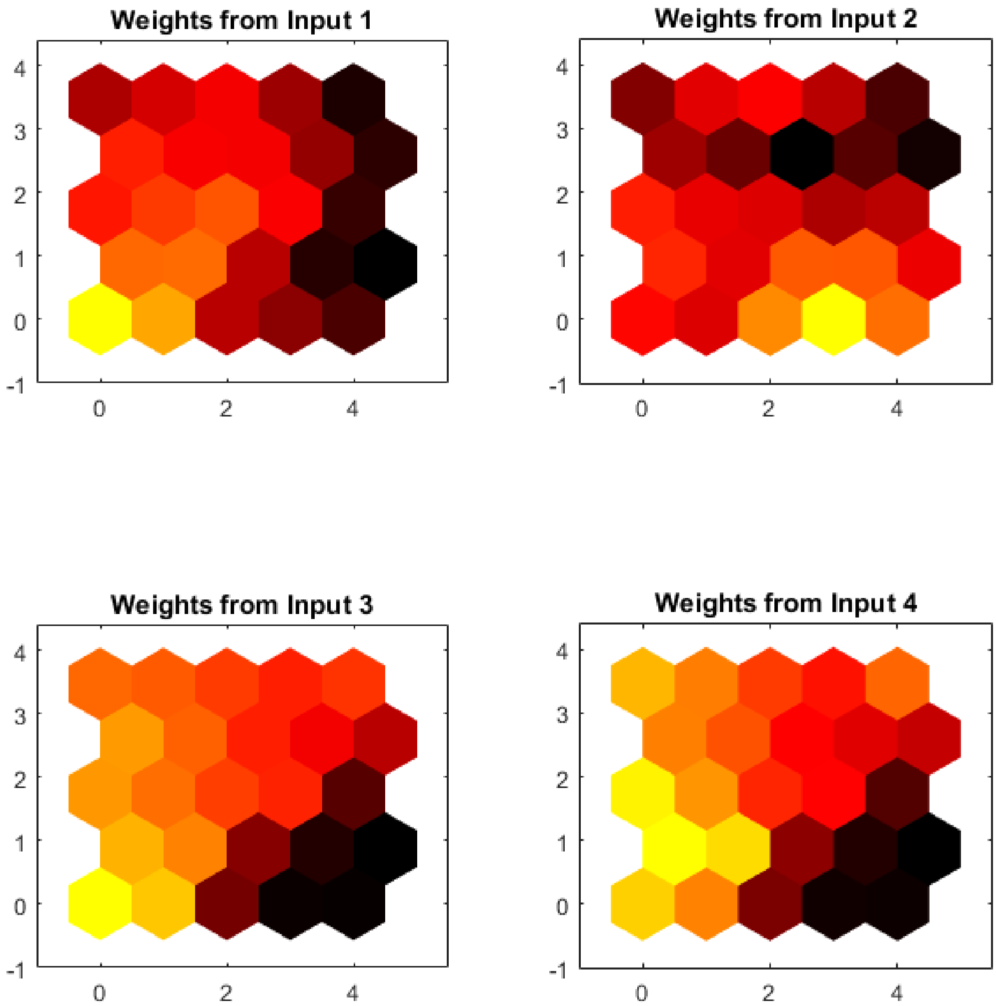
This plot supports `SOM` networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

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`

plotsompos

Plot self-organizing map weight positions

Syntax

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

Description

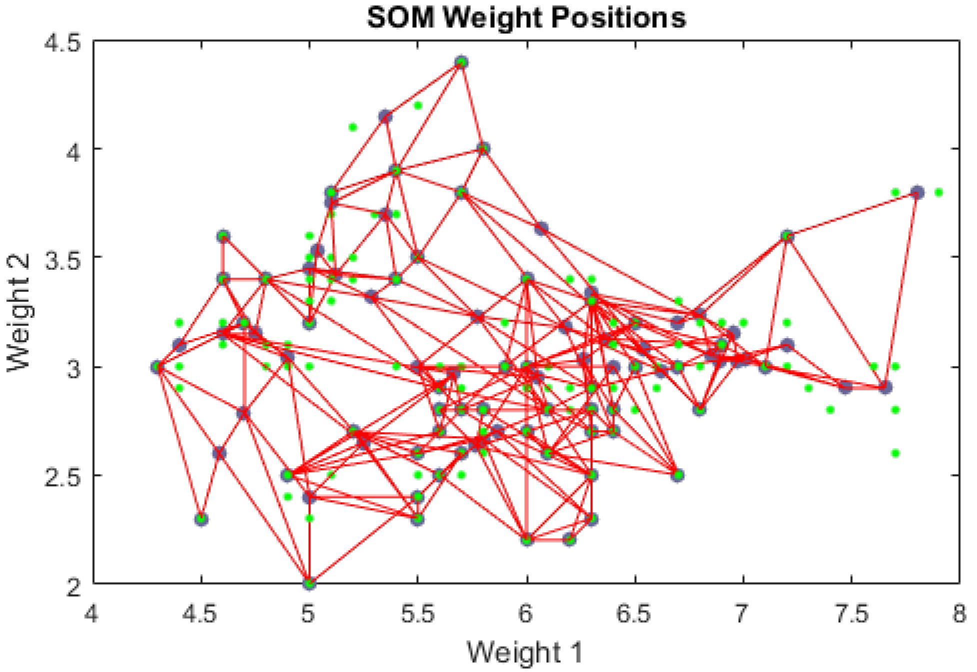
`plotsompos(net)` plots the input vectors as green dots and shows how the `SOM` classifies the input space by showing blue-gray dots for each neuron's weight vector and connecting neighboring neurons with red lines.

`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

`plotsomnd` | `plotsomplanes` | `plotsomhits`

plotsomtop

Plot self-organizing map topology

Syntax

`plotsomtop(net)`

Description

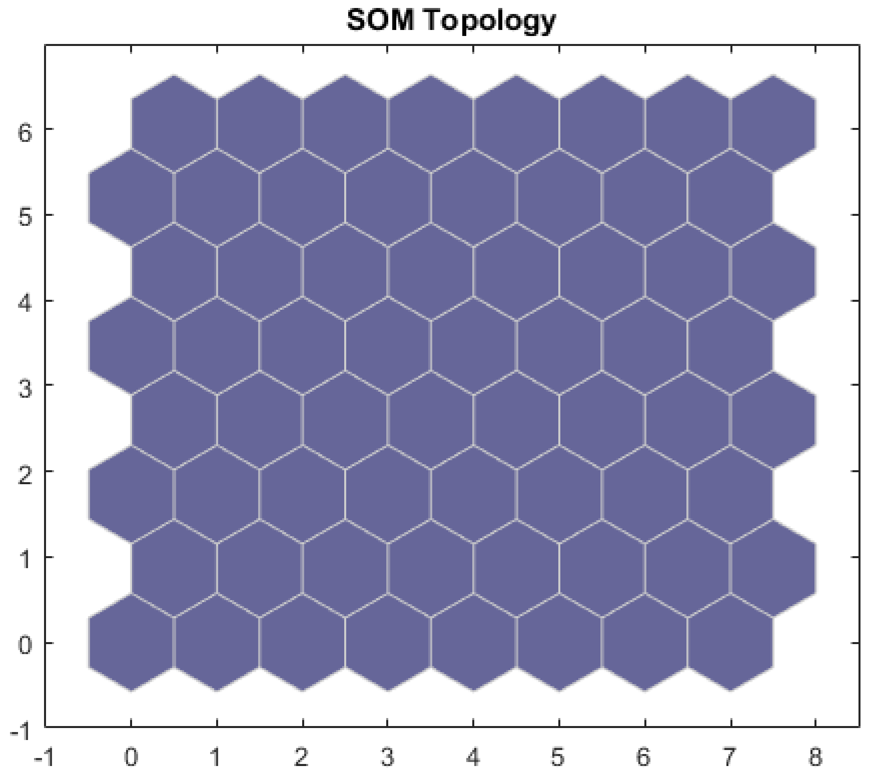
`plotsomtop(net)` plots the topology of a `SOM` layer.

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

plotsomnd | plotsomplanes | plotsomhits

plottrainstate

Plot training state values

Syntax

```
plottrainstate(tr)
```

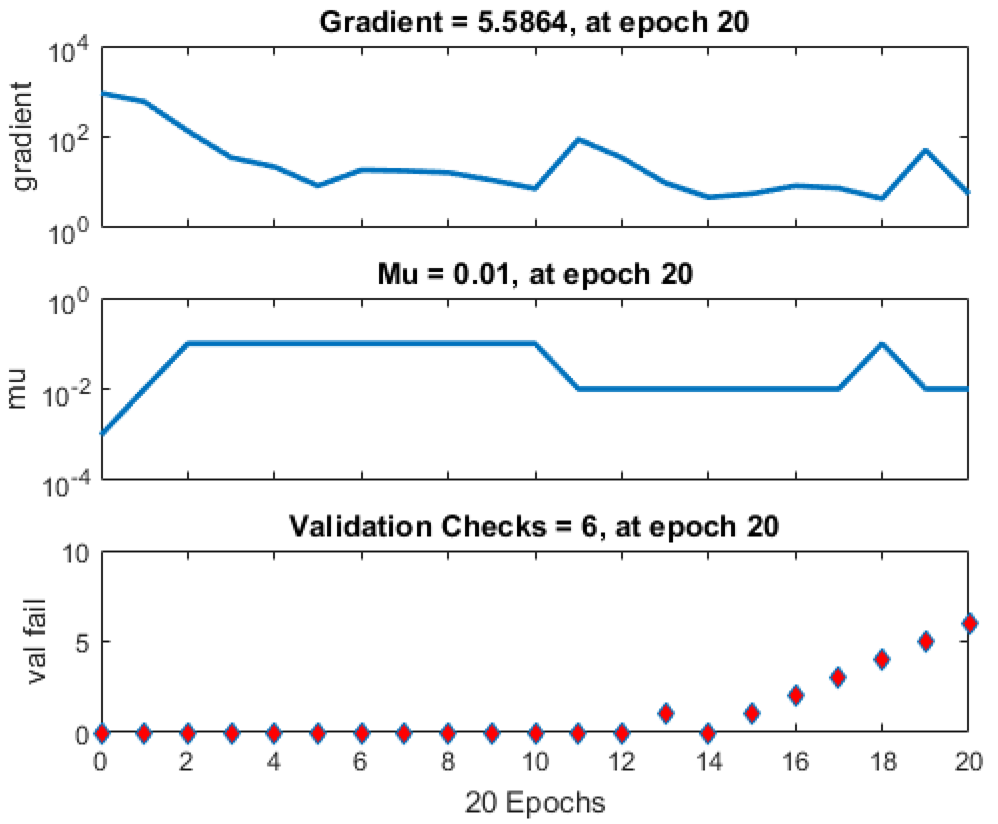
Description

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

Examples

Plot Training State Values

```
[x,t] = house_dataset;
net = feedforwardnet(10);
[net,tr] = train(net,x,t);
plottrainstate(tr)
```



See Also

`plotfit` | `plotperform` | `plotregression`

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 = '-')

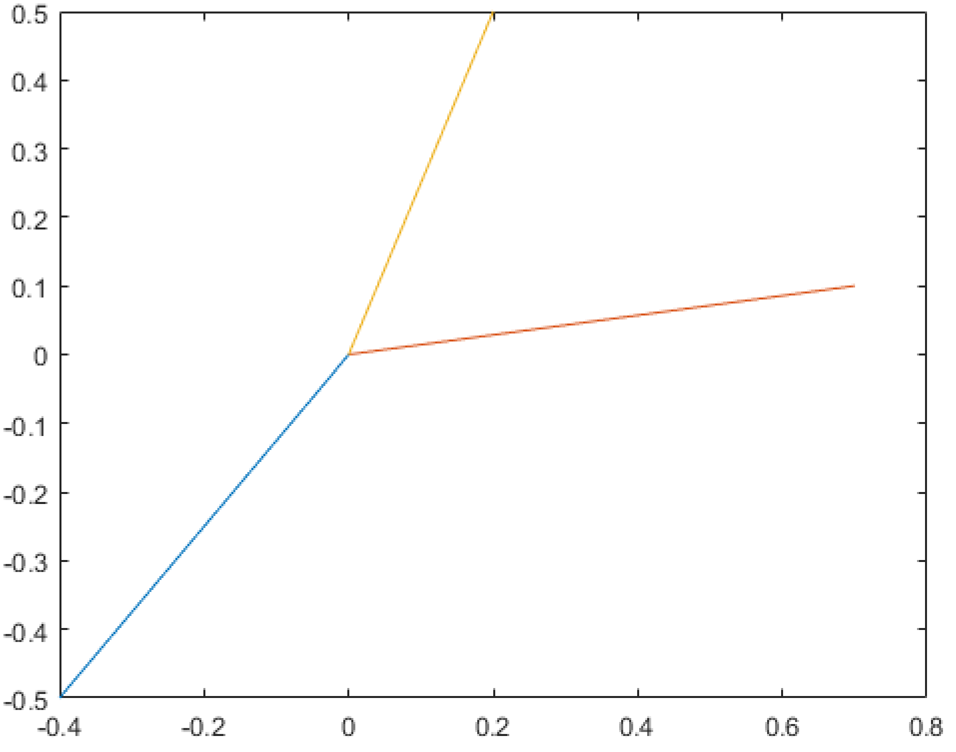
and plots the column vectors of M.

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

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,'-')
```



0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.4
-0.2
0
0.2
0.4
0.6
0.8

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 = `'+'`)

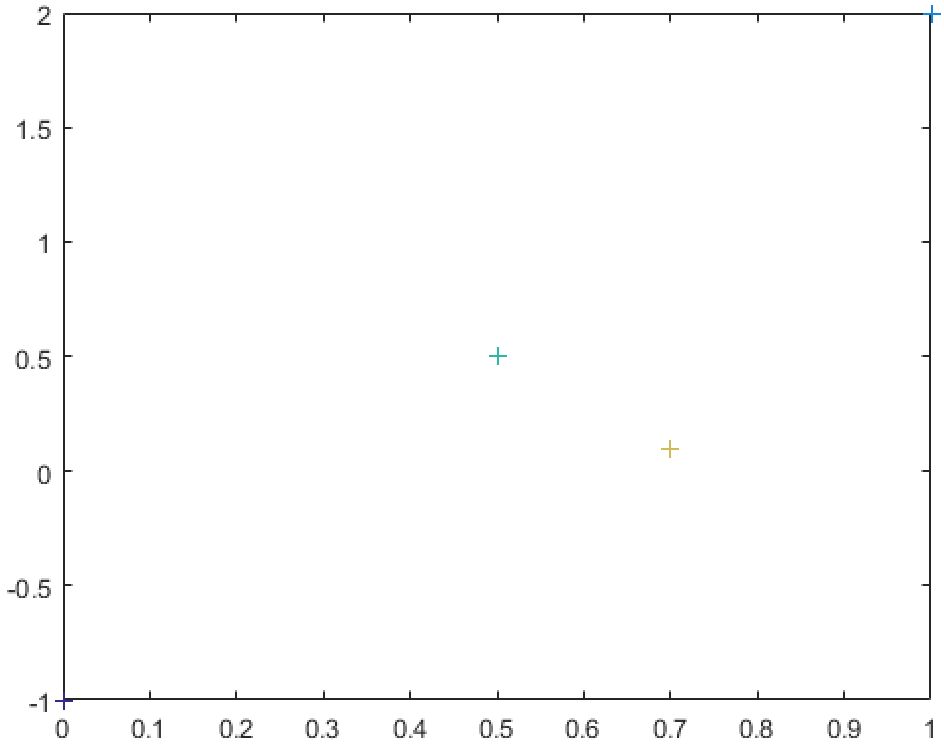
and plots each `i`th vector in `X` with a marker `M`, using the `i`th value in `C` as the color coordinate.

`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

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)
```


2
1.5
1
0.5
0
-0.5
-1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

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.

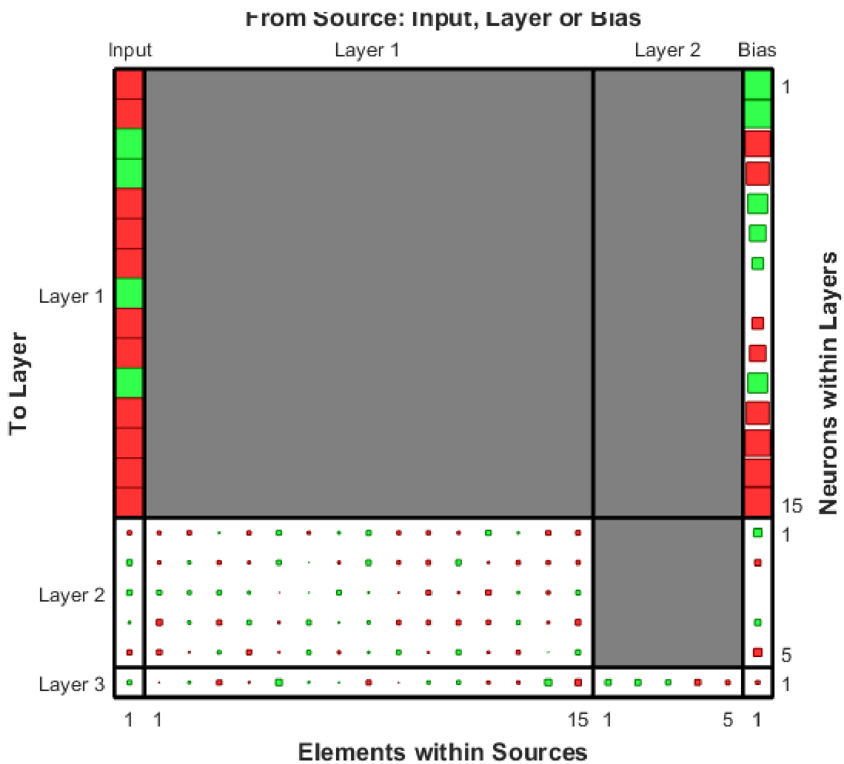
`plotwb(...,'fromLayers',fromLayers)` optionally defines which layers will have weights coming from them plotted.

`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.

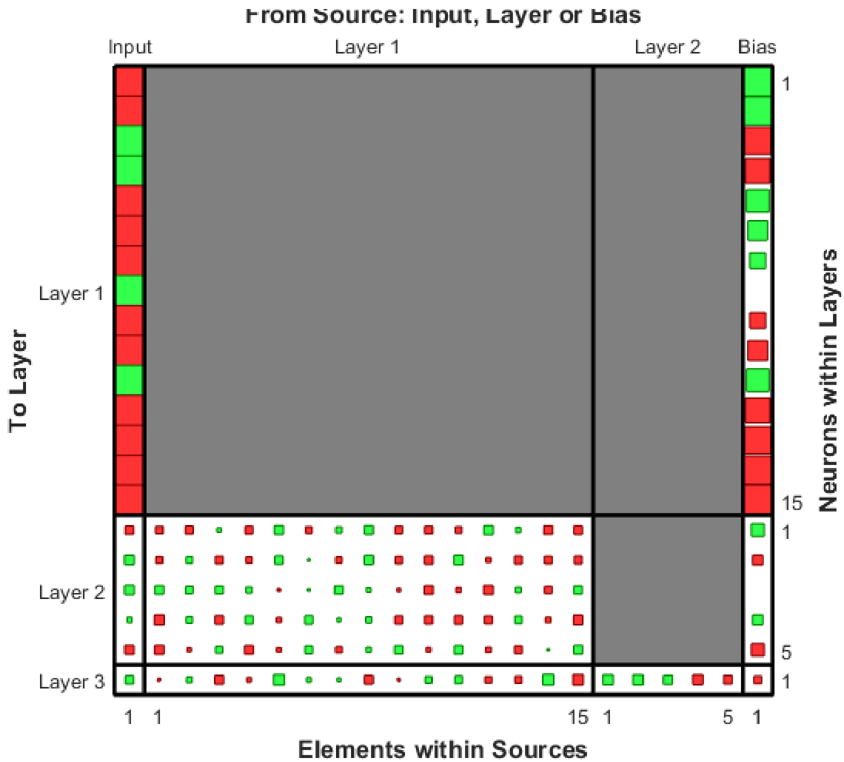
Examples

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

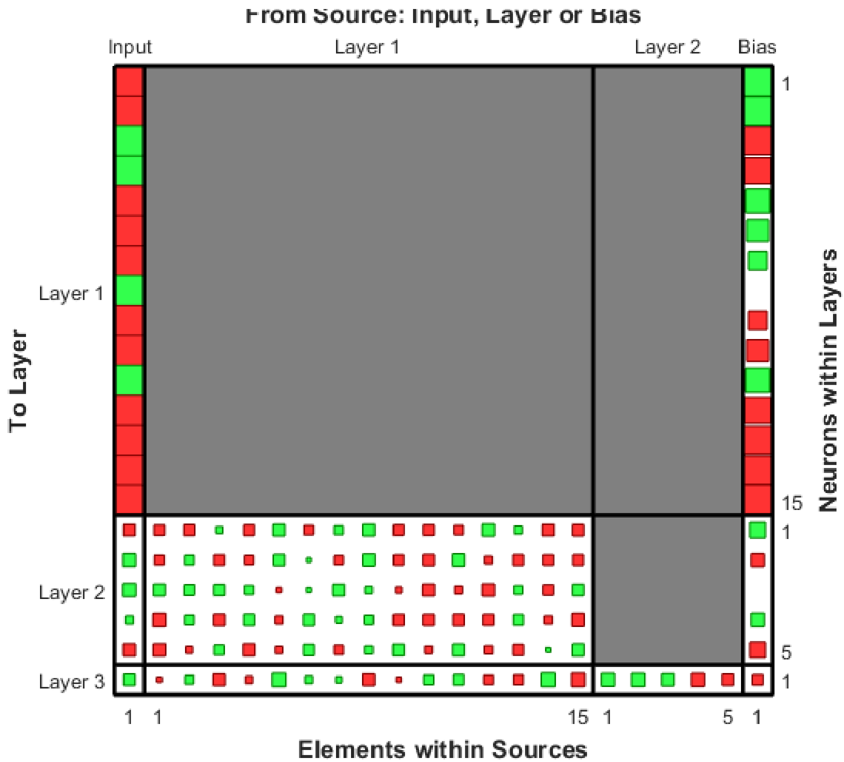
```
[x,t] = simplefit_dataset;
net = cascadeforwardnet([15 5]);
net = configure(net,x,t);
plotwb(net)
```



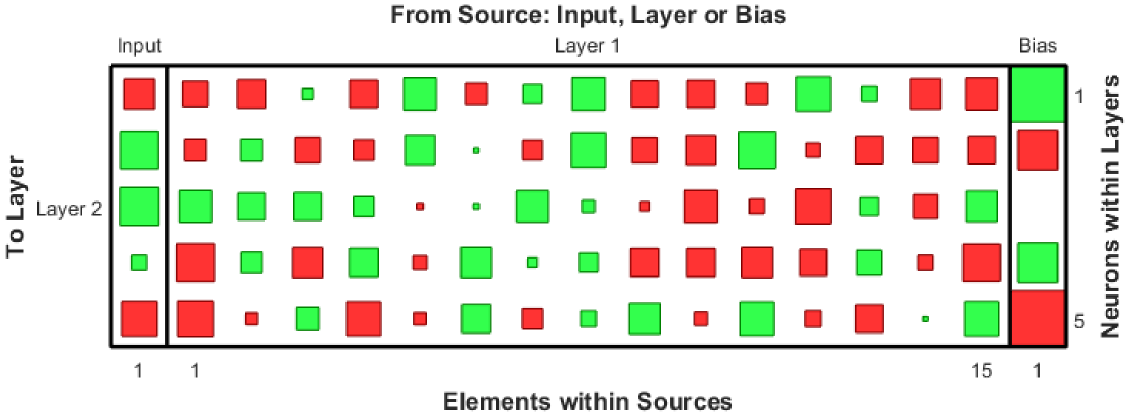

```
plotwb(net,'root',3)
```



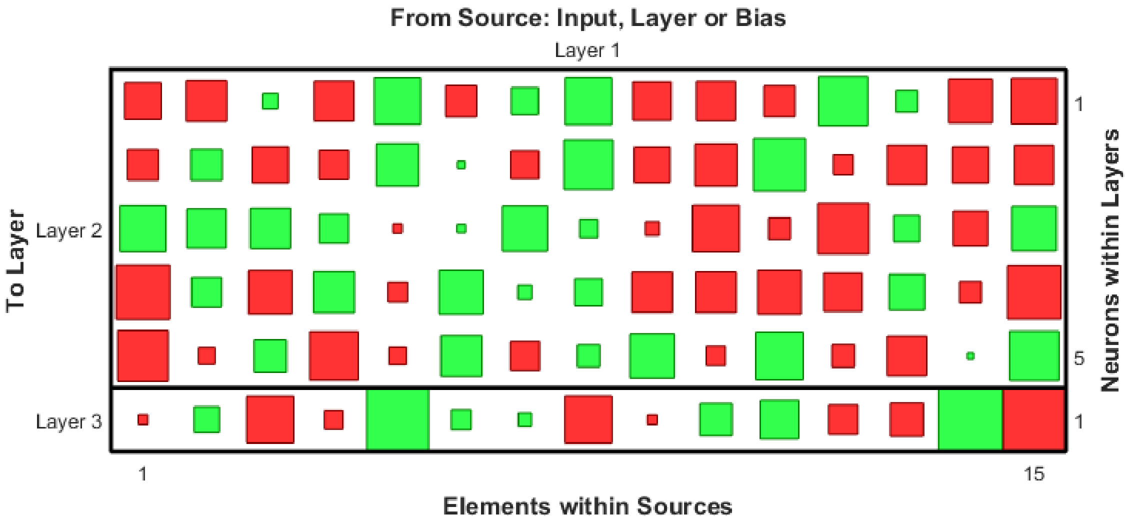

```
plotwb(net,'root',4)
```



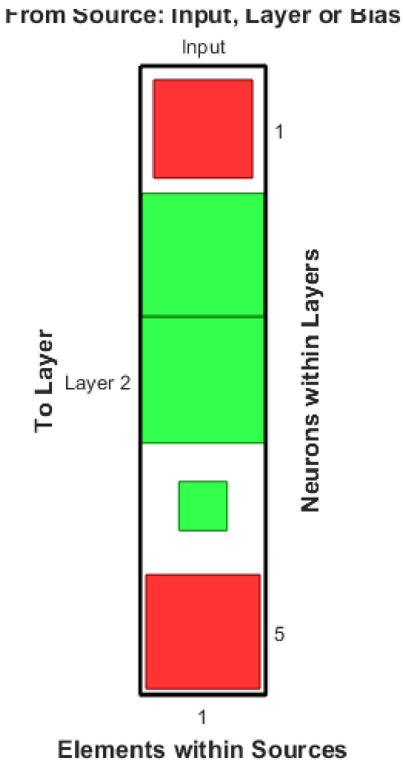

```
plotwb(net,'toLayers',2)
```




```
plotwb(net,'fromLayers',1)
```




```
plotwb(net,'toLayers',2,'fromInputs',1)
```



See Also

`plotsomplanes`

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)
```

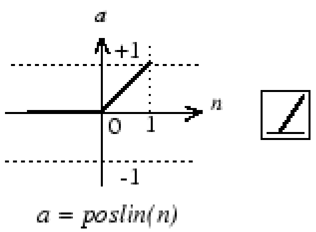
See Also

normc | normr

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`.

More About

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

sim | purelin | satlin | satlins

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,

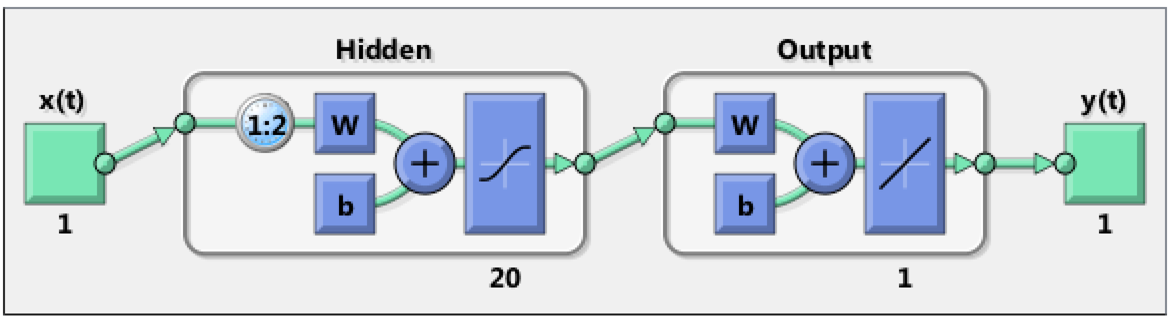
`Xs`	Shifted inputs

`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

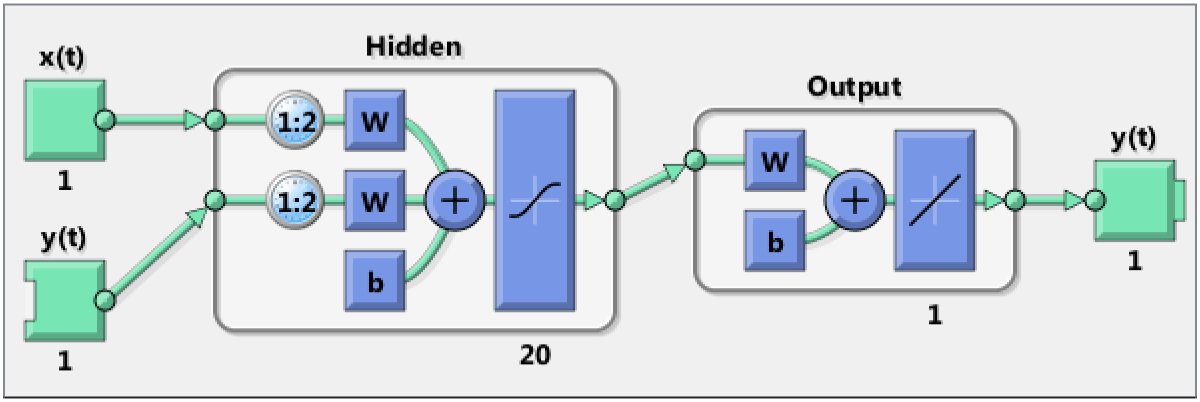
Here a time-delay network with 20 hidden neurons is created, trained and simulated.

```
[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);
```



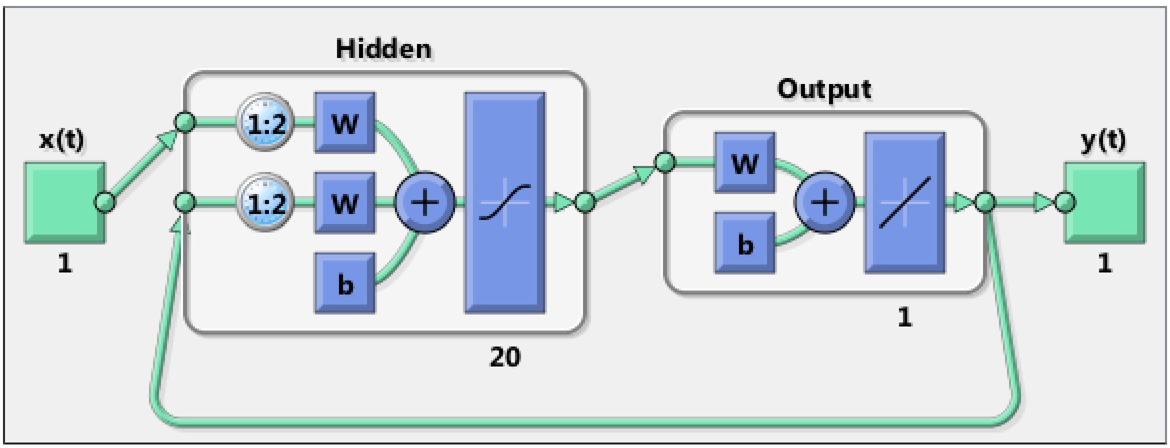
Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```
[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

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) + x_independent;
x1_redundant = x_independent + x_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) + x_independent;
x2_redundant = x_independent + x_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

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';
```

More About

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` | `mapstd` | `mapminmax`

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

prunedata | gensim

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)
```

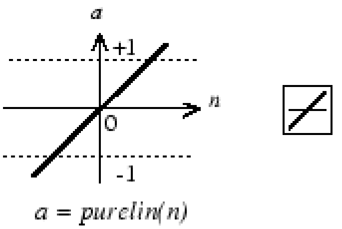
See Also

prune | gensim

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`* string:

`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';
```

More About

Algorithms

```
a = purelin(n) = n
```

See Also

sim | satlin | satlins

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`.

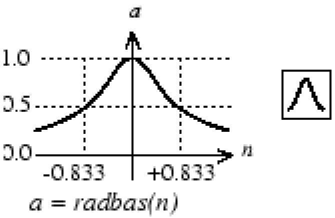
Examples

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

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';
```

More About

Algorithms

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

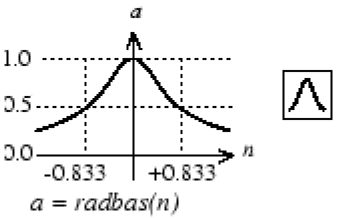
See Also

`sim` | `radbasn` | `tribas`

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';
```

More About

Algorithms

```
a = radbasn(n) = exp(-n^2) / sum(exp(-n^2))
```

See Also

`sim` | `radbas` | `tribas`

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

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

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

`randsmall` | `randnr` | `randnc` | `initwb` | `initlay` | `init`

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

`rands` | `randnr` | `randnc` | `initwb` | `initlay` | `init`

randtop

Random layer topology function

Syntax

```
pos = randtop(dim1,dim2,...,dimN)
```

Description

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

`pos = randtop(dim1,dim2,...,dimN)` takes `N` arguments,

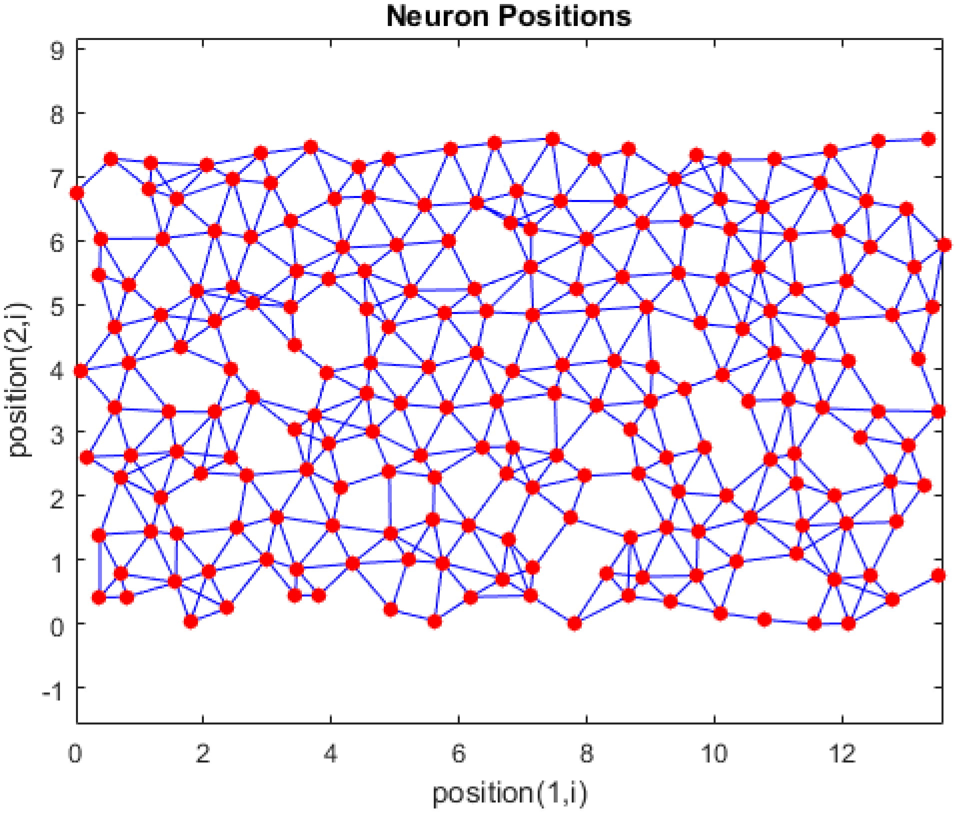
`dimi`	Length of layer in dimension `i`

and returns an `N`-by-`S` matrix of `N` coordinate vectors, where `S` is the product of `dim1*dim2*...*dimN`.

Examples

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

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

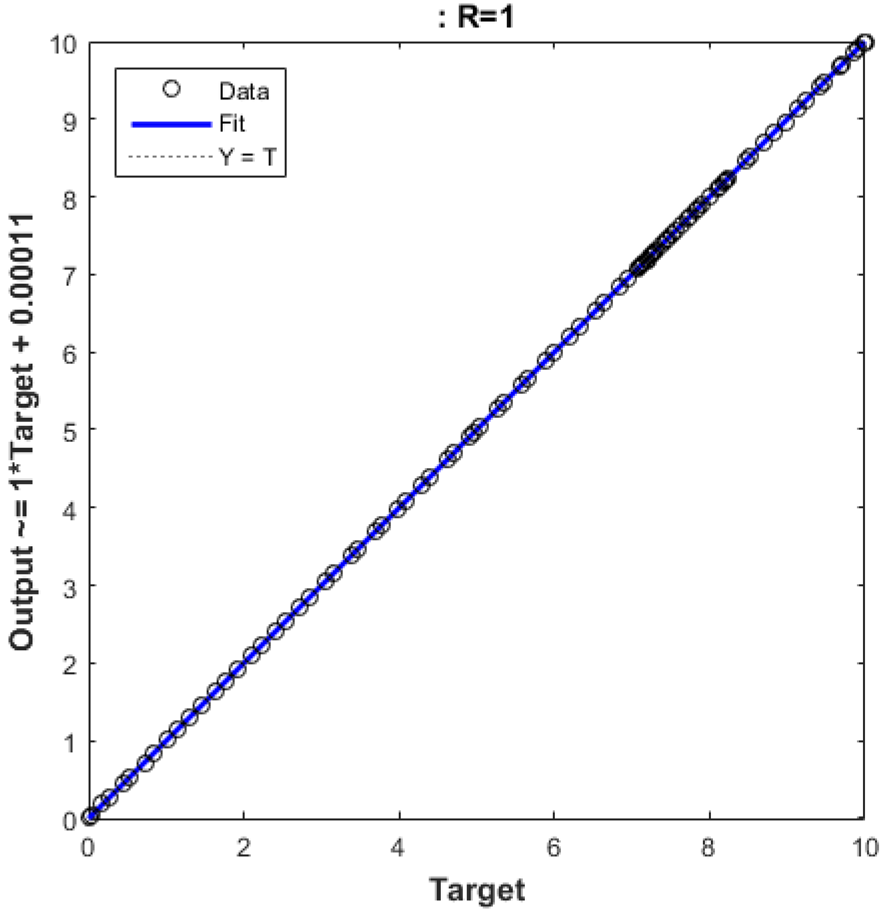
`[r,m,b] = regression(t,y,'one')` combines all matrix rows before regressing, and returns single scalar regression, slope, and offset values.

Examples

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)
plotregression(t,y)


r =

    1.0000


m =

    1.0000


b =

   1.0878e-04
```



See Also

`plotregression` | `confusion`

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` | `mapstd` | `mapminmax` | `processpca`

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

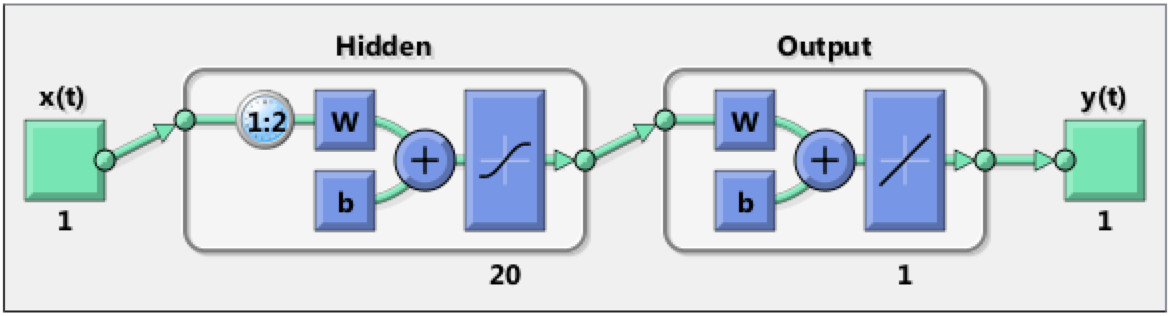
and returns the network with input delay connections decreased, and output feedback delays increased, by the specified number of delays `n`. The result is a network which behaves identically, except that outputs are produced `n` timesteps earlier.

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

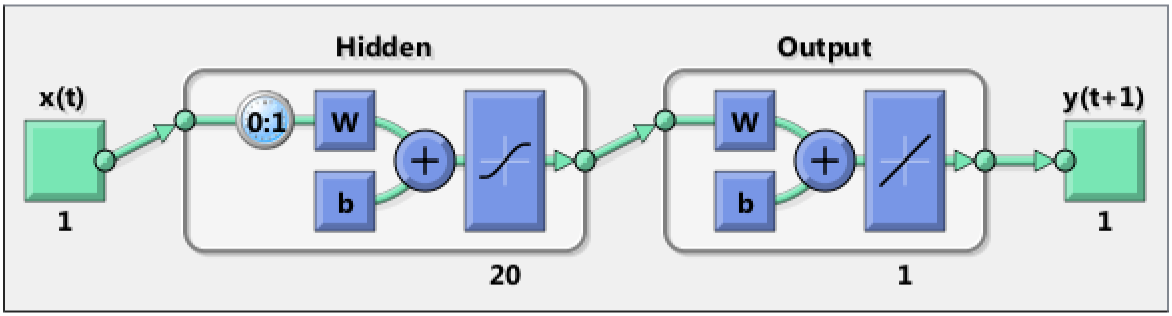
Examples

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.

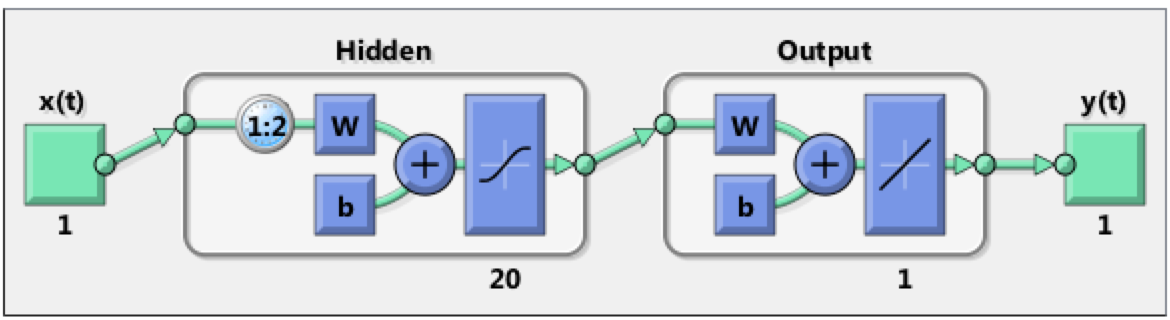
```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)
```




```
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`

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)
```

More About

Algorithms

In the reverse calculation, the unknown values of replaced rows are represented with `NaN` values.

See Also

`fixunknowns` | `mapstd` | `mapminmax` | `processpca`

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

`init` | `sim` | `adapt` | `train`

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 (the number of outputs greater or equal to the threshold, divided by the number of one targets), and the False Positive Ratio (the number of outputs less than the threshold, divided by the number of zero targets).

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

`plotroc` | `confusion`

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

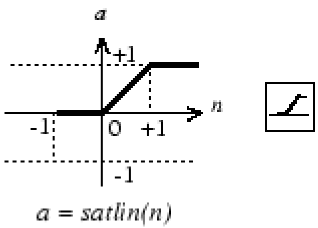
To prepare a custom network to be trained with `sae`, set `net.performFcn` to `'sae'`. This automatically sets `net.performParam` to the default function parameters.

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

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`* string:

`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';
```

More About

Algorithms

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

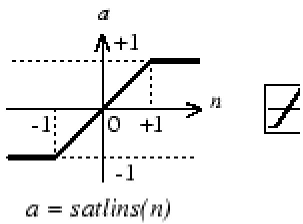
See Also

sim | poslin | satlins | purelin

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`* string:

`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)
```

More About

Algorithms

```
satlins(n) = -1, if n <= -1
n, if -1 <= n <= 1
1, if 1 <= n
```

See Also

sim | satlin | poslin | purelin

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 `help newp` and `help newlin` for simulation examples.

See Also

`dotprod` | `sim` | `dist` | `negdist` | `normprod`

selforgmap

Self-organizing map

Syntax

```
selforgmap(dimensions,coverSteps,initNeighbor,topologyFcn,distanceFcn)
```

Description

Self-organizing maps learn to cluster data based on similarity, topology, with a preference (but no guarantee) of assigning the same number of instances to each class.

Self-organizing maps are used both to cluster data and to reduce the dimensionality of data. They are inspired by the sensory and motor mappings in the mammal brain, which also appear to automatically organizing information topologically.

`selforgmap(dimensions,coverSteps,initNeighbor,topologyFcn,distanceFcn)` takes these arguments,

`dimensions`	Row vector of dimension sizes (default = `[8 8]`)
`coverSteps`	Number of training steps for initial covering of the input space (default = 100)
`initNeighbor`	Initial neighborhood size (default = 3)
`topologyFcn`	Layer topology function (default = `'hextop'`)
`distanceFcn`	Neuron distance function (default = `'linkdist'`)

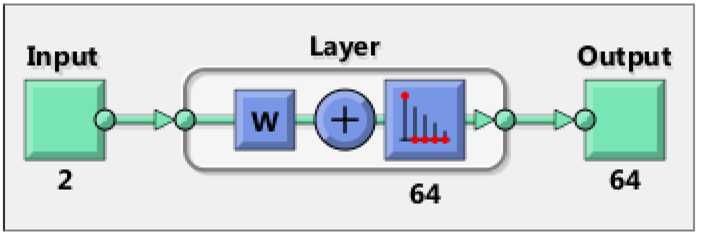
and returns a self-organizing map.

Examples

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

lvqnet | competlayer | nctool

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

`getwb` | `formwb` | `setwb`

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`

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

`nndata` | `numelements` | `getelements` | `catelements` | `setsamples` | `setsignals` | `settimesteps`

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

`nndata` | `numsamples` | `getsamples` | `catsamples` | `setelements` | `setsignals` | `settimesteps`

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

`nndata` | `numsignals` | `getsignals` | `catsignals` | `setelements` | `setsamples` | `settimesteps`

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`

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

`nndata` | `numtimesteps` | `gettimesteps` | `cattimesteps` | `setelements` | `setsamples` | `setsignals`

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:

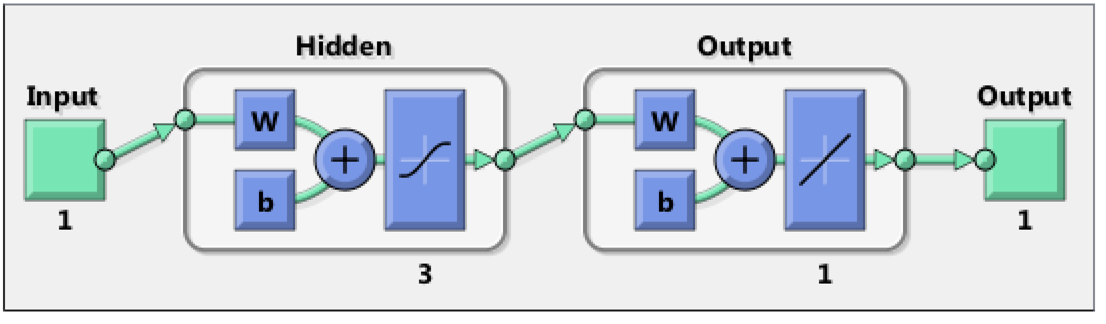
`net`	Neural network
`wb`	Vector of weight and bias values

Examples

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

getwb | formwb | separatewb

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
D	=	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, such as Hopfield networks, 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 neighborhood characteristics, `x`, to median house prices, `t`. A feedforward network with 10 neurons is created and trained on that data, then simulated.

```
[x,t] =  house_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');
```

More About

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

`init` | `adapt` | `train` | `revert`

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)
```

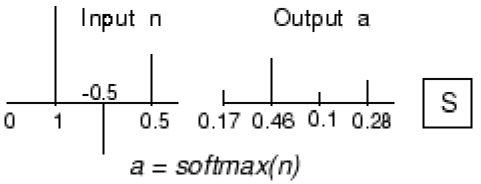
See Also

nndata | nndata2sim

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';
```

More About

Algorithms

```
a = softmax(n) = exp(n)/sum(exp(n))
```

See Also

sim | compet

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)

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 is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchbac search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
```

```
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchbac';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchbac` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchbac`,

1 Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.
2 Set `net.trainParam.searchFcn` to `'srchbac'`.

The `srchbac` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Definitions

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.

More About

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`

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)

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 is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchbac search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchbre';
net.trainParam.epochs = 50;
```

```
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchbre` with `newff`, `newcf`, or `newelm`. To prepare a custom network to be trained with `traincgf`, using the line search function `srchbre`,

1. Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam.searchFcn` to `'srchbre'`.

The `srchbre` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Definitions

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.

More About

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`

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)

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 is created. The network's input ranges from `[0 to 10]`. The first layer has two `tansig` neurons, and the second layer has one `logsig` neuron. The `traincgf` network training function and the `srchcha` search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchcha';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
```

```
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchcha` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchcha`,

1 Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.
2 Set `net.trainParam.searchFcn` to `'srchcha'`.

The `srchcha` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Definitions

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.

More About

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`

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)

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 is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchgol search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchgol';
net.trainParam.epochs = 50;
```

```
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchgol` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchgol`,

1 Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.

2 Set `net.trainParam.searchFcn` to `'srchgol'`.

The `srchgol` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Definitions

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.

More About

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`

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)

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 is created. The network's input ranges from [0 to 10]. The first layer has two tansig neurons, and the second layer has one logsig neuron. The traincgf network training function and the srchhyb search function are to be used.

Create and Test a Network

```
net = newff([0 5],[2 1],{'tansig','logsig'},'traincgf');
a = sim(net,p)
```

Train and Retest the Network

```
net.trainParam.searchFcn = 'srchhyb';
net.trainParam.epochs = 50;
net.trainParam.show = 10;
net.trainParam.goal = 0.1;
net = train(net,p,t);
```

```
a = sim(net,p)
```

Network Use

You can create a standard network that uses `srchhyb` with `newff`, `newcf`, or `newelm`.

To prepare a custom network to be trained with `traincgf`, using the line search function `srchhyb`,

1 Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.
2 Set `net.trainParam.searchFcn` to `'srchhyb'`.

The `srchhyb` function can be used with any of the following training functions: `traincgf`, `traincgb`, `traincgp`, `trainbfg`, `trainoss`.

Definitions

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.

More About

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`

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.

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

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

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]})
```

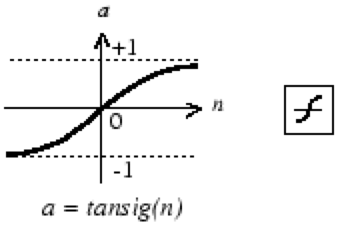
See Also

`meanabs` | `meansqr` | `sumabs`

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';
```

More About

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

`sim` | `logsig`

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

`nndata` | `extendts` | `preparets`

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'`)

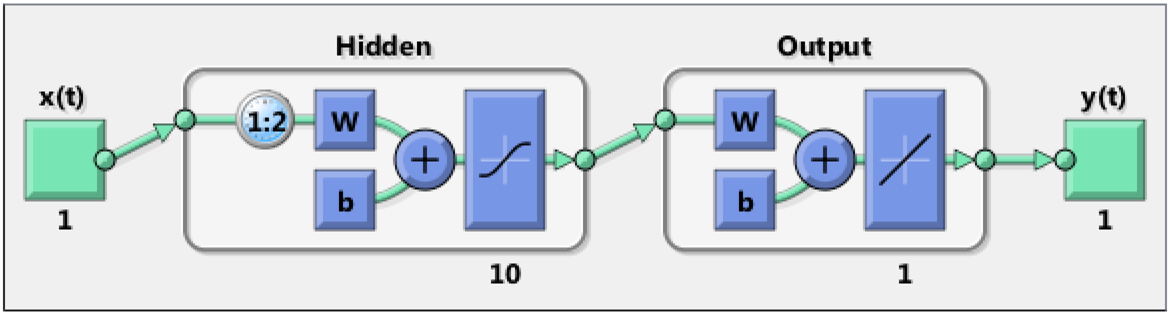
and returns a time delay neural network.

Examples

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

```
[X,T] = simpleseries_dataset;
net = timedelaynet(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)
```

```
perf =

    0.0225
```



See Also

preparets | removedelay | distdelaynet | narnet | narxnet

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

nndata | fromnndata | nndata2sim | sim2nndata

train

Train neural network

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)
```

To Get Help

Type `help network/train`.

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.outptus{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.

More About

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

`init` | `revert` | `sim` | `adapt`

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.showCommandLin`	`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`.

More About

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`

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

Here a neural network is trained to predict median house prices.

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainbfg');
net = train(net,x,t);
y = net(x)
```

Definitions

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 x 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.

More About

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` | `traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `traincgp` | `trainoss` | `feedforwardnet`

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` string:

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

More About

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

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.

More About

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.

The parameter mem_reduc indicates how to use memory and speed to calculate the Jacobian jX. If mem_reduc is 1, then trainlm runs the fastest, but can require a lot of memory. Increasing mem_reduc to 2 cuts some of the memory required by a factor of two, but slows trainlm somewhat. Higher values continue to decrease the amount of memory needed and increase the training times.

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

MacKay, *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415–447

Foresee and Hagan, *Proceedings of the International Joint Conference on Neural Networks*, June, 1997

See Also

`cascadeforwardnet` | `traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `traincgp` | `trainbfg` | `feedforwardnet`

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.showGUI`	`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.

More About

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`

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.

More About

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`

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`	`'srchch`	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

Here a neural network is trained to predict median house prices.

```
[x,t] = house_dataset;
net = feedforwardnet(10,'traincgb');
net = train(net,x,t);
y = net(x)
```

Definitions

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).

More About

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

`traingdm` | `traingda` | `traingdx` | `trainlm` | `traincgp` | `traincgf` | `trainscg` | `trainoss` | `trainbfg`

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

Here a neural network is trained to predict median house prices.

```
[x,t] = house_dataset;
net = feedforwardnet(10,'traincgf');
net = train(net,x,t);
y = net(x)
```

Definitions

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.

More About

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

`traingdm` | `traingda` | `traingdx` | `trainlm` | `traincgb` | `trainscg` | `traincgp` | `trainoss` | `trainbfg`

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

Examples

Here a neural network is trained to predict median house prices.

```
[x,t] = house_dataset;
net = feedforwardnet(10,'traincgp');
net = train(net,x,t);
y = net(x)
```

Definitions

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).

More About

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

traingdm | traingda | traingdx | trainlm | trainrp | traincgf | traincgb | trainscg | trainoss | trainbfg

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

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.

More About

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

traingdm | traingda | traingdx | trainlm

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

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)
```

More About

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

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

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.

More About

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`

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.

Definitions

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.

More About

Algorithms

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

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.

trainlm is the default training function for several network creation functions including newcf, newdtdnn, newff, and newnarx.

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

Here a neural network is trained to predict median house prices.

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainlm');
net = train(net,x,t);
y = net(x)
```

Definitions

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.

Limitations

This function uses the Jacobian for calculations, which assumes that performance is a mean or sum of squared errors. Therefore, networks trained with this function must use either the `mse` or `sse` performance function.

More About

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`.

The parameter `mem_reduc` indicates how to use memory and speed to calculate the Jacobian `jX`. If `mem_reduc` is 1, then `trainlm` runs the fastest, but can require a lot of memory. Increasing `mem_reduc` to `2` cuts some of the memory required by a factor of two, but slows `trainlm` somewhat. Higher states continue to decrease the amount of memory needed and increase training times.

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).

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

Here a neural network is trained to predict median house prices.

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainoss');
net = train(net,x,t);
```

```
y = net(x)
```

Definitions

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.

More About

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

`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainscg` | `traincgp` | `trainbfg`

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.

More About

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`

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

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.

More About

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

`traingdm` | `traingda` | `traingdx` | `trainlm` | `traincgp` | `traincgf` | `traincgb` | `trainscg` | `trainoss` | `trainbfg`

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`.

More About

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`

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.

More About

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`

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.

More About

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

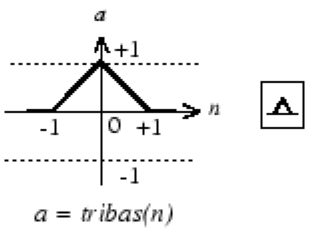
See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincgf` | `traincgb` | `trainbfg` | `traincgp` | `trainoss`

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';
```

More About

Algorithms

```
a = tribas(n) = 1 - abs(n), if -1 <= n <= 1
              = 0, otherwise
```

See Also

sim | radbas

tritop

Triangle layer topology function

Syntax

```
pos = tritop(dim1,dim2,...,dimN)
```

Description

`tritop` calculates neuron positions for layers whose neurons are arranged in an `N`-dimensional triangular grid.

`pos = tritop(dim1,dim2,...,dimN)` takes `N` arguments,

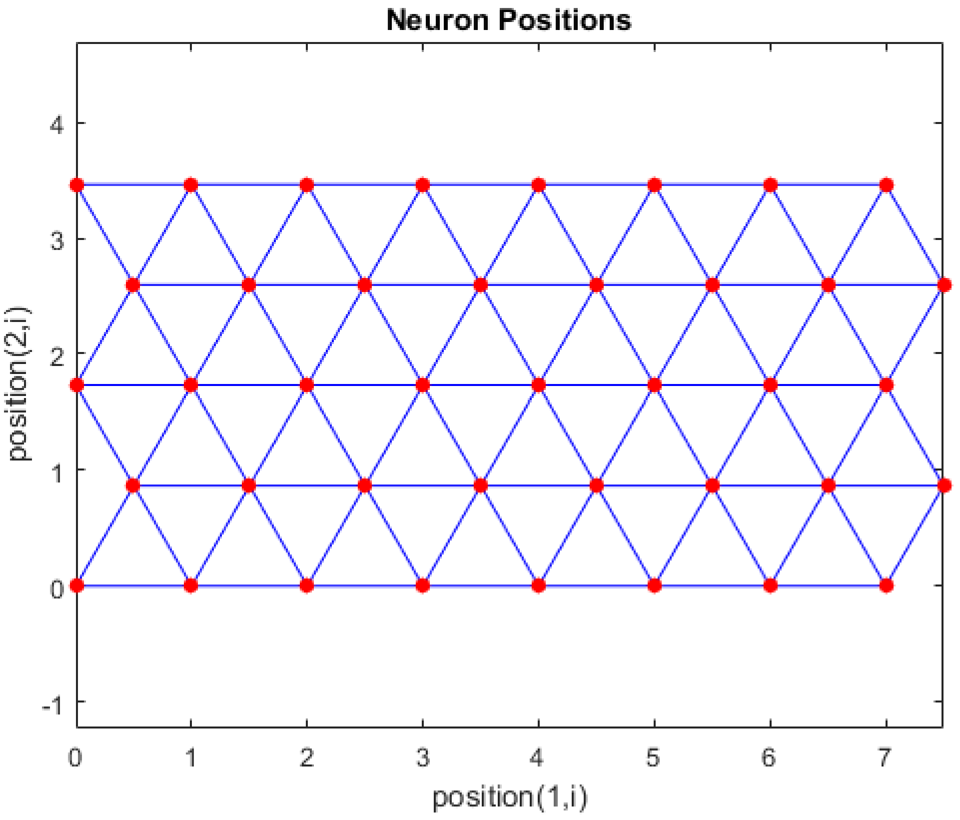
`dimi`	Length of layer in dimension `i`

and returns an `N`-by-`S` matrix of `N` coordinate vectors, where `S` is the product of `dim1*dim2*...*dimN`.

Examples

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

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`

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

ind2vec | sub2ind | ind2sub

view

View neural network

Syntax

```
view(net)
```

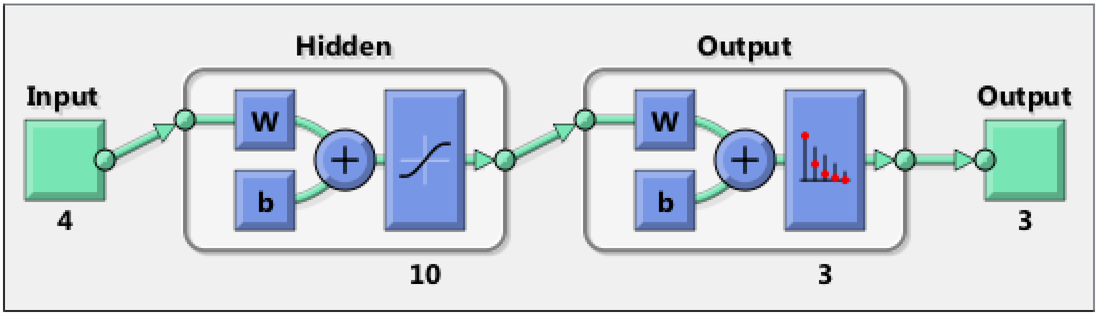
Description

`view(net)` opens a window that shows your neural network (specified in `net`) as a graphical diagram.

Example

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)
```



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)` 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 in the MATLAB documentation.

See Also

`trainAutoencoder`

More About

- Class Attributes
- Property Attributes

Introduced in R2015b

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 =

    0.0045
```

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.

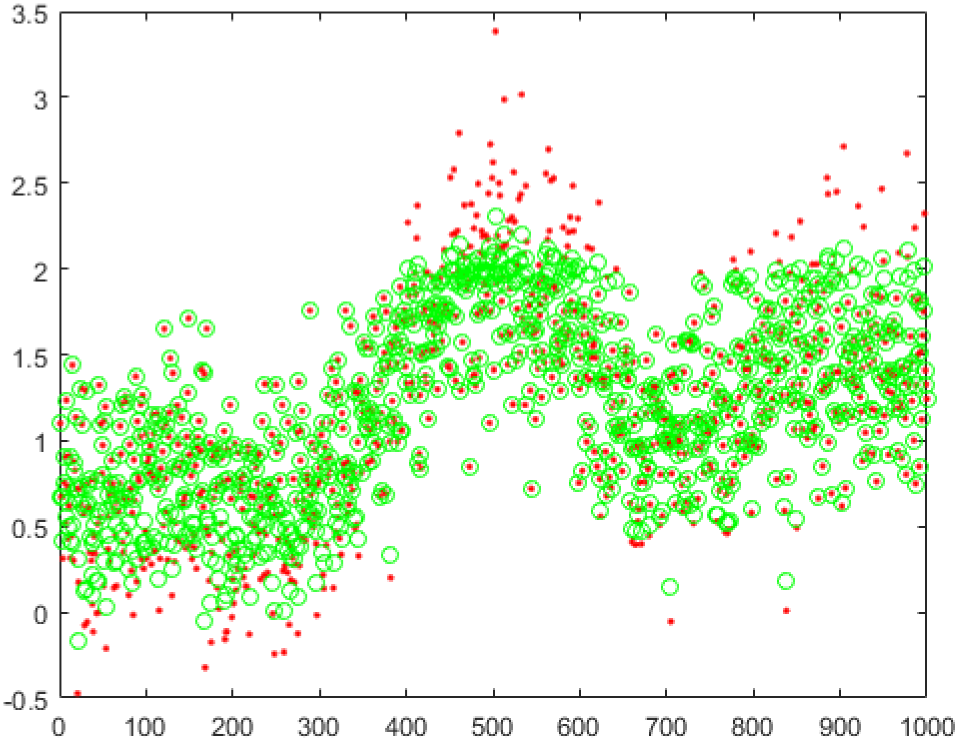
```
n = 1000;
r = sort(-10 + 20*rand(n,1));
xtest = 1 + r*5e-2 + sin(r)./r + 0.4*randn(n,1);
```

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.

```
X = digittrain_dataset;
```

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(X,hiddenSize,...
        'L2WeightRegularization',0.004,...
```

```
        'SparsityRegularization',4,...
        'SparsityProportion',0.15);
```

Load the test data.

```
x = digittest_dataset;
```

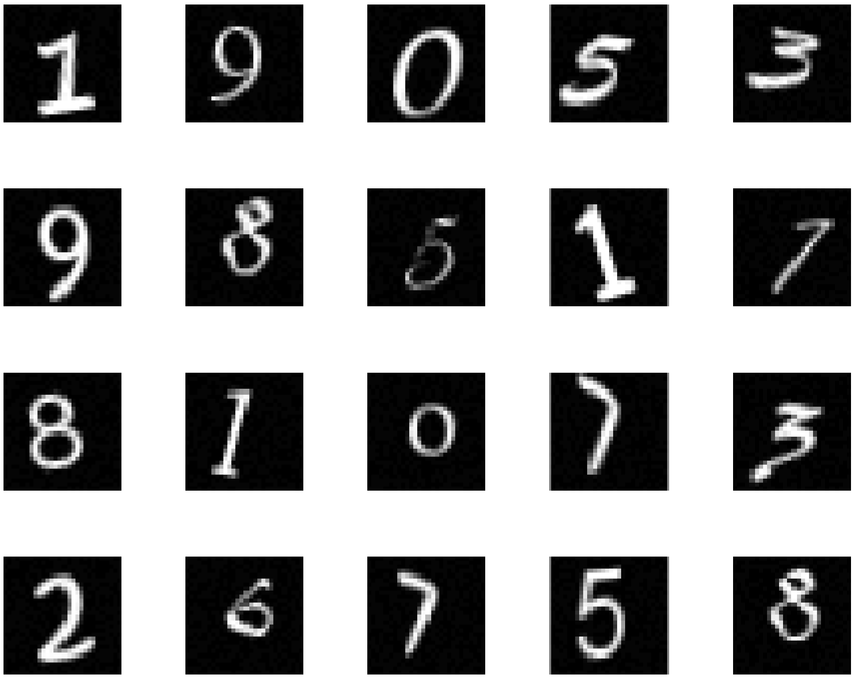
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, `autoenc`.

```
xReconstructed = predict(autoenc,x);
```

View the actual test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(X{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 $$f(z)=\begin{cases}0, & \text{if } z\leq 0\\ z, & \text{if } 0<z<1\\ 1, & \text{if } z\geq 1\end{cases}$$

Example: `'EncoderTransferFunction','satlin'`

`'DecoderTransferFunction'` — Transfer function for the decoder

`'logsig'` (default) | `'satlin'` | `'purelin'`

Transfer function for the decoder, specified as the comma-separated pair consisting of `'DecoderTransferFunction'` and one of the following.

Transfer Function Option	Definition
`'logsig'`	Logistic sigmoid function $$f(z)=\frac{1}{1+e^{-z}}$$
`'satlin'`	Positive saturating linear transfer function $$f(z)=\begin{cases}0, & \text{if } z\leq 0\\ z, & \text{if } 0<z<1\\ 1, & \text{if } z\geq 1\end{cases}$$
`'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 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 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 and β is the coefficient for the sparsity regularization term. 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.

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 in the cost function, specified as the comma-separated pair consisting of `'SparsityRegularization'` and a positive scalar value.

Example: `'SparsityRegularization',1.6`

`'TrainingAlgorithm'` — The algorithm to use for training the autoencoder

`'trainscg'` (default)

The algorithm to use for training the autoencoder, specified as the comma-separated pair consisting of `'TrainingAlgorithm'` and `'trainscg'`. It stands for scaled conjugate gradient descent [1].

`'ScaleData'` — Indicator to rescale the input data

`true` (default) | `false`

Indicator to rescale the input data, specified as the comma-separated pair consisting of `'ScaleData'` and either `true` or `false`.

Autoencoders attempt to replicate their input at their output. For it to be possible, the range of the input data must match the range of the transfer function for the decoder. `trainAutoencoder` automatically scales the training data to this range when training an autoencoder. If the data was scaled while training an autoencoder, the `predict`, `encode`, and `decode` methods also scale the data.

Example: `'ScaleData',false`

`'UseGPU'` — Indicator to use GPU for training

`false` (default) | `true`

Indicator to use GPU for training, specified as the comma-separated pair consisting of `'UseGPU'` and either `true` or `false`.

Example: `'UseGPU',true`

Output Arguments

`autoenc` — Trained autoencoder
`Autoencoder` object

Trained autoencoder, returned as an `Autoencoder` object. For information on the properties and methods of this object, see `Autoencoder` class page.

More About

Autoencoders

An autoencoder is a neural network which is trained to replicate its input at its output. Autoencoders can be used as tools to learn deep neural networks. Training an autoencoder is unsupervised in the sense that no labeled data is needed. The training process is still based on the optimization of a cost function. The cost function measures the error between the input x and its reconstruction at the output $\hat{x}$.

An autoencoder is composed of an encoder and a decoder. The encoder and decoder can have multiple layers, but for simplicity consider that each of them has only one layer.

If the input to an autoencoder is a vector $\boldsymbol{x} \in \mathbb{R}^{D_x}$, then the encoder maps the vector x to another vector $\boldsymbol{z} \in \mathbb{R}^{D^{(1)}}$ as follows:

$$\boldsymbol{z}^{(1)} = h^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

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:

$$\hat{\boldsymbol{x}} = h^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{x} + \boldsymbol{b}^{(2)}\right),$$

where the superscript (2) represents the second layer. $h^{(2)}:\mathbb{R}^{D_x}\rightarrow\mathbb{R}^{D_x}$ is the transfer function for the decoder, $W^{(1)}\in\mathbb{R}^{D_x\times D^{(1)}}$ is a weight matrix, and $b^{(2)}\in\mathbb{R}^{D_x}$ is a bias vector.

Sparse Autoencoders

Encouraging sparsity of an autoencoder is possible by adding a regularizer to the cost function [2]. This regularizer is a function of the average output activation value of a neuron. The average output activation measure of a neuron i is defined as:

$$\hat{\rho}_i=\frac{1}{n}\sum_{j=1}^{n}z_i^{(1)}\left(x_j\right)=\frac{1}{n}\sum_{j=1}^{n}h\left(w_i^{(1)T}x_j+b_i^{(1)}\right),$$

where n is the total number of training examples. x_j is the jth training example, $w_i^{(1)T}$ is the ith row of the weight matrix $\boldsymbol{W}^{(1)}$, and $b_i^{(1)}$ is the ith 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 is an adjusted mean squared error function 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 and β is the coefficient for the sparsity regularization term. 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`

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.

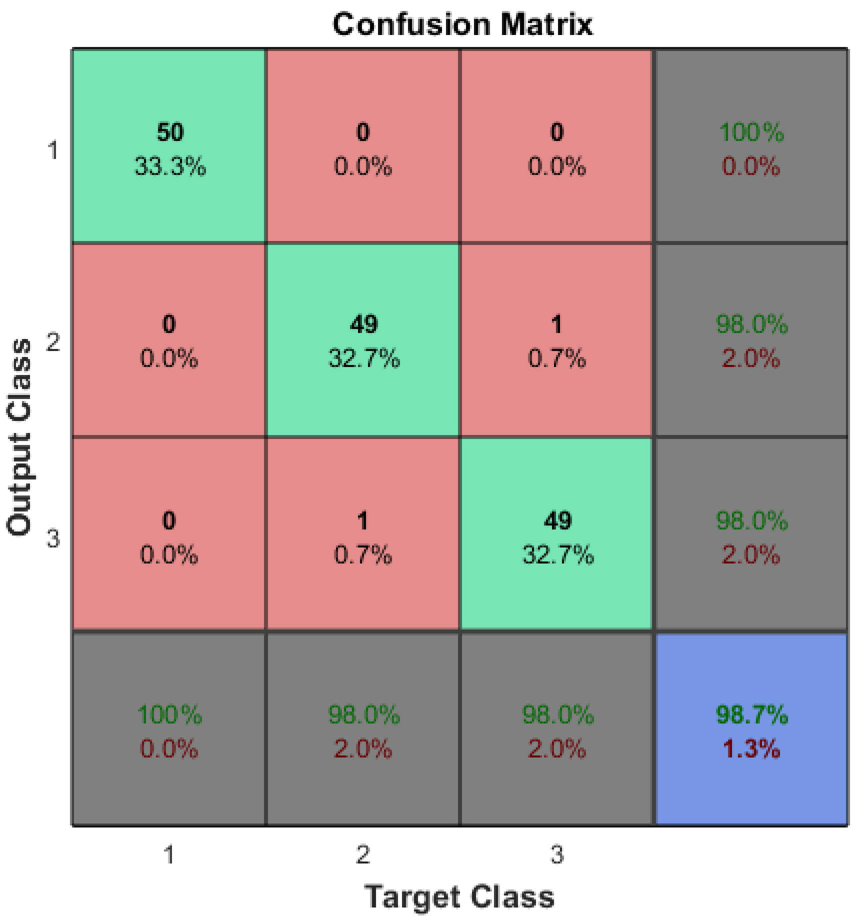
```
net = trainSoftmaxLayer(X,T);
```

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);
```



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 `'trainscg'`, which stands for scale 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

decode

Class: Autoencoder

Decode encoded data

Syntax

```
Y = decode(autoenc,Z)
```

Description

`Y = decode(autoenc,Z)`returns the decoded data `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 = digitsmall_dataset;
```

`X` is a 1-by-500 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 = digittest_dataset;
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:

$$\boldsymbol{z}^{(1)} = h^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

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:

$$\hat{\boldsymbol{x}} = h^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{x} + \boldsymbol{b}^{(2)}\right),$$

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, `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 = digitsmall_dataset;
```

`X` is a 1-by-500 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 = digittest_dataset;
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:

$$\boldsymbol{z}^{(1)} = h^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

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,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,Name,Value)` generates a complete stand-alone function with additional options specified by one or more `Name,Value` pair arguments.

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`.

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`.

`'MatrixOnly'` — Indicator for the generated code to use only matrices
`false` (default) | `true`

Indicator for the generated code to use only matrices, to make it compatible with MATLAB Coder, specified as the comma-separated pair consisting of `'MatrixOnly'` and either `true` or `false`.

Example: `'MatrixOnly',true`

Data Types: `logical`

`'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
```

See Also

generateSimulink | genFunction

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.

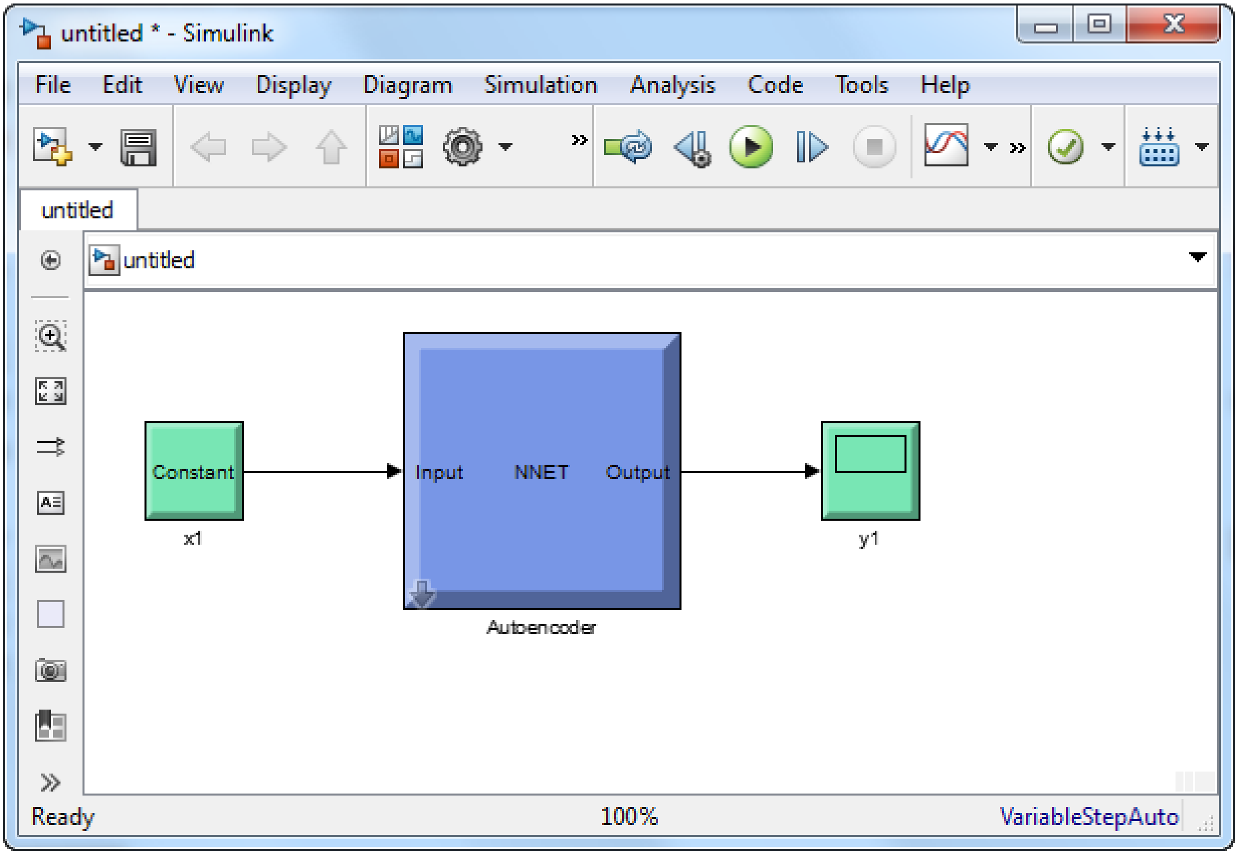
Train an autoencoder with a hidden layer containing 25 neurons.

```
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 = house_dataset;
```

X is a 13-by-506 matrix defining thirteen attributes of 506 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.

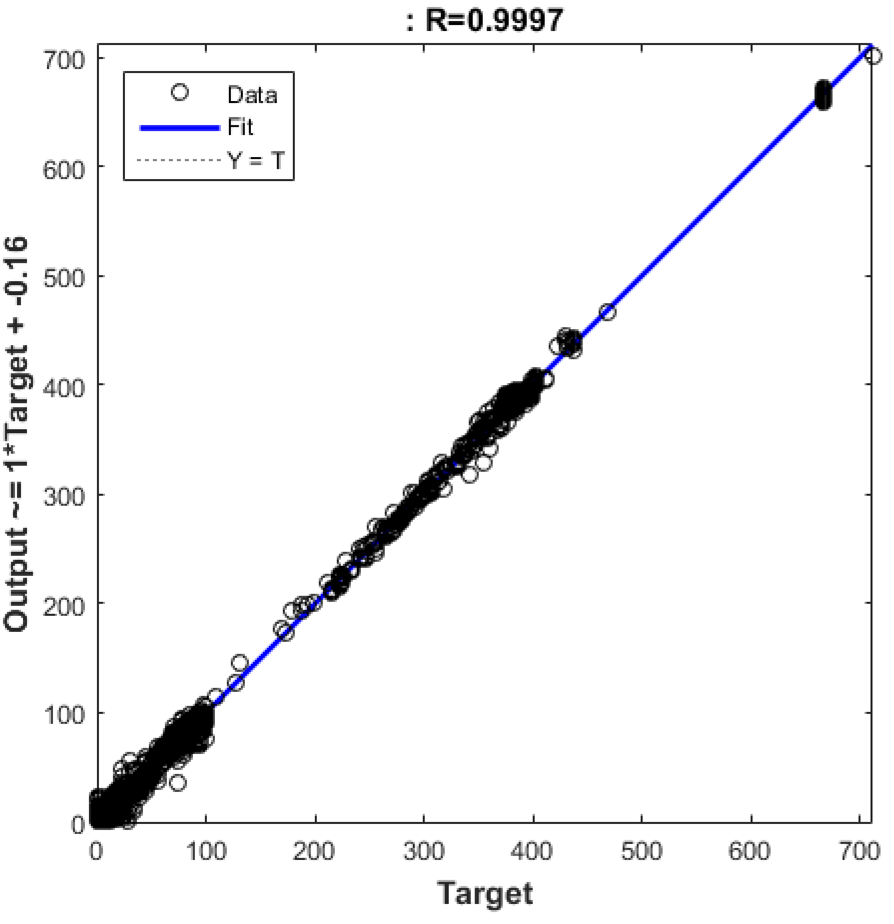
```
[C,S,B] = regression(X,Xpred,'one')
```

```
C =

    0.9997


S =

    1.0005


B =

   -0.1606
```

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`.

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.

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 = digitsmall_dataset;
```

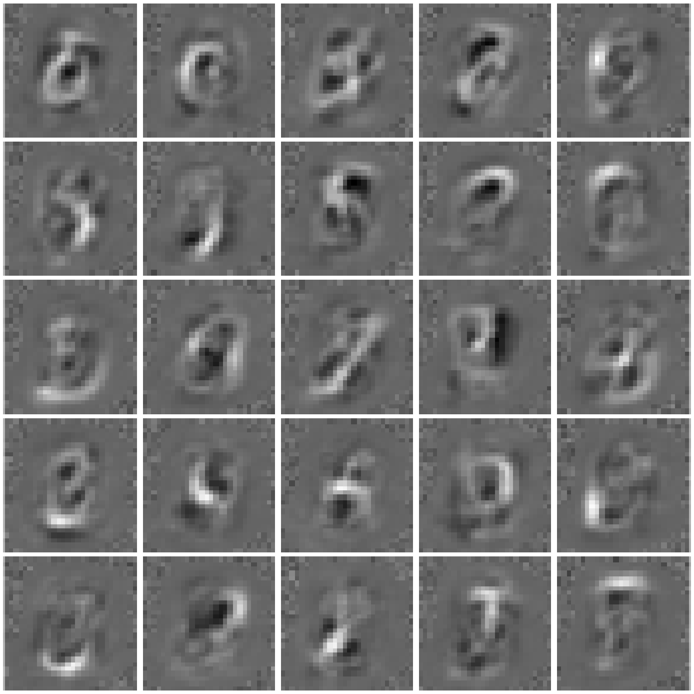
The training data is a 1-by-500 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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);
```



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

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');
```

```
autoenc =

  Autoencoder with properties:

               HiddenSize: 10
    EncoderTransferFunction: 'satlin'
             EncoderWeights: [10x4 double]
              EncoderBiases: [10x1 double]
```

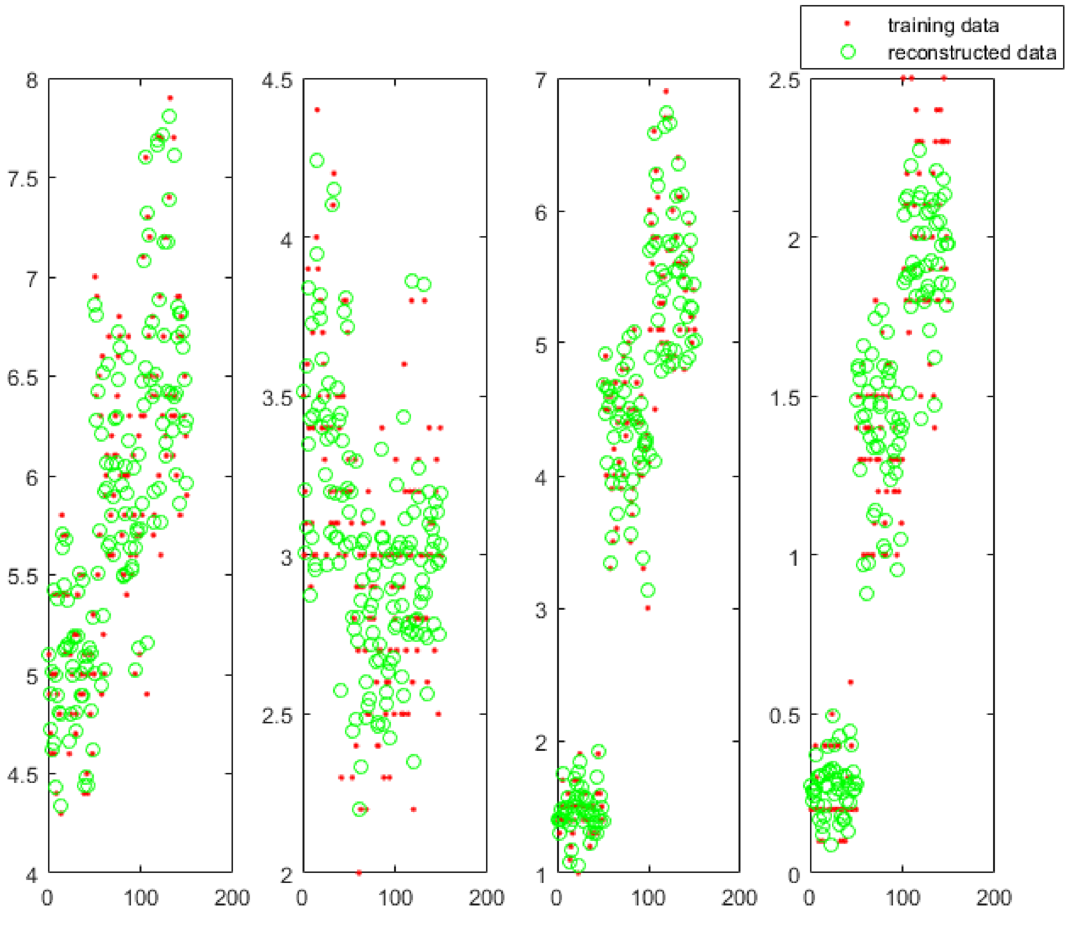
```
    DecoderTransferFunction: 'purelin'
             DecoderWeights: [4x10 double]
              DecoderBiases: [4x1 double]
                  ScaleData: 1
```

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.

```
h = figure()
for i = 1:4
subplot(1,4,i);
plot(X(i,:),'r.');
hold on
plot(xReconstructed(i,:),'go');
hold off;
end
legend('training data','reconstructed data','Location','Best');
```



Reconstruct Handwritten Digit Images

Load the training data.

```
X = digittrain_dataset;
```

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(X,hiddenSize,...
        'L2WeightRegularization',0.004,...
        'SparsityRegularization',4,...
        'SparsityProportion',0.15);
```

Load the test data.

```
x = digittest_dataset;
```

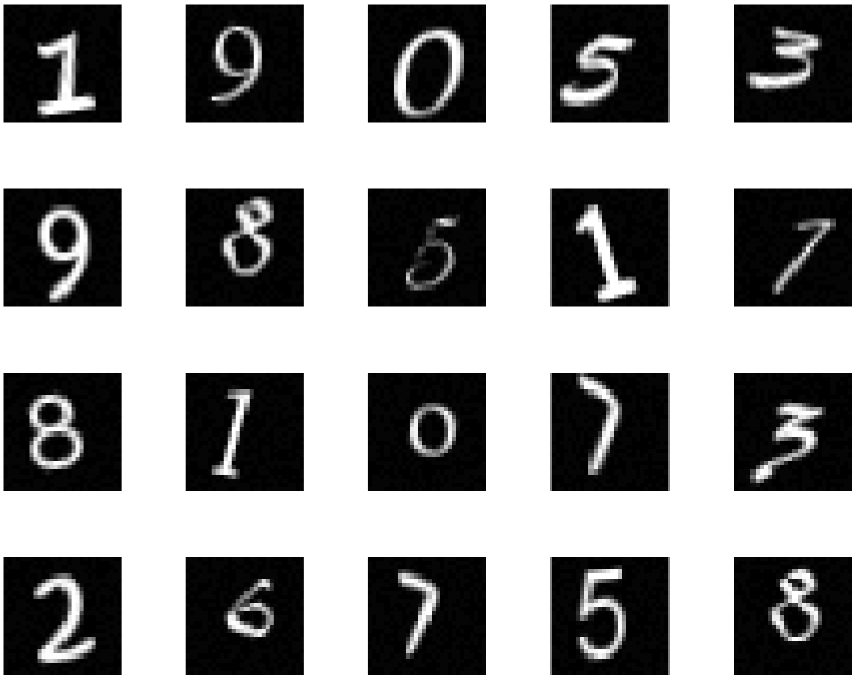
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Reconstruct the test image data using the trained autoencoder, `autoenc`.

```
xReconstructed = predict(autoenc,x);
```

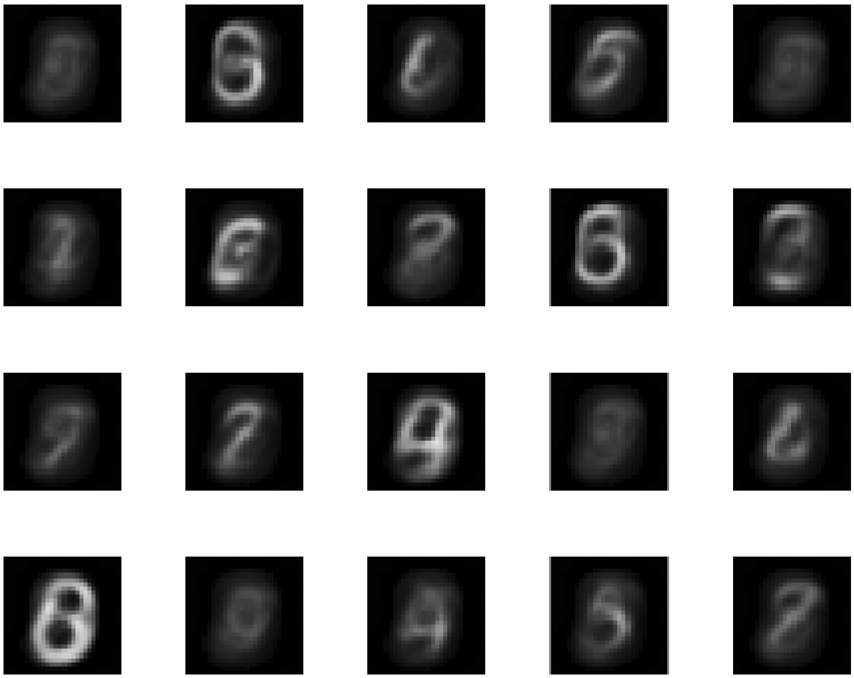
View the actual test data.

```
figure;
for i = 1:20
    subplot(4,5,i);
    imshow(X{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.

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`.

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` .

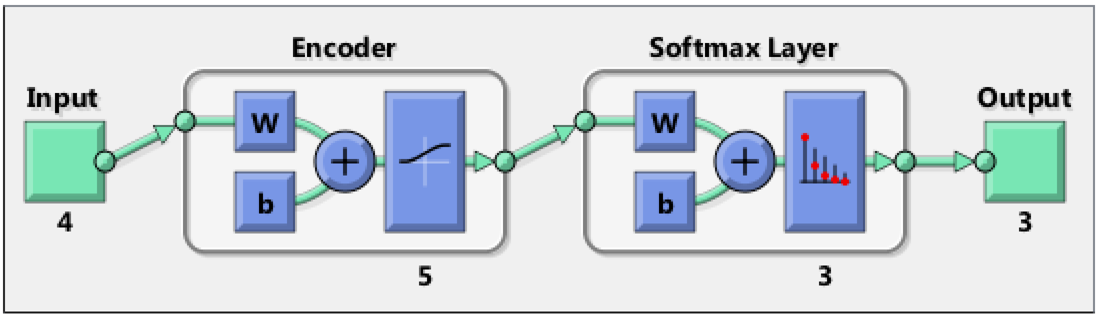
```
softnet = trainSoftmaxLayer(features,T);
```

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”

See Also

Autoencoder | `trainAutoencoder`

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;
```

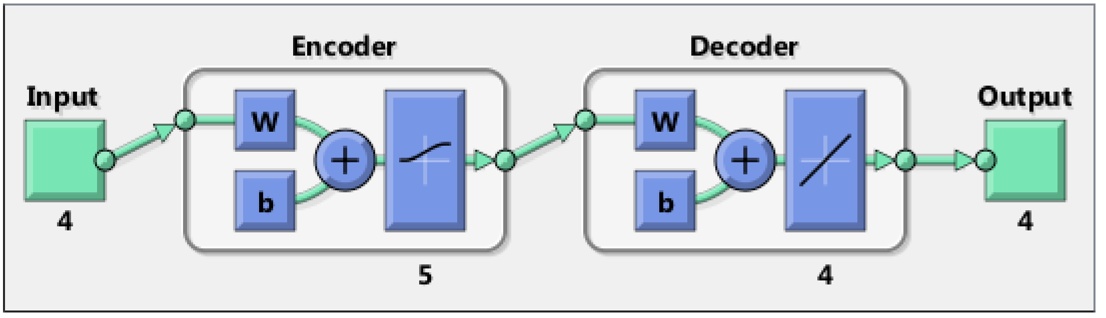
Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
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