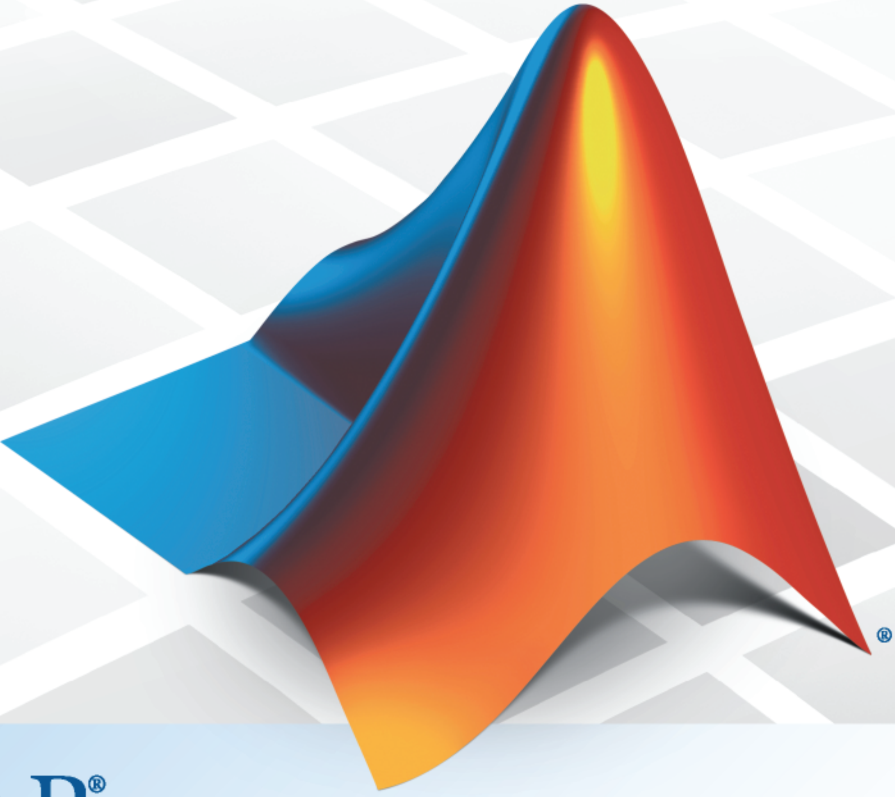
Neural Network Toolbox™ 7
User's Guide

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth



MATLAB®

MathWorks®
Accelerating the pace of engineering and science

How to Contact MathWorks

🖳	`www.mathworks.com`	Web
	`comp.soft-sys.matlab`	Newsgroup
	`www.mathworks.com/contact_TS.html`	Technical support
@	`suggest@mathworks.com`	Product enhancement suggestions
	`bugs@mathworks.com`	Bug reports
	`doc@mathworks.com`	Documentation error reports
	`service@mathworks.com`	Order status, license renewals, passcodes
	`info@mathworks.com`	Sales, pricing, and general information

☎ 508-647-7000 (Phone)

📠 508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Neural Network Toolbox™ User's Guide

Trademarks

Patents

Revision History

June 1992	First printing	
April 1993	Second printing	
January 1997	Third printing	
July 1997	Fourth printing	
January 1998	Fifth printing	Revised for Version 3 (Release 11)
September 2000	Sixth printing	Revised for Version 4 (Release 12)
June 2001	Seventh printing	Minor revisions (Release 12.1)
July 2002	Online only	Minor revisions (Release 13)
January 2003	Online only	Minor revisions (Release 13SP1)
June 2004	Online only	Revised for Version 4.0.3 (Release 14)
October 2004	Online only	Revised for Version 4.0.4 (Release 14SP1)
October 2004	Eighth printing	Revised for Version 4.0.4
March 2005	Online only	Revised for Version 4.0.5 (Release 14SP2)
March 2006	Online only	Revised for Version 5.0 (Release 2006a)
September 2006	Ninth printing	Minor revisions (Release 2006b)
March 2007	Online only	Minor revisions (Release 2007a)
September 2007	Online only	Revised for Version 5.1 (Release 2007b)
March 2008	Online only	Revised for Version 6.0 (Release 2008a)
October 2008	Online only	Revised for Version 6.0.1 (Release 2008b)
March 2009	Online only	Revised for Version 6.0.2 (Release 2009a)
September 2009	Online only	Revised for Version 6.0.3 (Release 2009b)
March 2010	Online only	Revised for Version 6.0.3 (Release 2010a)
September 2010	Online only	Revised for Version 7.0 (Release 2010b)

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 Chapter 4, "Dynamic Networks," and in programming the neural network controllers described in Chapter 5, "Control Systems."

Permissions

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

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. Demonstration programs from the book are used in various chapters of this user's guide. (You can find all the book demonstration programs in the Neural Network Toolbox software by typing `nnd`.)

This book can be obtained from John Stovall at (303) 492-3648, or by e-mail at `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 e-mail at `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.

Contents

1 Getting Started

2 Network Objects, Data and Training Styles

3 Multilayer Networks and Backpropagation Training

4 Dynamic Networks

5 Control Systems

6 Radial Basis Networks

7 Self-Organizing and Learning Vector Quantization Nets

8 Adaptive Filters and Adaptive Training

9 Advanced Topics

10 Historical Networks

11 Network Object Reference

12 Function Reference

13 Functions — Alphabetical List

A Mathematical Notation

B Blocks for the Simulink® Environment

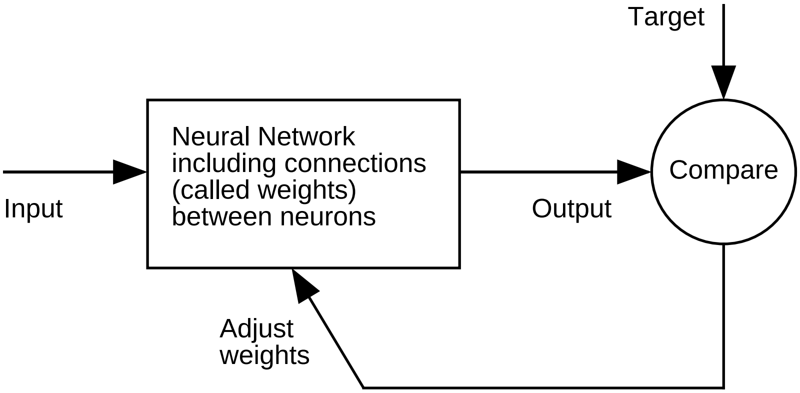
C Code Notes

Getting Started

Product 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. There, 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.

This chapter explains how to use four graphical tools for training neural networks to solve problems in function fitting, pattern recognition, clustering, and time series. Using these four tools will give you an excellent introduction to the use of the Neural Network Toolbox™ software.

Using the Toolbox and Its Documentation

There are four ways you can use the Neural Network Toolbox™ software. The first way is through the four graphical user interfaces (GUIs) that are described in this chapter. (You can open these GUIs from a master GUI, which you can open with the command `nnstart`.) These provide a quick and easy way to access the power of the toolbox for the following tasks:

- Function fitting
- Pattern recognition
- Data clustering
- Time series analysis

The second way to use the toolbox is through basic command-line operations. The command-line operations offer more flexibility than the GUIs, but with some added complexity. This chapter introduces some of the command-line functions, but the next seven chapters cover command-line operations in more detail. The next two chapters are important to understanding the use of the command line, and the fundamentals of training neural networks. You should read them before advancing to later topics.

- Chapter 2, "Network Objects, Data and Training Styles," presents the fundamentals of the neuron model and the architectures of neural networks. It also describes the network object, which is used by the Neural Network Toolbox™ software to store all of the information that defines a neural network. It is important to understand the structure of the network object, especially when using command-line operations. This chapter also describes how data is stored and used in the toolbox, and how networks are trained.
- Chapter 3, "Multilayer Networks and Backpropagation Training," explains the basic steps involved in designing the multilayer network. This network is the workhorse of the toolbox, and it can be used for both function fitting and pattern recognition. Most of the design steps for this network can be applied to the design of any other network in the toolbox.

If this is your first experience with the toolbox, the GUIs provide the best introduction. In addition, the GUIs 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 GUIs 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 will still be able to train them using existing toolbox training functions (as long as the network components are differentiable). Customizing the toolbox is described in Chapter 9, “Advanced Topics”. An example of creating and training a customized network is given in “Training Custom Networks” in Chapter 4.

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 wizards 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 GUIs described in this chapter form an important part of the documentation for the Neural Network Toolbox™ software. The GUIs 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 GUIs can automatically generate both simple and advanced MATLAB® scripts that can reproduce the steps performed by the GUI, but with the option to override default settings. These scripts can provide you with a template 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 the GUIs.

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 chapter will demonstrate 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

Industry	Business Applications
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 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
Speech	Speech recognition, speech compression, vowel classification, and text-to-speech synthesis
Securities	Market analysis, automatic bond rating, and stock trading advisory systems

Industry	**Business Applications**
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 chapter, 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 six primary steps. (Data collection, while important, generally occurs outside the MATLAB® environment, so it is step 0.)

0 Collect data.

1 Create the network.

2 Configure the network.

3 Initialize the weights and biases.

4 Train the network.

5 Validate the network.

6 Use the network.

You will follow these steps using both the GUI tools and command-line operations in the next four sections. More detailed discussions of the six design steps are in Chapter 2, "Network Objects, Data and Training Styles" and Chapter 3, "Multilayer Networks and Backpropagation Training."

Fitting a Function

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 [HaRu78]. 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 1-10.
- Use command-line functions, as described in “Using Command-Line Functions” on page 1-21.

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 “Sample Data Sets” on page 1-89). 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 Chapter 2, “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:

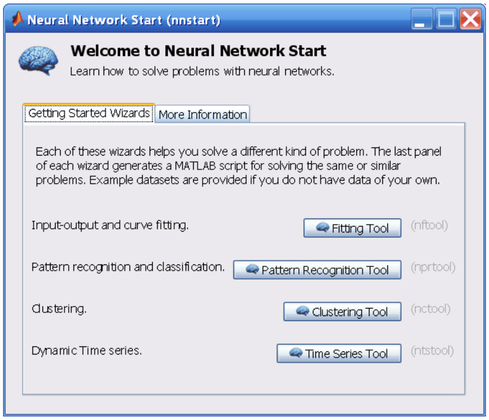
```
inputs = [0 1 0 1; 0 0 1 1];
targets = [0 0 0 1];
```

The next section demonstrates 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
```



Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

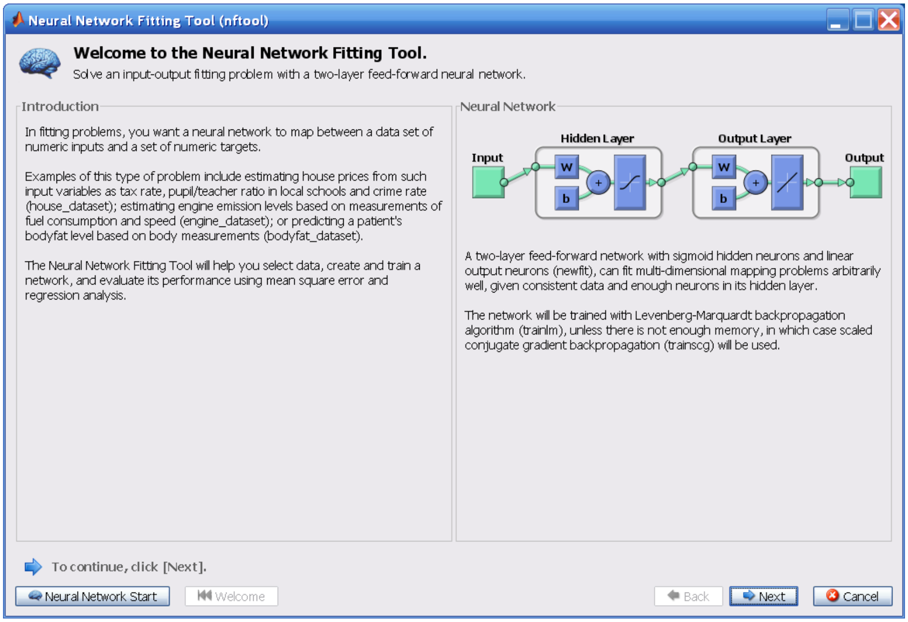
Input-output and curve fitting.

Pattern recognition and classification.

Clustering.

Dynamic Time series.

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



Welcome to the Neural Network Fitting Tool.

Solve an input-output fitting problem with a two-layer feed-forward neural network.

In fitting problems, you want a neural network to map between a data set of numeric inputs and a set of numeric targets.

Examples of this type of problem include estimating house prices from such input variables as tax rate, pupil/teacher ratio in local schools and crime rate (house_dataset); estimating engine emission levels based on measurements of fuel consumption and speed (engine_dataset); or predicting a patient's bodyfat level based on body measurements (bodyfat_dataset).

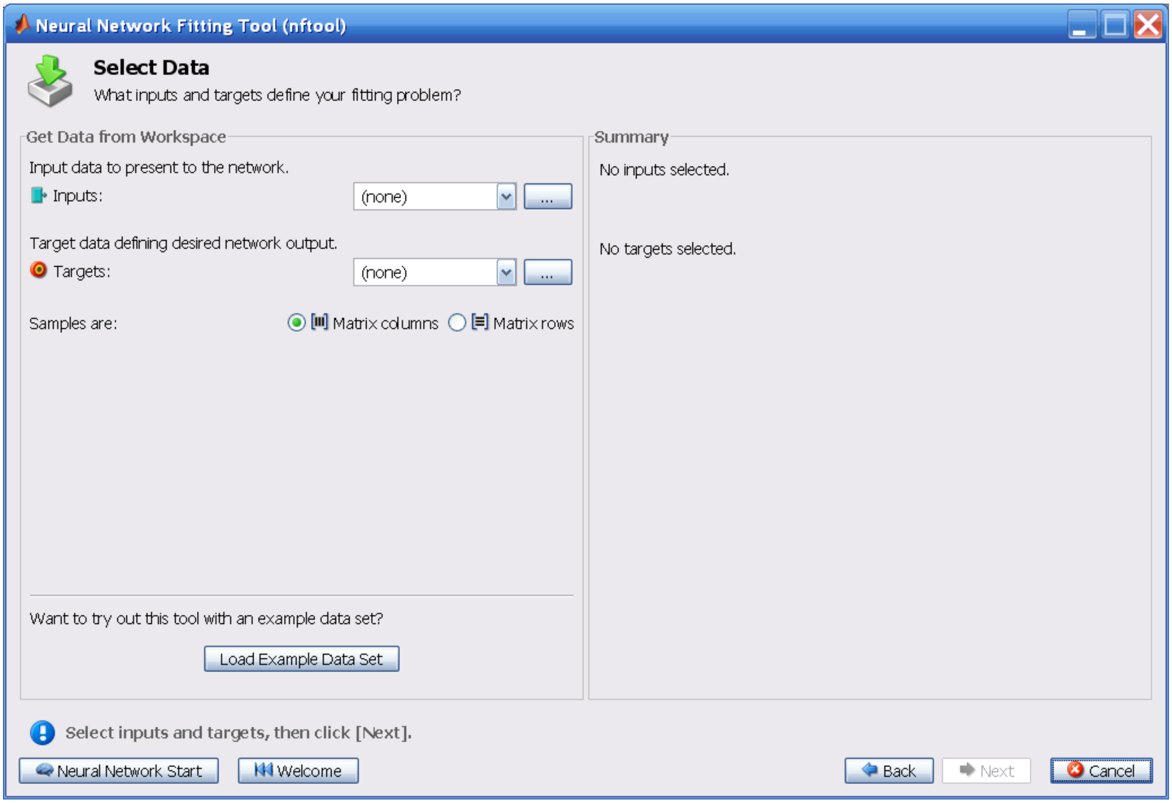
The Neural Network Fitting Tool will help you select data, create and train a network, and evaluate its performance using mean square error and regression analysis.

A two-layer feed-forward network with sigmoid hidden neurons and linear output neurons (newfit), can fit multi-dimensional mapping problems arbitrarily well, given consistent data and enough neurons in its hidden layer.

The network will be trained with Levenberg-Marquardt backpropagation algorithm (trainlm), unless there is not enough memory, in which case scaled conjugate gradient backpropagation (trainscg) will be used.

To continue, click [Next].

3 Click **Next** to proceed.



Neural Network Fitting Tool (nftool)

Select Data

What inputs and targets define your fitting problem?

Get Data from Workspace

Input data to present to the network.

Inputs: (none)

Target data defining desired network output.

Targets: (none)

Samples are: ● Matrix columns ○ Matrix rows

Want to try out this tool with an example data set?

Summary

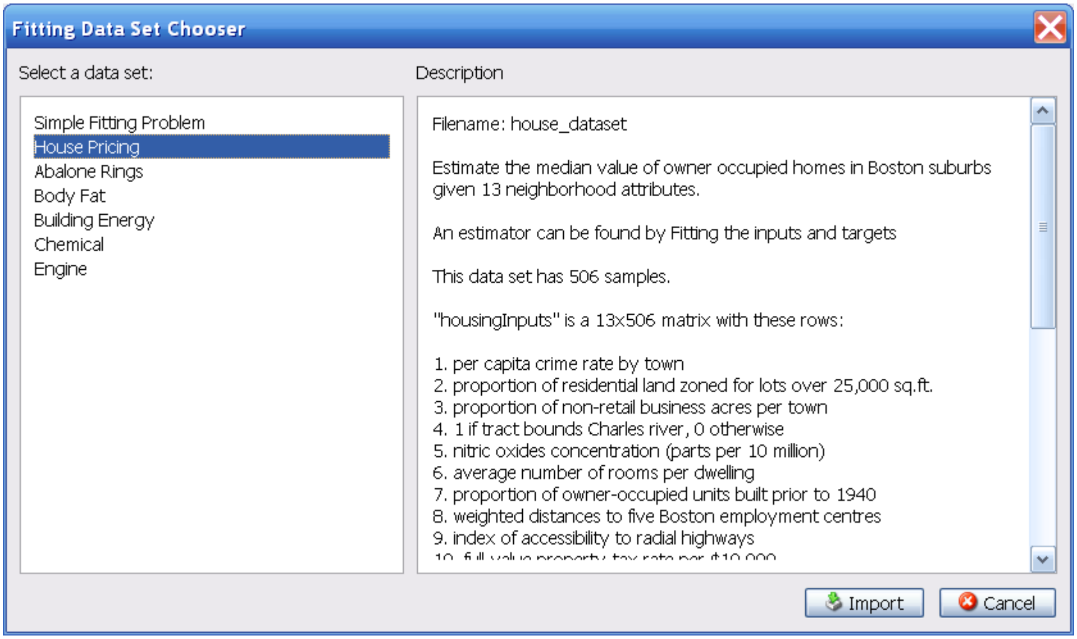
No inputs selected.

No targets selected.

Select inputs and targets, then click [Next].

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.



Filename: house_dataset

Estimate the median value of owner occupied homes in Boston suburbs given 13 neighborhood attributes.

An estimator can be found by Fitting the inputs and targets

This data set has 506 samples.

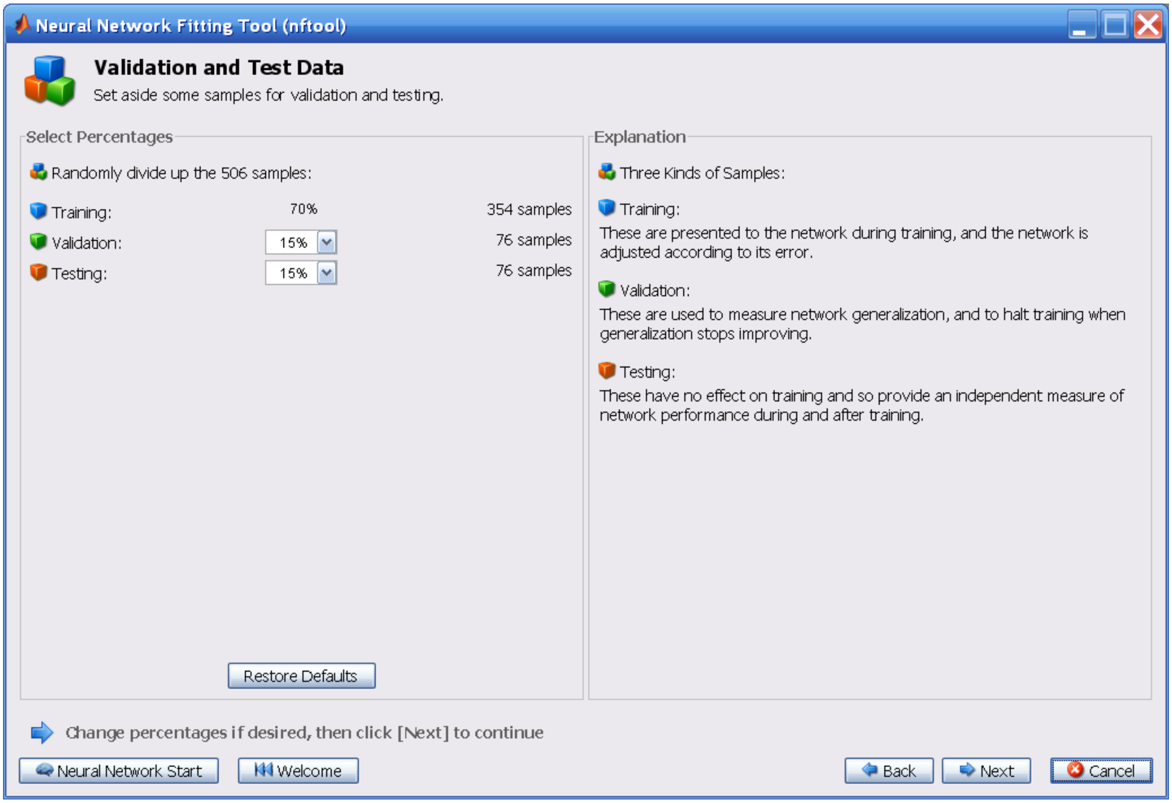
"housingInputs" is a 13x506 matrix with these rows:

1. per capita crime rate by town
2. proportion of residential land zoned for lots over 25,000 sq.ft.
3. proportion of non-retail business acres per town
4. 1 if tract bounds Charles river, 0 otherwise
5. nitric oxides concentration (parts per 10 million)
6. average number of rooms per dwelling
7. proportion of owner-occupied units built prior to 1940
8. weighted distances to five Boston employment centres
9. index of accessibility to radial highways
10. full-value property-tax rate per $10,000

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.



Validation and Test Data

Set aside some samples for validation and testing.

Randomly divide up the 506 samples:

Training: 70% 354 samples

Validation: 15% 76 samples

Testing: 15% 76 samples

Three Kinds of Samples:

Training:
These are presented to the network during training, and the network is adjusted according to its error.

Validation:
These are used to measure network generalization, and to halt training when generalization stops improving.

Testing:
These have no effect on training and so provide an independent measure of network performance during and after training.

Change percentages if desired, then click [Next] to continue

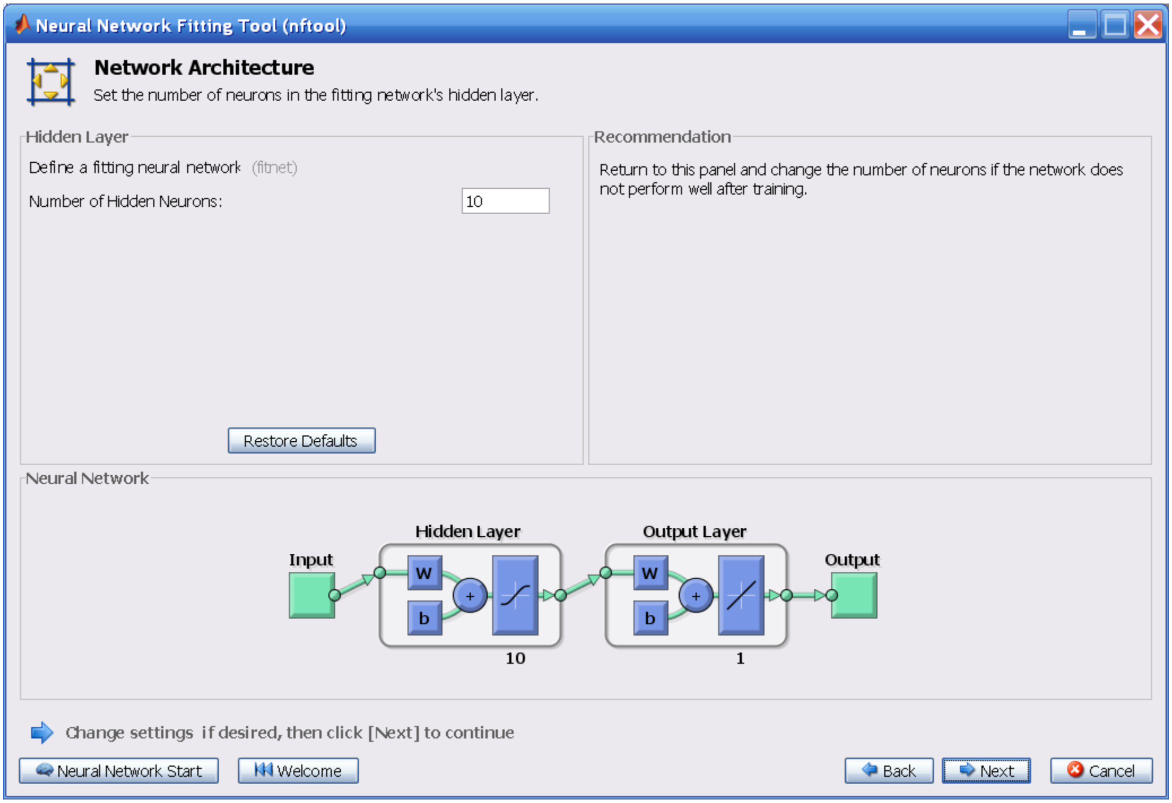
With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

- 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" on page 3-10 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. (This network is discussed in more detail in Chapter 3, "Multilayer Networks and Backpropagation Training" and in Chapters 11-12 of [HDB96].) 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.



Neural Network Fitting Tool (nftool)

Network Architecture

Set the number of neurons in the fitting network's hidden layer.

Hidden Layer

Define a fitting neural network (fitnet)

Number of Hidden Neurons: 10

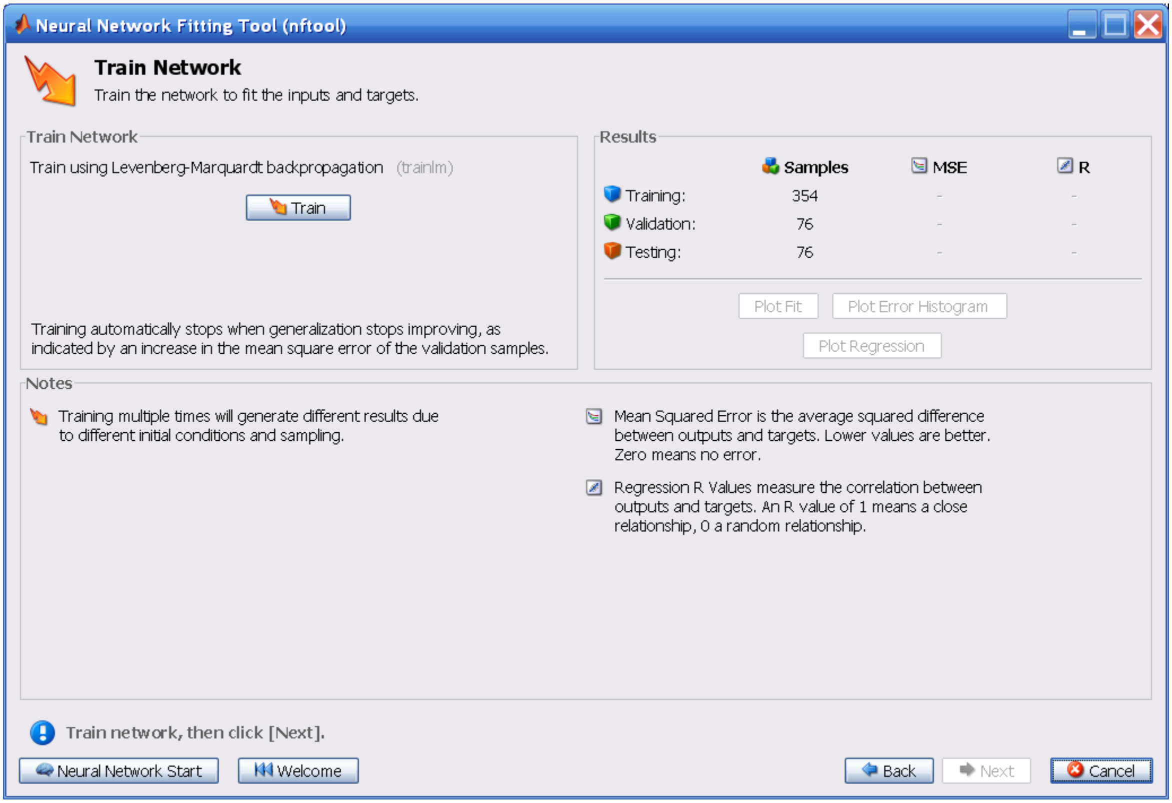
Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

Neural Network

Change settings if desired, then click [Next] to continue

8 Click **Next**.



Train Network

Train the network to fit the inputs and targets.

Train Network

Train using Levenberg-Marquardt backpropagation (trainlm)

Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Results

	Samples	MSE	R
Training:	354	-	-
Validation:	76	-	-
Testing:	76	-	-

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

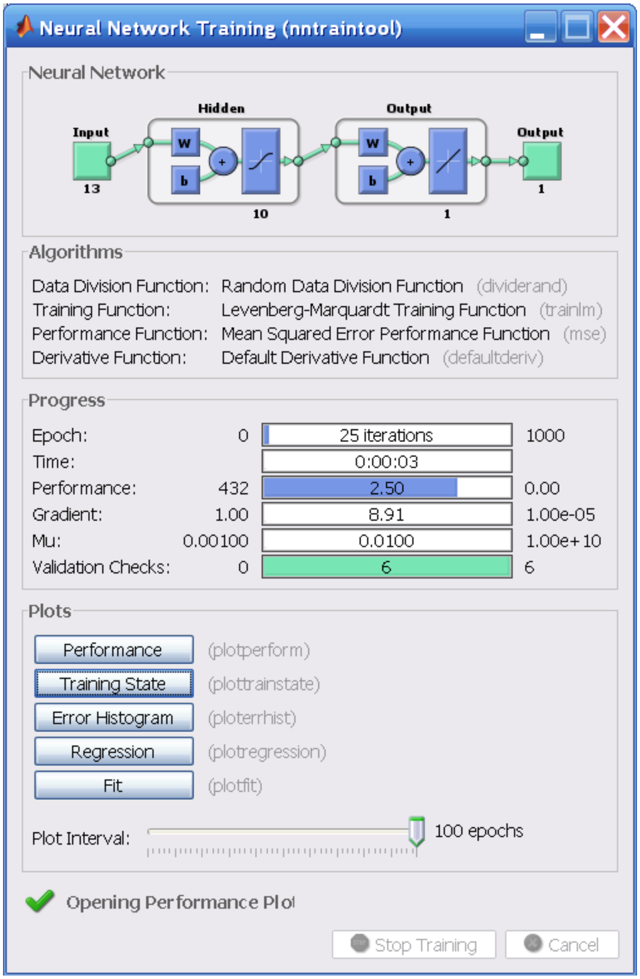
Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Train network, then click [Next].

9 Click **Train.**

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



Data Division Function: Random Data Division Function (dividerand)
Training Function: Levenberg-Marquardt Training Function (trainlm)
Performance Function: Mean Squared Error Performance Function (mse)
Derivative Function: Default Derivative Function (defaultderiv)

Epoch:	0	25 iterations	1000
Time:		0:00:03	
Performance:	432	2.50	0.00
Gradient:	1.00	8.91	1.00e-05
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

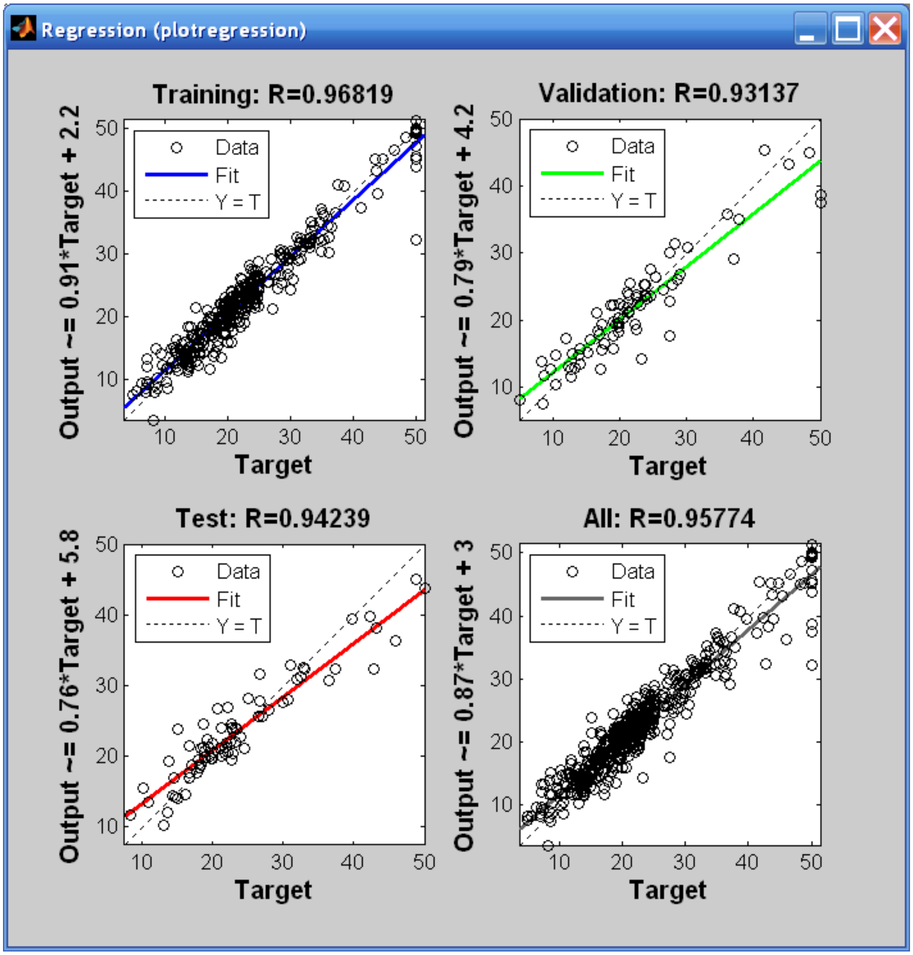
(plotperform)
(plottrainstate)
(ploterrhist)
(plotregression)
(plotfit)

Plot Interval: 100 epochs

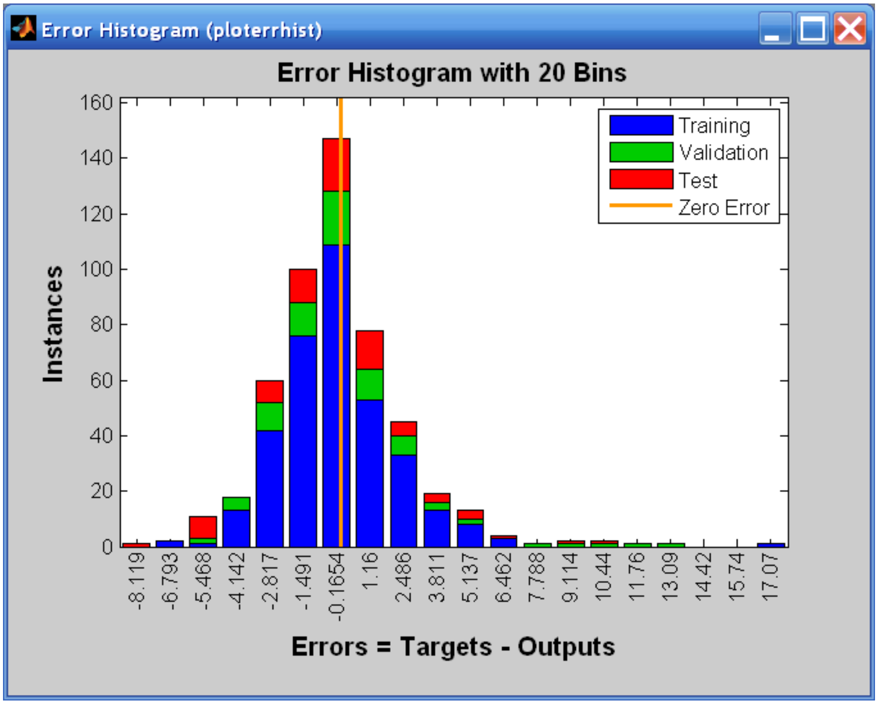
✓ Opening Performance Plot

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.

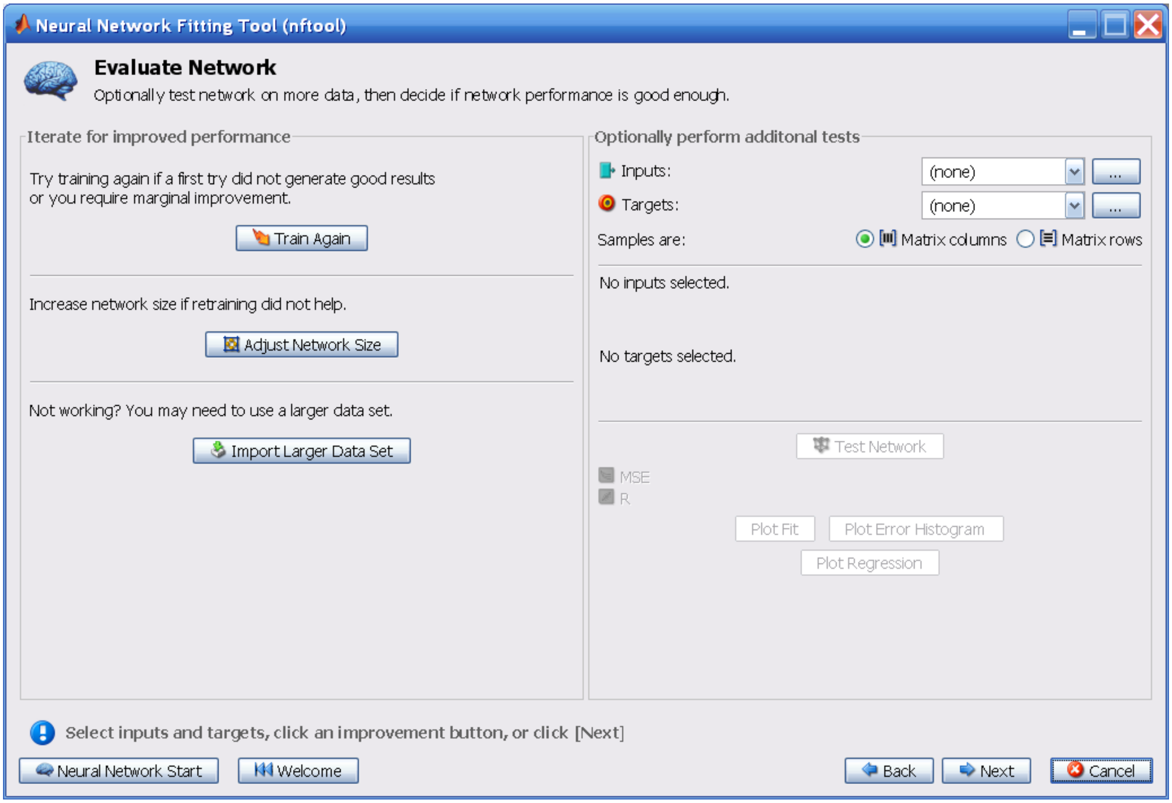

Regression (plotregression)
Training: R=0.96819
Output ~= 0.91*Target + 2.2
Validation: R=0.93137
Output ~= 0.79*Target + 4.2
Test: R=0.94239
Output ~= 0.76*Target + 5.8
All: R=0.95774
Output ~= 0.87*Target + 3
Data
Fit
Y = T
Target
10
20
30
40
50

11 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.

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



Neural Network Fitting Tool (nftool)

Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

Try training again if a first try did not generate good results or you require marginal improvement.

Increase network size if retraining did not help.

Not working? You may need to use a larger data set.

Optionally perform additonal tests

Inputs: (none)

Targets: (none)

Samples are: ● Matrix columns ○ Matrix rows

No inputs selected.

No targets selected.

Select inputs and targets, click an improvement button, or click [Next]

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

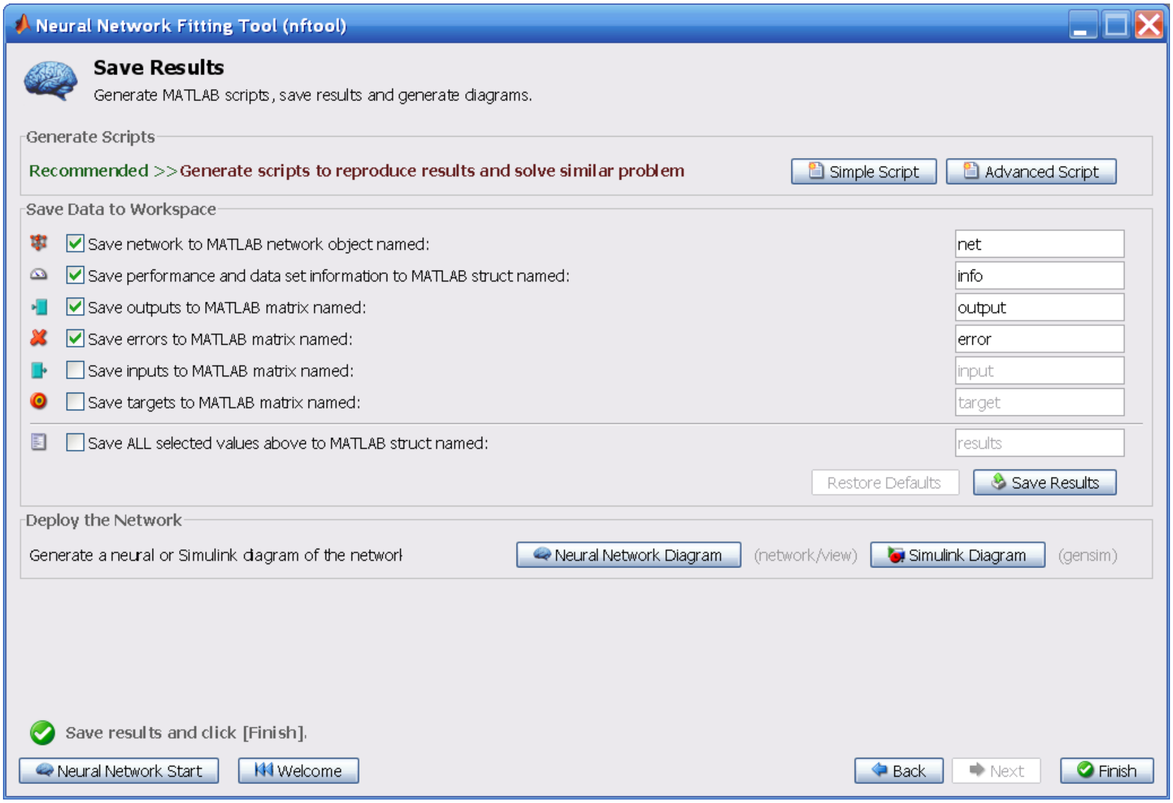
If you are dissatisfied with the network's performance on the original or new data, you can do one of the following:

- 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.

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

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



Neural Network Fitting Tool (nftool)

Save Results

Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >> Generate scripts to reproduce results and solve similar problem

Save Data to Workspace

☑ Save network to MATLAB network object named: net

☑ Save performance and data set information to MATLAB struct named: info

☑ Save outputs to MATLAB matrix named: output

☑ Save errors to MATLAB matrix named: error

□ Save inputs to MATLAB matrix named: input

□ Save targets to MATLAB matrix named: target

□ Save ALL selected values above to MATLAB struct named: results

Deploy the Network

Generate a neural or Simulink diagram of the networl

Save results and click [Finish].

- 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 1-21, 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.

15 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.

0 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 "Sample Data Sets" on page 1-89). 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`.

1 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.

2 Set up the division of data.

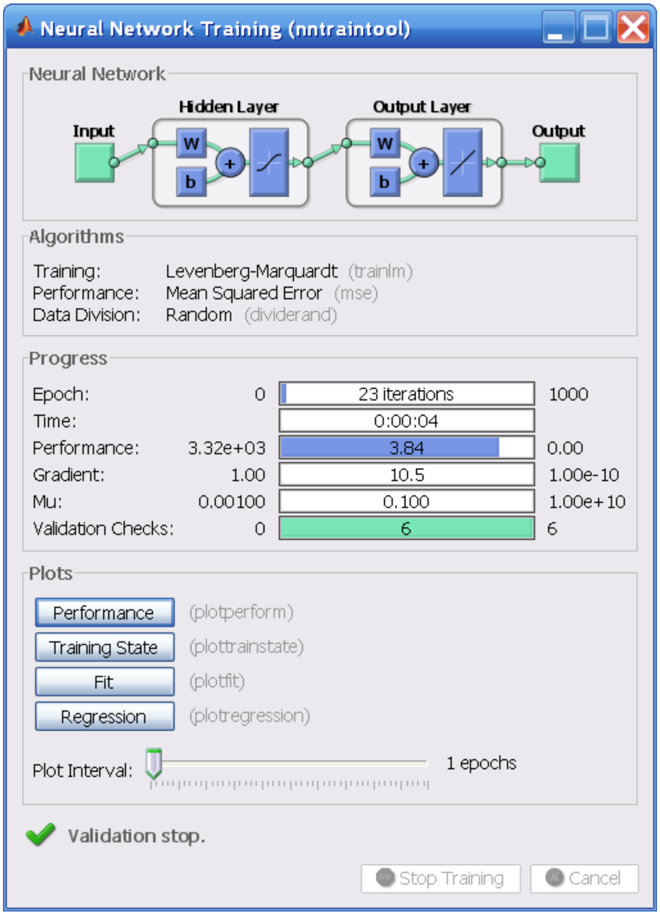
```
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
```

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" on page 3-10 for more discussion of the data division process.)

3 Train the network. The network uses the default Levenberg-Marquardt algorithm for training (`trainlm`). 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**.



Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Data Division: Random (dividerand)

Epoch:	0	23 iterations	1000
Time:		0:00:04	
Performance:	3.32e+03	3.84	0.00
Gradient:	1.00	10.5	1.00e-10
Mu:	0.00100	0.100	1.00e+10
Validation Checks:	0	6	6

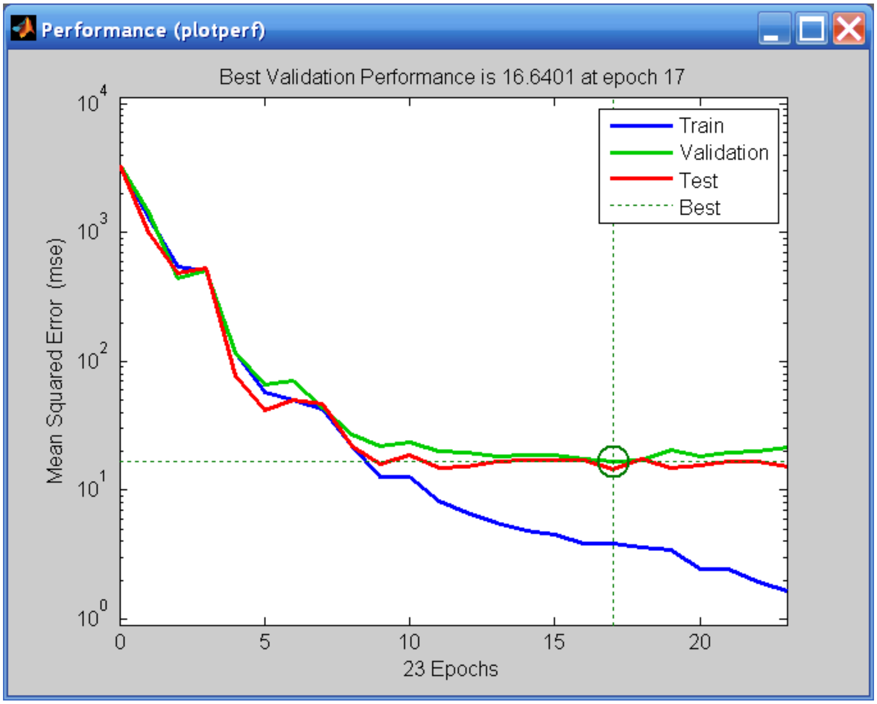
(plotperform)
(plottrainstate)
(plotfit)
(plotregression)

Plot Interval: 1 epochs

✓ Validation stop.

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).



4 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.

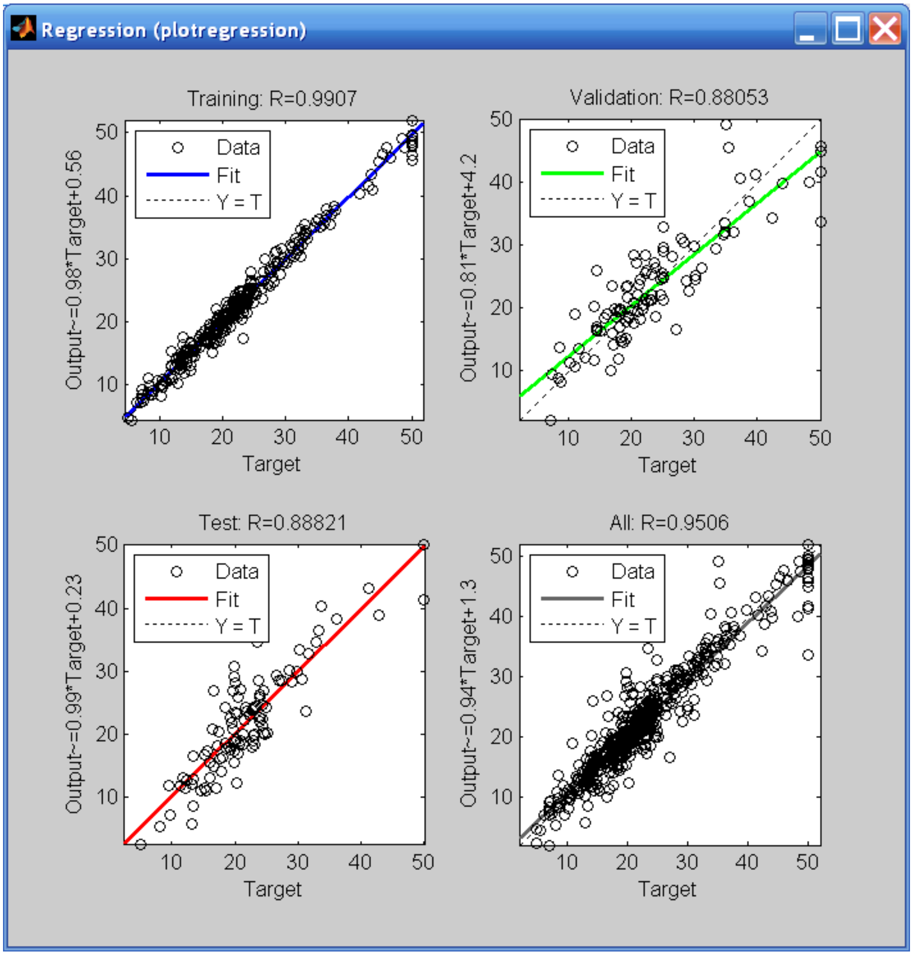
```
outputs = net(inputs);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)
```

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 "Post-Training Analysis (Network Validation)" on page 3-21 for a full description of the training record.)

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

5 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:

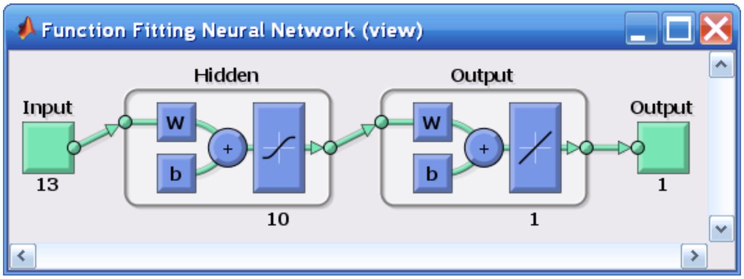
- Reset the initial network weights and biases to new values with `init` and train again (see "Initializing Weights (init)" on page 3-13).
- Increase the number of hidden neurons.
- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see "Training Algorithms" on page 3-15).

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

6 View the network diagram.

```
view(net)
```

This creates the following diagram of the network.



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.

Recognizing Patterns

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. [MuAh94]. 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 1-30.
- Use a command-line solution, as described in "Using Command-Line Functions" on page 1-60.

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 Chapter 2, "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 exclusive-or classification problem as follows:

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

Alternately, target vectors can 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];
```

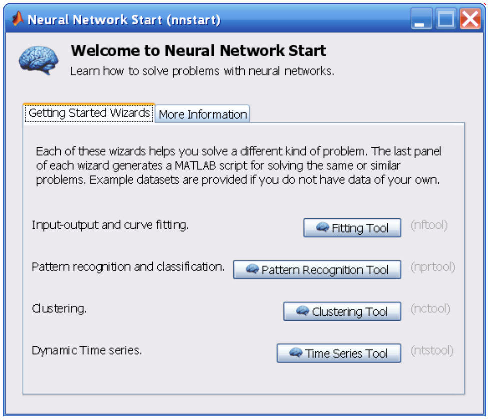
Classification problems involving only two classes can be represented using either format. The targets can consist of either scalar 1/0 elements or two-element vectors, with one element being 1 and the other element being 0.

The next section demonstrates 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
```



Neural Network Start (nnstart)

Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

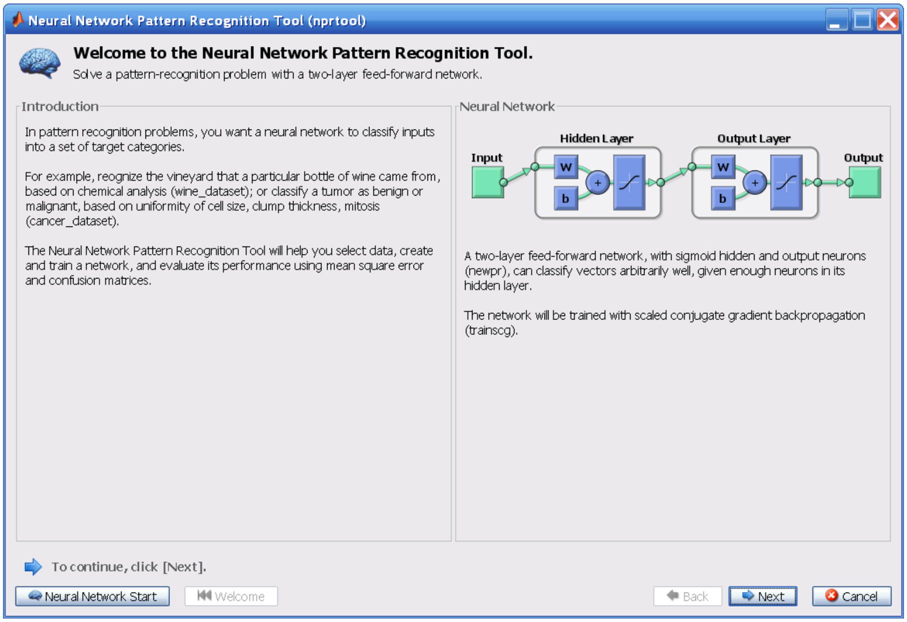
Input-output and curve fitting. Fitting Tool (nftool)

Pattern recognition and classification. Pattern Recognition Tool (nprtool)

Clustering. Clustering Tool (nctool)

Dynamic Time series. Time Series Tool (ntstool)

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



Neural Network Pattern Recognition Tool (nprtool)

Welcome to the Neural Network Pattern Recognition Tool.

Solve a pattern-recognition problem with a two-layer feed-forward network.

Introduction

In pattern recognition problems, you want a neural network to classify inputs into a set of target categories.

For example, reognize the vineyard that a particular bottle of wine came from, based on chemical analysis (wine_dataset); or classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis (cancer_dataset).

The Neural Network Pattern Recognition Tool will help you select data, create and train a network, and evaluate its performance using mean square error and confusion matrices.

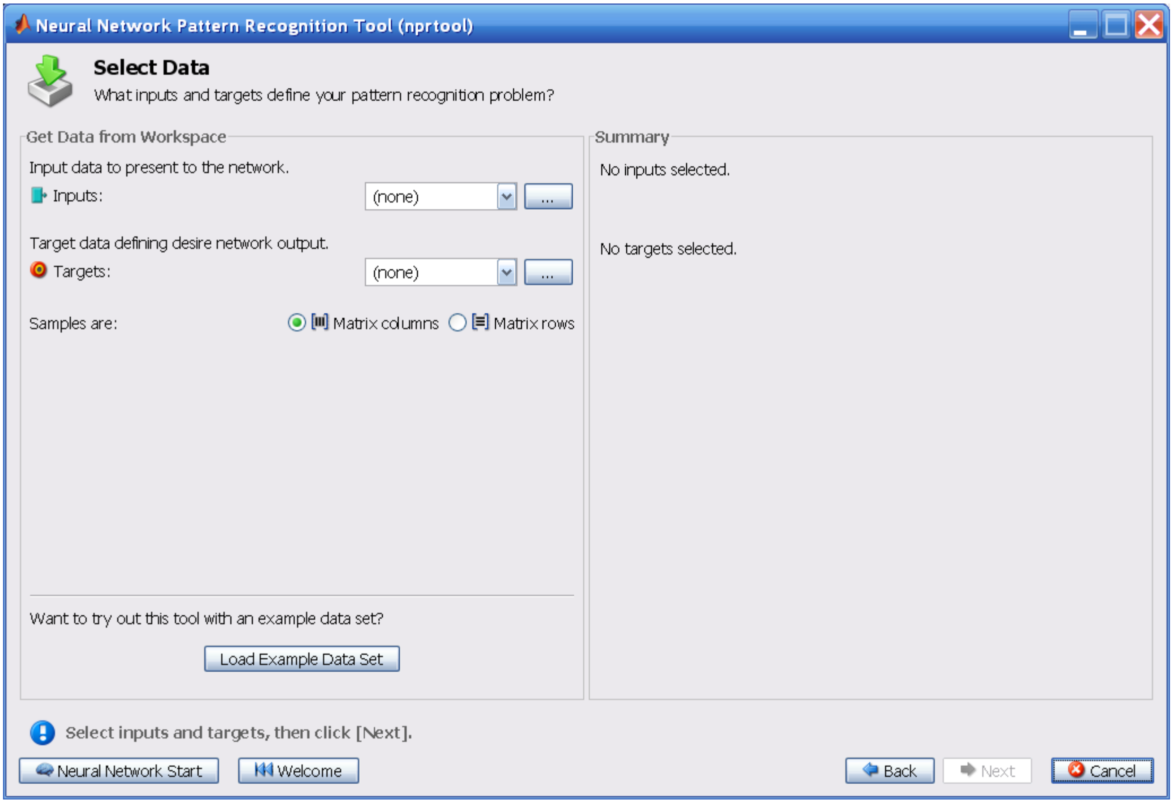
Neural Network

A two-layer feed-forward network, with sigmoid hidden and output neurons (newpr), can classify vectors arbitrarily well, given enough neurons in its hidden layer.

The network will be trained with scaled conjugate gradient backpropagation (trainscg).

To continue, click [Next].

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



Neural Network Pattern Recognition Tool (nprtool)

Select Data

What inputs and targets define your pattern recognition problem?

Get Data from Workspace

Input data to present to the network.

Inputs: (none)

Target data defining desire network output.

Targets: (none)

Samples are: ● Matrix columns ○ Matrix rows

Want to try out this tool with an example data set?

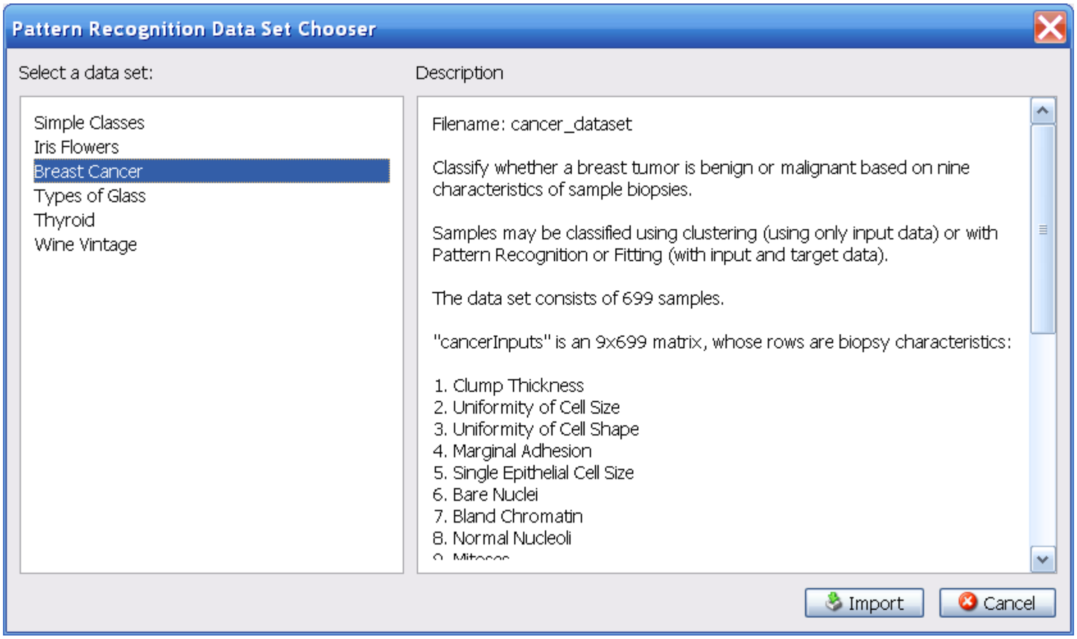
Summary

No inputs selected.

No targets selected.

Select inputs and targets, then click [Next].

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



Filename: cancer_dataset

Classify whether a breast tumor is benign or malignant based on nine characteristics of sample biopsies.

Samples may be classified using clustering (using only input data) or with Pattern Recognition or Fitting (with input and target data).

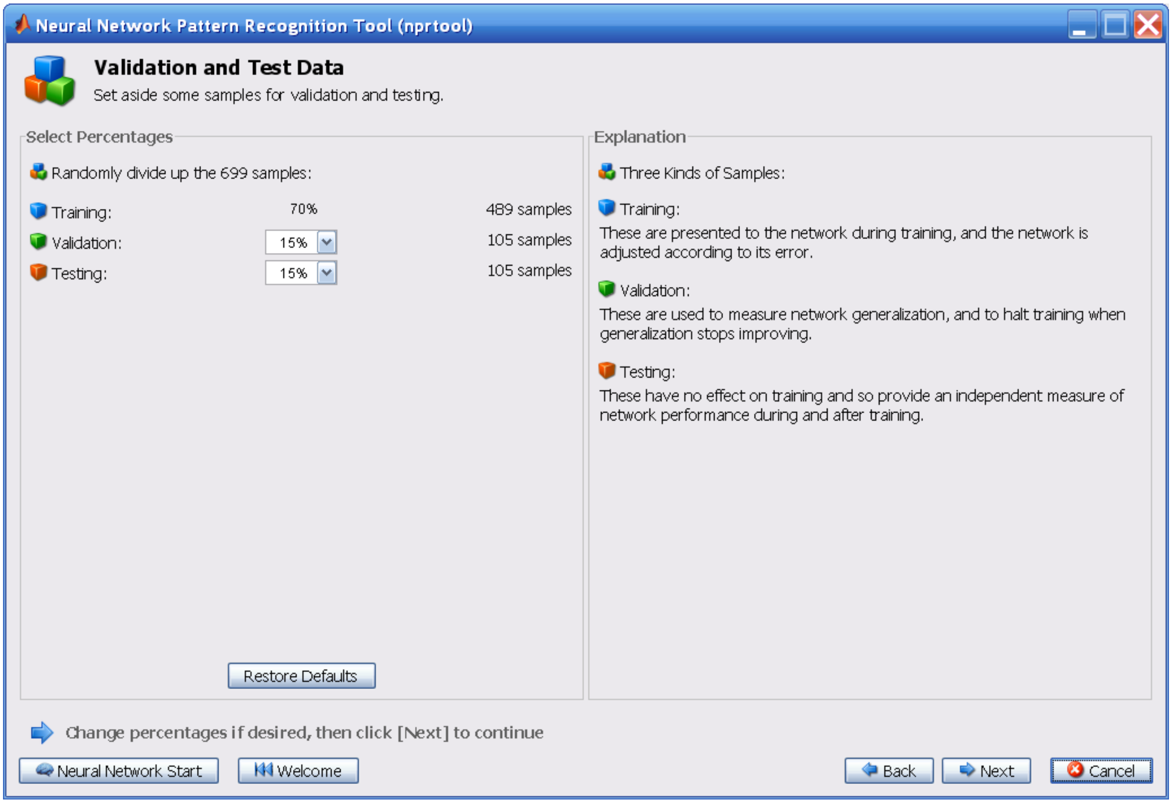
The data set consists of 699 samples.

"cancerInputs" is an 9x699 matrix, whose rows are biopsy characteristics:

1. Clump Thickness
2. Uniformity of Cell Size
3. Uniformity of Cell Shape
4. Marginal Adhesion
5. Single Epithelial Cell Size
6. Bare Nuclei
7. Bland Chromatin
8. Normal Nucleoli
9. Mitoses

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

Set aside some samples for validation and testing.

Randomly divide up the 699 samples:

Training:	70%	489 samples
Validation:	15%	105 samples
Testing:	15%	105 samples

Three Kinds of Samples:

Training:
These are presented to the network during training, and the network is adjusted according to its error.

Validation:
These are used to measure network generalization, and to halt training when generalization stops improving.

Testing:
These have no effect on training and so provide an independent measure of network performance during and after training.

Change percentages if desired, then click [Next] to continue

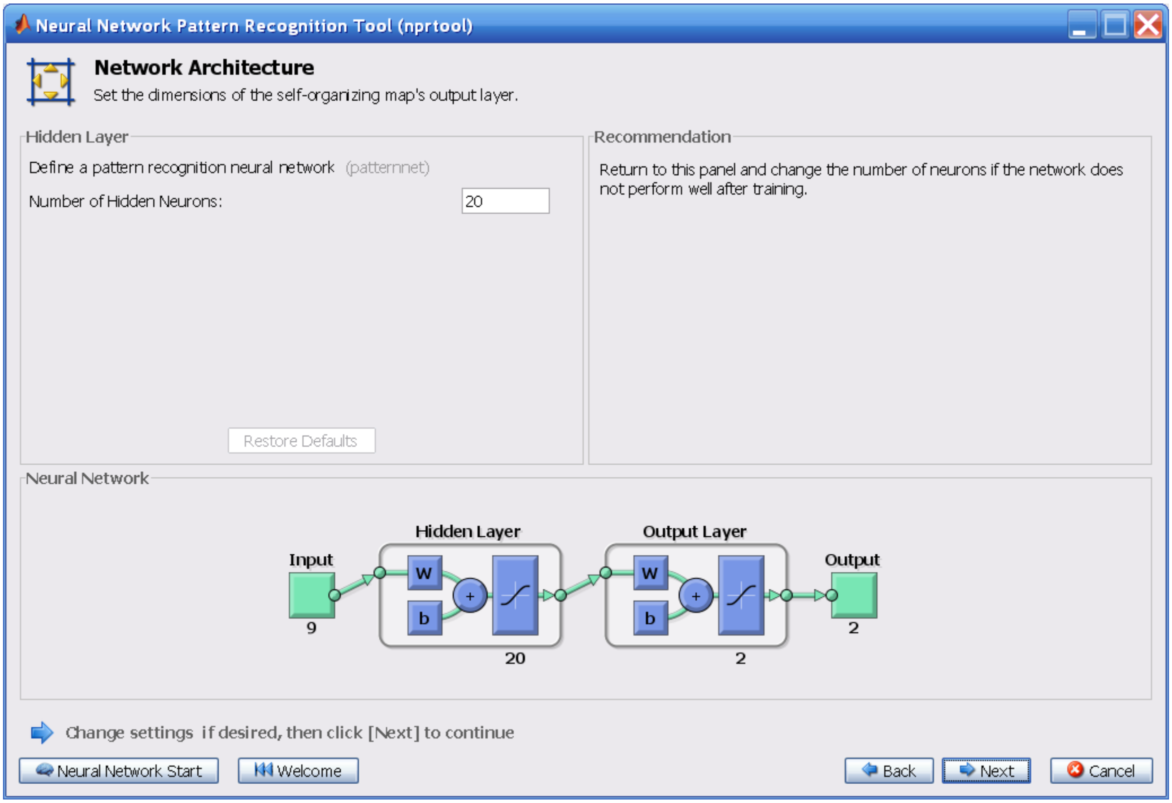
Validation and test data sets are each set to 15% of the original data. With these settings, the input vectors and target vectors will be randomly divided into three sets as follows:

- 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” on page 3-10 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 sigmoid transfer functions in both the hidden layer and the output layer. This network architecture is discussed in more detail in Chapter 2, "Network Objects, Data and Training Styles," Chapter 3, "Multilayer Networks and Backpropagation Training" and in Chapters 11-12 of [HDB96]. 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).



Neural Network Pattern Recognition Tool (nprtool)

Network Architecture

Set the dimensions of the self-organizing map's output layer.

Hidden Layer

Define a pattern recognition neural network (patternnet)

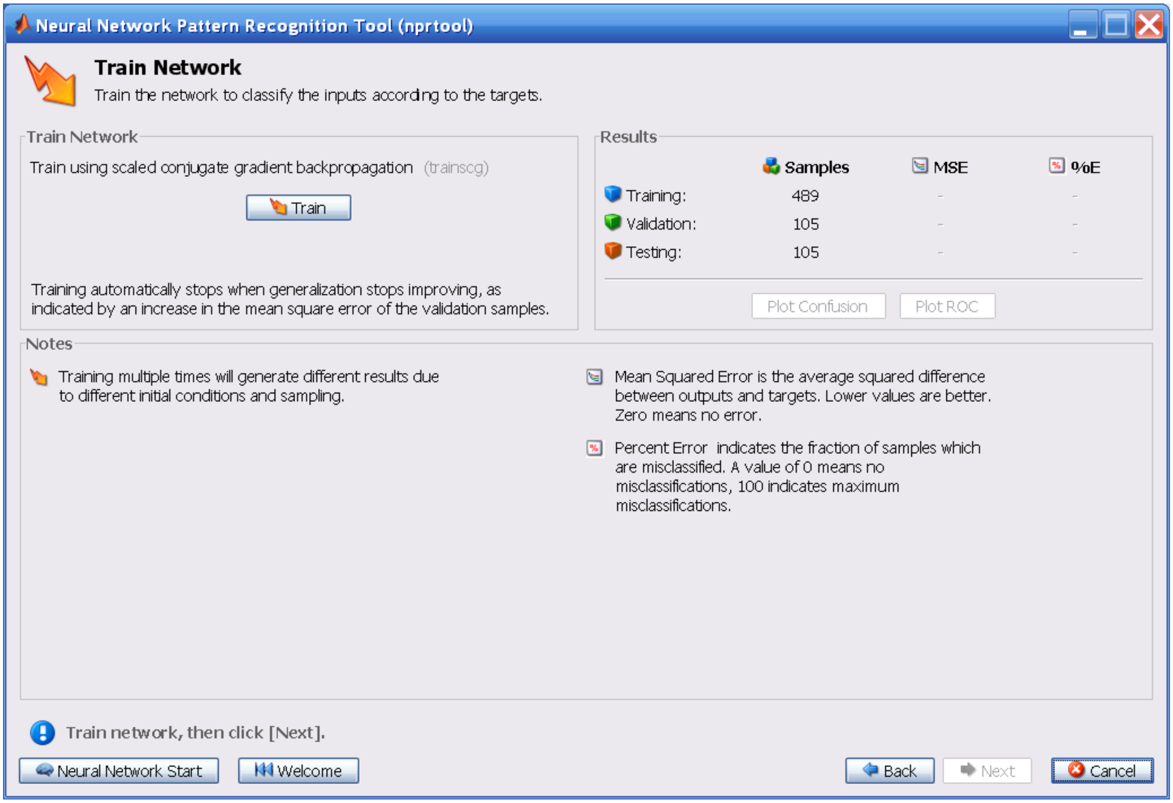
Number of Hidden Neurons: 20

Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

Change settings if desired, then click [Next] to continue

8 Click **Next**.



Neural Network Pattern Recognition Tool (nprtool)

Train Network

Train the network to classify the inputs according to the targets.

Train Network

Train using scaled conjugate gradient backpropagation (trainscg)

Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Results

	Samples	MSE	%E
Training:	489	-	-
Validation:	105	-	-
Testing:	105	-	-

Notes

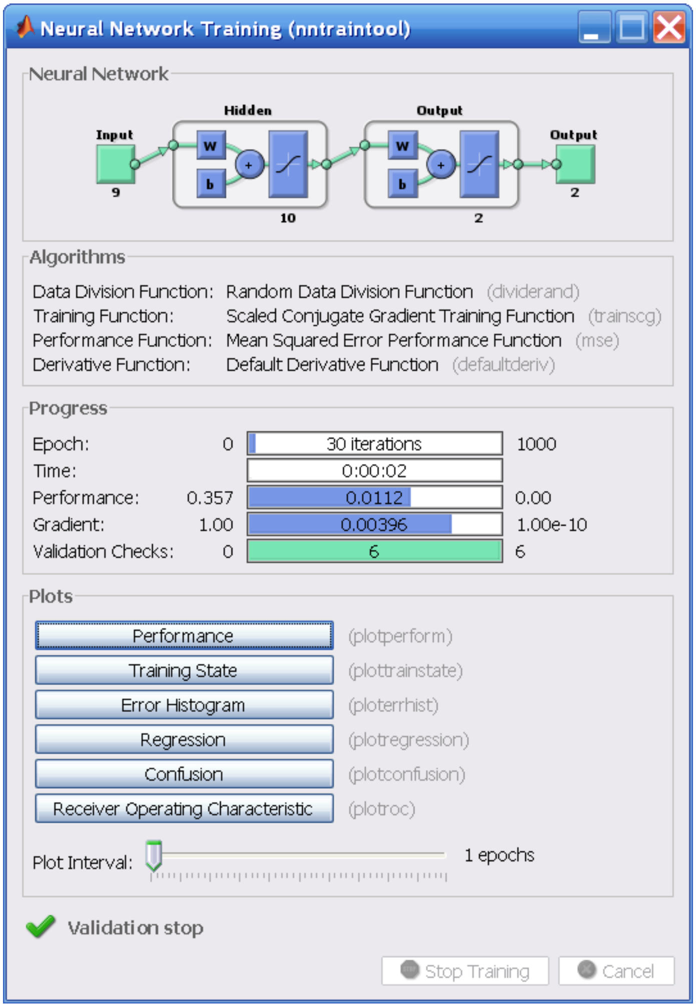
Training multiple times will generate different results due to different initial conditions and sampling.

Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Percent Error indicates the fraction of samples which are misclassified. A value of 0 means no misclassifications, 100 indicates maximum misclassifications.

Train network, then click [Next].

9 Click **Train**.



Algorithms

Data Division Function:	Random Data Division Function (dividerand)
Training Function:	Scaled Conjugate Gradient Training Function (trainscg)
Performance Function:	Mean Squared Error Performance Function (mse)
Derivative Function:	Default Derivative Function (defaultderiv)

Progress

Epoch:	0	30 iterations	1000
Time:		0:00:02	
Performance:	0.357	0.0112	0.00
Gradient:	1.00	0.00396	1.00e-10
Validation Checks:	0	6	6

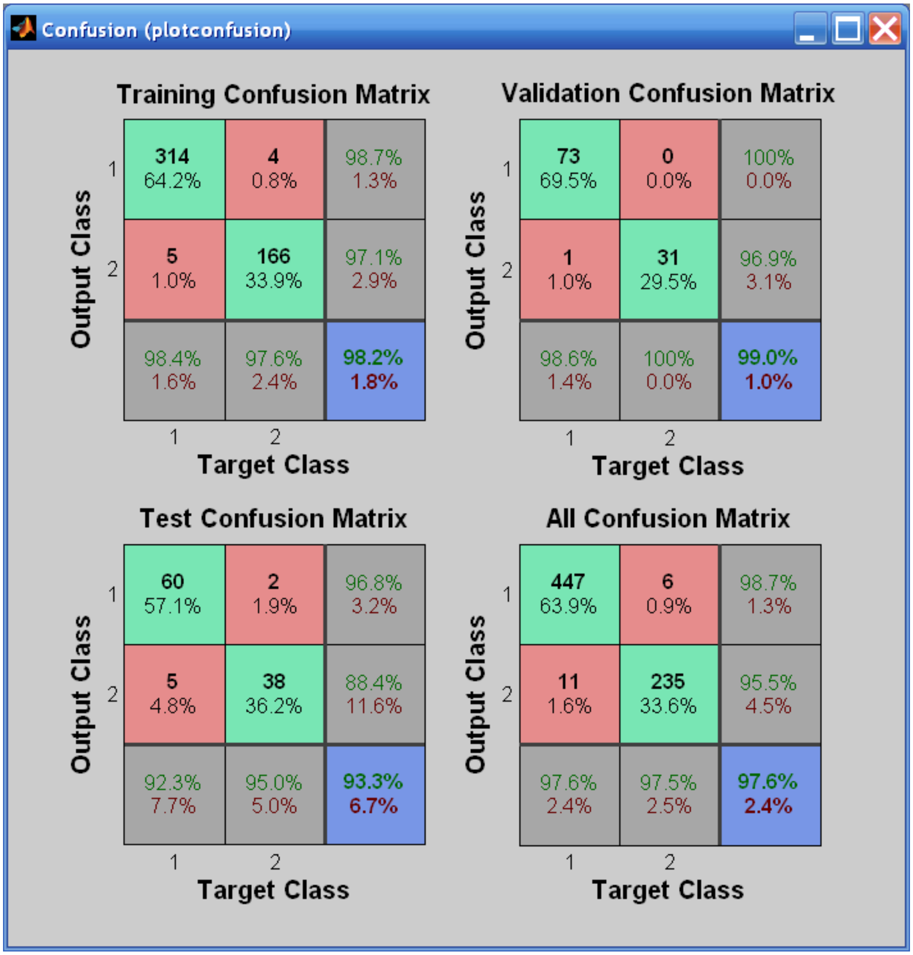
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)
Confusion (plotconfusion)
Receiver Operating Characteristic (plotroc)
Plot Interval:
1 epochs

✓ Validation stop

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.



Training Confusion Matrix

Output Class \ Target Class	1	2	
1	314 64.2%	4 0.8%	98.7% 1.3%
2	5 1.0%	166 33.9%	97.1% 2.9%
	98.4% 1.6%	97.6% 2.4%	98.2% 1.8%

Validation Confusion Matrix

Output Class \ Target Class	1	2	
1	73 69.5%	0 0.0%	100% 0.0%
2	1 1.0%	31 29.5%	96.9% 3.1%
	98.6% 1.4%	100% 0.0%	99.0% 1.0%

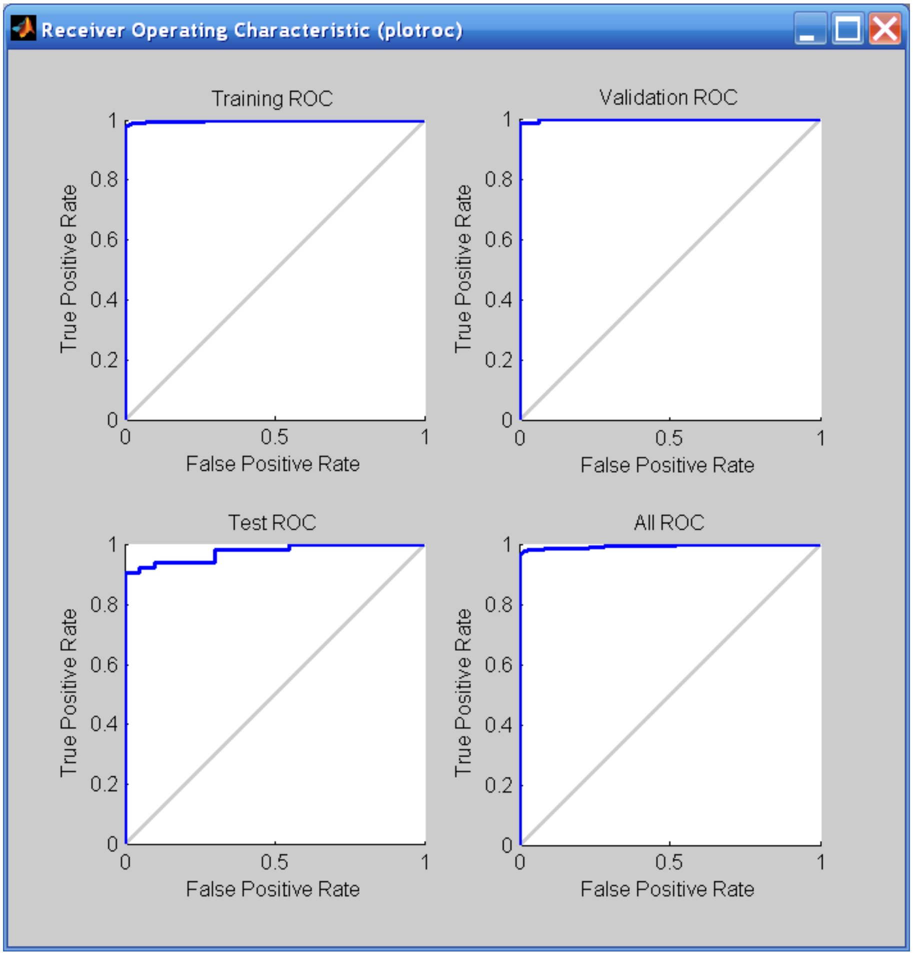
Test Confusion Matrix

Output Class \ Target Class	1	2	
1	60 57.1%	2 1.9%	96.8% 3.2%
2	5 4.8%	38 36.2%	88.4% 11.6%
	92.3% 7.7%	95.0% 5.0%	93.3% 6.7%

All Confusion Matrix

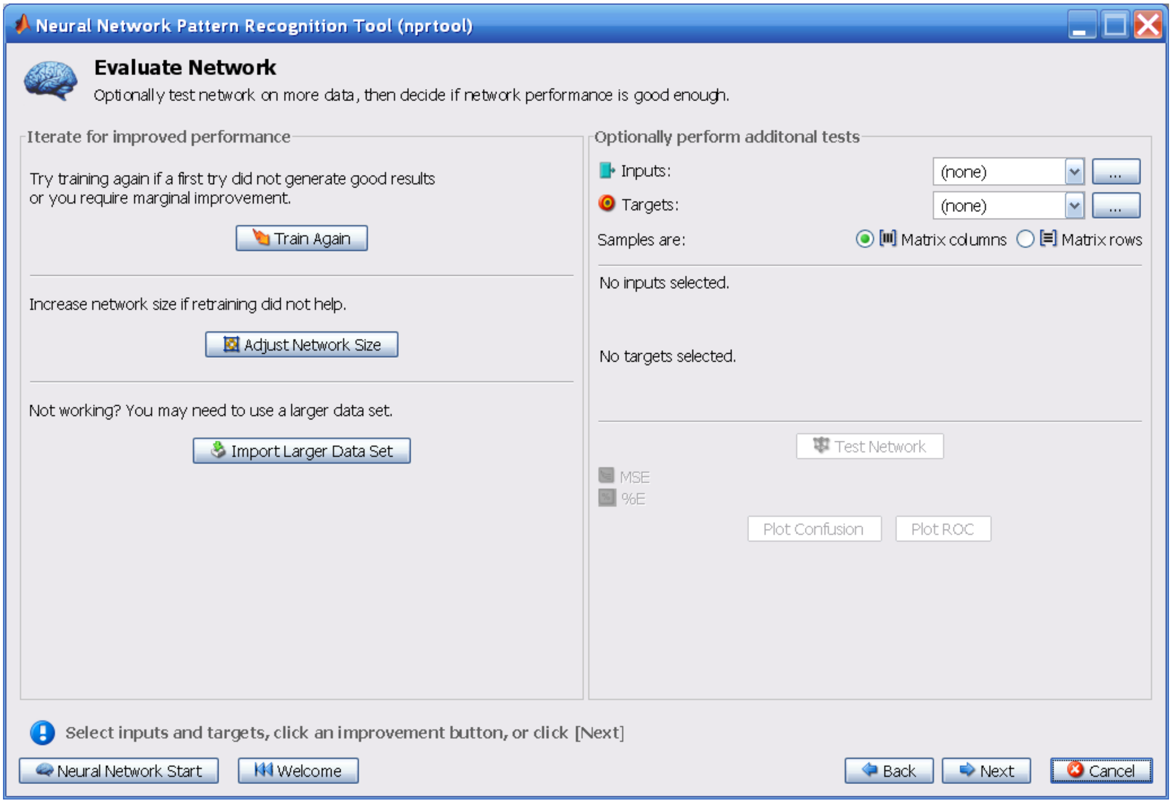
Output Class \ Target Class	1	2	
1	447 63.9%	6 0.9%	98.7% 1.3%
2	11 1.6%	235 33.6%	95.5% 4.5%
	97.6% 2.4%	97.5% 2.5%	97.6% 2.4%

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 - 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.



Neural Network Pattern Recognition Tool (nprtool)

Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

Try training again if a first try did not generate good results or you require marginal improvement.

Increase network size if retraining did not help.

Not working? You may need to use a larger data set.

Optionally perform additonal tests

Inputs: (none)

Targets: (none)

Samples are: ● Matrix columns ○ Matrix rows

No inputs selected.

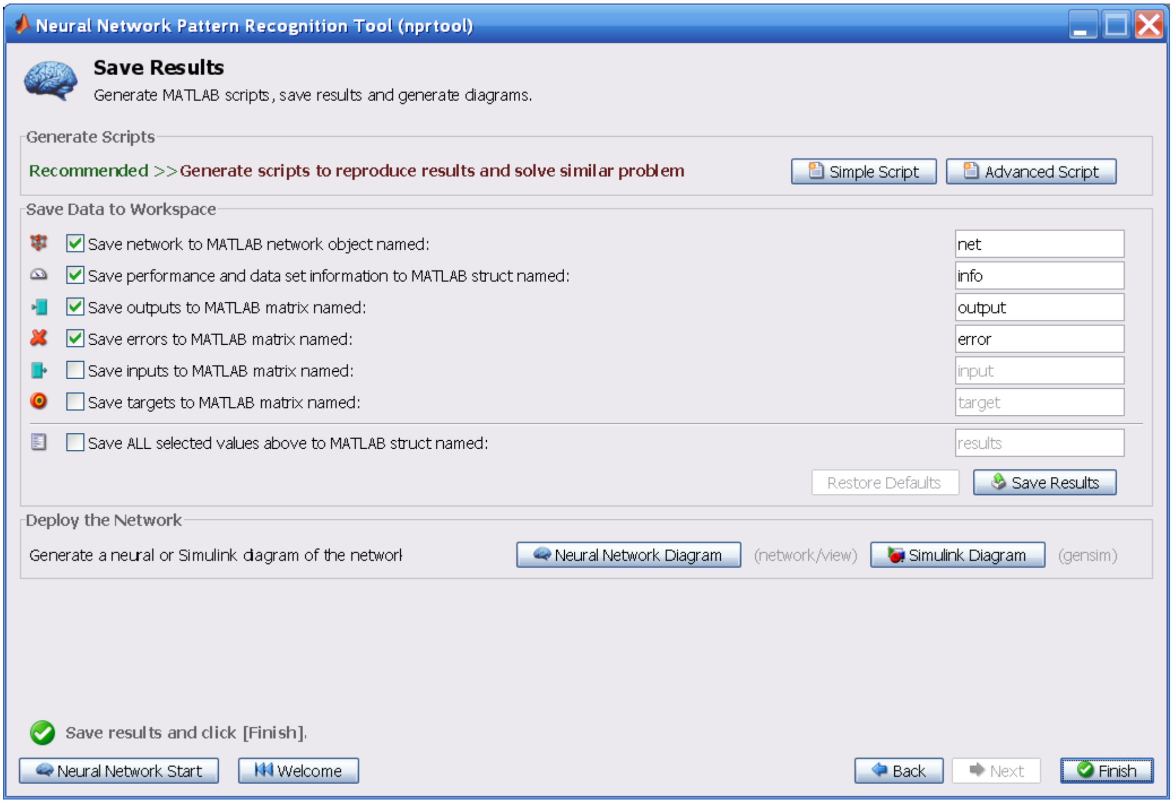
No targets selected.

Select inputs and targets, click an improvement button, or click [Next]

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**.



Neural Network Pattern Recognition Tool (nprtool)

Save Results

Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >>Generate scripts to reproduce results and solve similar problem

Save Data to Workspace

☑ Save network to MATLAB network object named: net

☑ Save performance and data set information to MATLAB struct named: info

☑ Save outputs to MATLAB matrix named: output

☑ Save errors to MATLAB matrix named: error

□ Save inputs to MATLAB matrix named: input

□ Save targets to MATLAB matrix named: target

□ Save ALL selected values above to MATLAB struct named: results

Deploy the Network

Generate a neural or Simulink diagram of the networl

✓ Save results and click [Finish].

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 1-41, 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. As an example, let's 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;
inputs = 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.

0 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 cancer_dataset
    inputs = cancerInputs;
    targets = cancerTargets;
    ```

1 Create the network. The default network for function fitting (or regression) problems, `patternnet`, is a feedforward network with the default tan-sigmoid transfer functions in both the hidden and output layers. 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.

2 Set up the division of data.

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

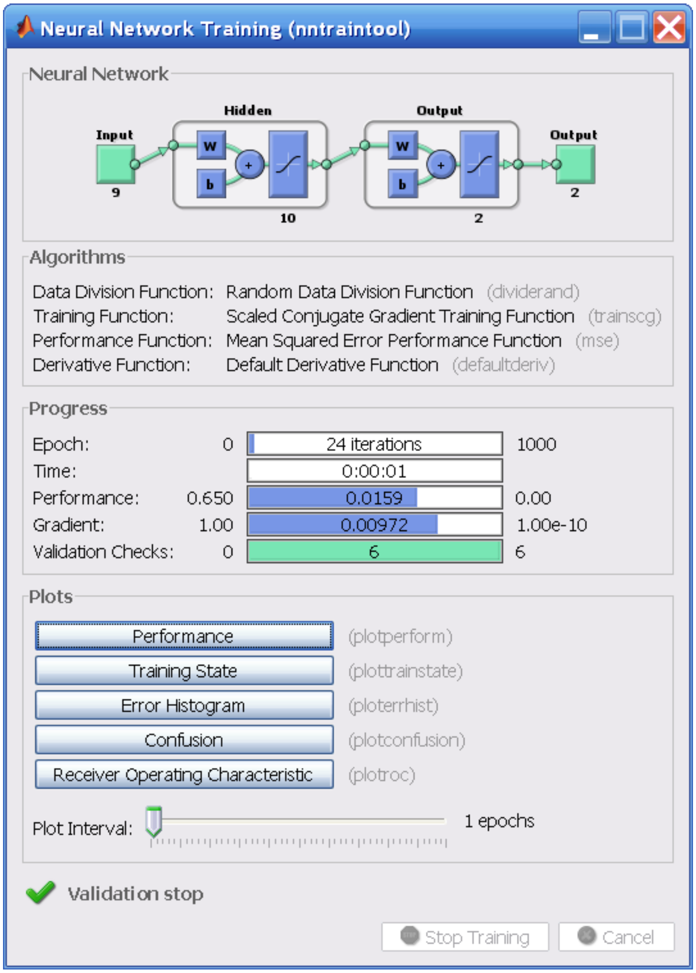
With these settings, the input vectors and target vectors will be randomly divided, with 70% used for training, 15% for validation and 15% for testing.

(See “Dividing the Data” on page 3-10 for more discussion of the data division process.)

3 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**.



Neural Network Training (nntraintool)

Neural Network

Algorithms

Data Division Function: Random Data Division Function (dividerand)
Training Function: Scaled Conjugate Gradient Training Function (trainscg)
Performance Function: Mean Squared Error Performance Function (mse)
Derivative Function: Default Derivative Function (defaultderiv)

Progress

Epoch: 0 24 iterations 1000
Time: 0:00:01
Performance: 0.650 0.0159 0.00
Gradient: 1.00 0.00972 1.00e-10
Validation Checks: 0 6 6

Plots

Plot Interval: 1 epochs

✓ Validation stop

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

4 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)
```

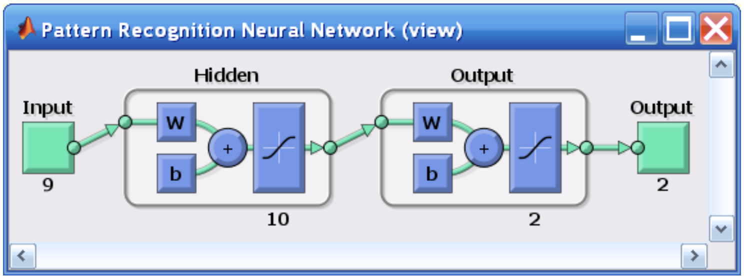
It is also possible to calculate the network performance only on the test set, by using the testing indices, which are located in the training record.

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

5 View the network diagram.

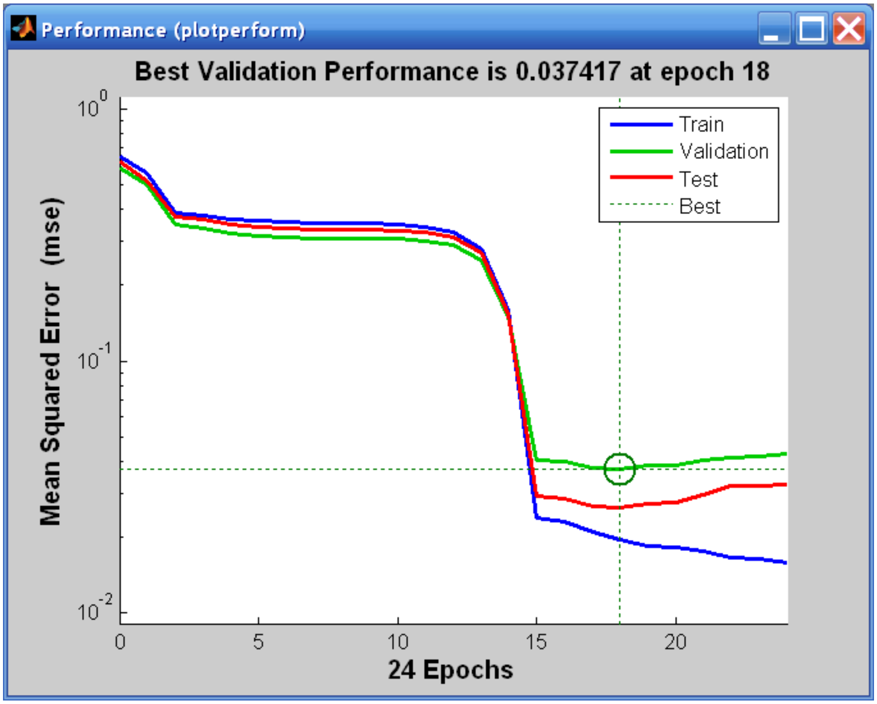
```
view(net)
```

This creates the following diagram of the network.



6 Plot the training, validation and test performance.

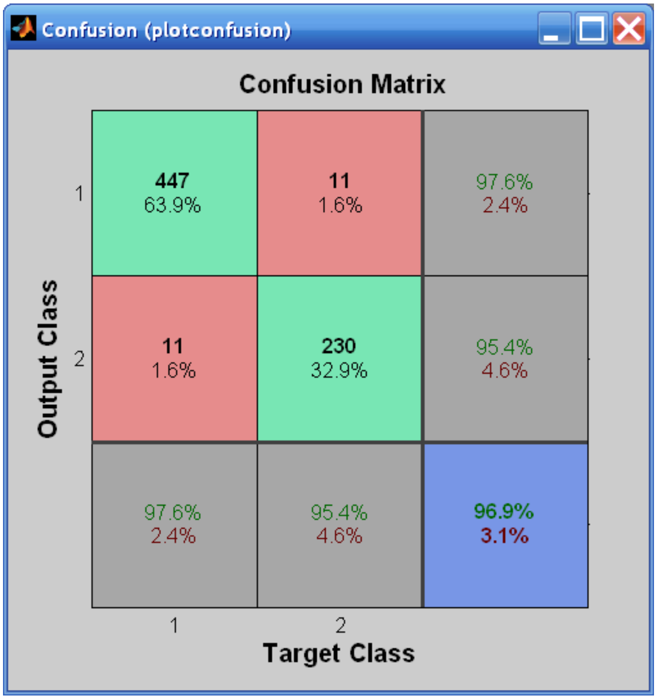
```
figure, plotperform(tr)
```



7 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 next figure shows the results.



	1	2	
1	**447** 63.9%	**11** 1.6%	97.6% 2.4%
2	**11** 1.6%	**230** 32.9%	95.4% 4.6%
	97.6% 2.4%	95.4% 4.6%	**96.9%** **3.1%**

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” on page 3-15).

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.

Clustering Data

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 [MuAh94]. 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 1-50.
- Use a command-line solution, as described in “Using Command-Line Functions” on page 1-60.

Defining a Problem

To define a clustering problem, simply arrange Q input vectors to be clustered as columns in an input matrix (see Chapter 2, “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:

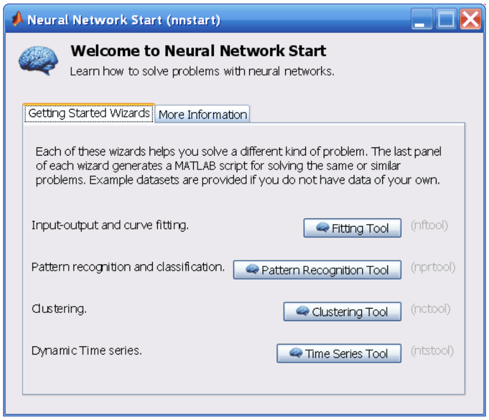
```
inputs = [7 0 6 2 6 5 6 1 0 1; 6 2 5 0 7 5 5 1 2 2]
```

The next section demonstrates 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
```



Neural Network Start (nnstart)

Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

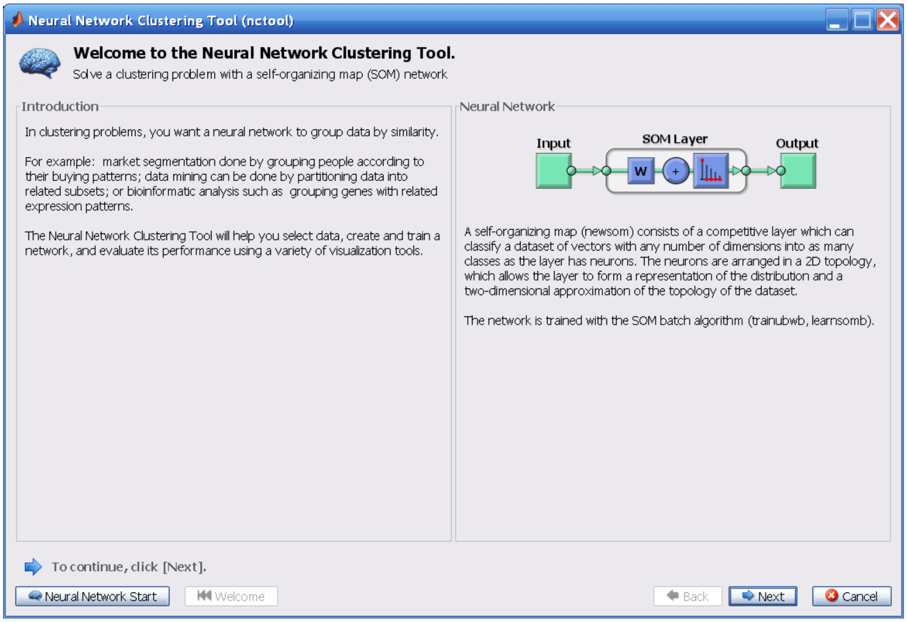
Input-output and curve fitting. Fitting Tool (nftool)

Pattern recognition and classification. Pattern Recognition Tool (nprtool)

Clustering. Clustering Tool (nctool)

Dynamic Time series. Time Series Tool (ntstool)

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



Neural Network Clustering Tool (nctool)

Welcome to the Neural Network Clustering Tool.

Solve a clustering problem with a self-organizing map (SOM) network

Introduction

In clustering problems, you want a neural network to group data by similarity.

For example: market segmentation done by grouping people according to their buying patterns; data mining can be done by partitioning data into related subsets; or bioinformatic analysis such as grouping genes with related expression patterns.

The Neural Network Clustering Tool will help you select data, create and train a network, and evaluate its performance using a variety of visualization tools.

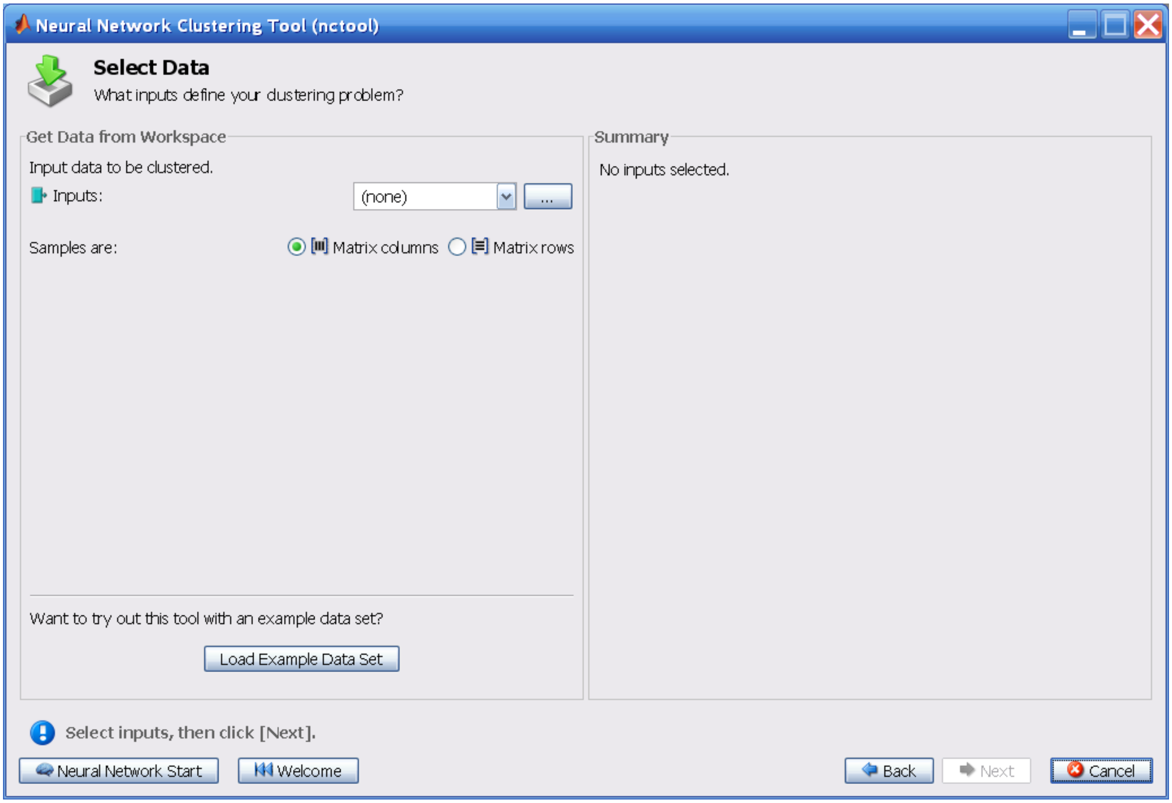
Neural Network

A self-organizing map (newsom) consists of a competitive layer which can classify a dataset of vectors with any number of dimensions into as many classes as the layer has neurons. The neurons are arranged in a 2D topology, which allows the layer to form a representation of the distribution and a two-dimensional approximation of the topology of the dataset.

The network is trained with the SOM batch algorithm (trainbwb, learnsomb).

To continue, click [Next].

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



Neural Network Clustering Tool (nctool)

Select Data

What inputs define your clustering problem?

Get Data from Workspace

Input data to be clustered.

Inputs: (none)

Samples are: ● Matrix columns ○ Matrix rows

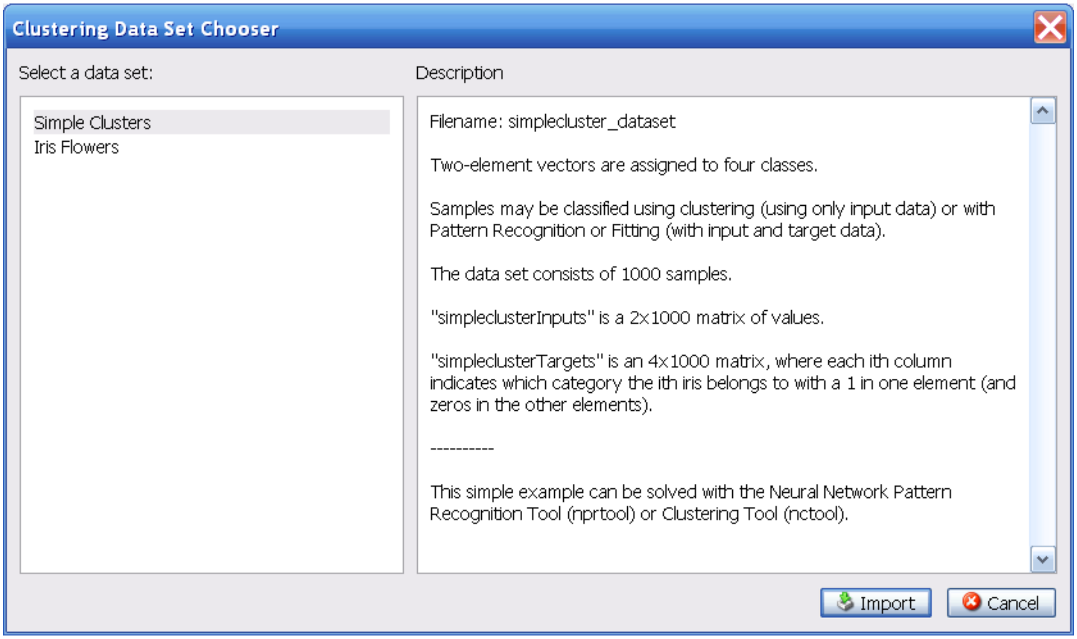
Want to try out this tool with an example data set?

Summary

No inputs selected.

Select inputs, then click [Next].

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



Clustering Data Set Chooser

Select a data set:

Simple Clusters
Iris Flowers

Description

Filename: simplecluster_dataset

Two-element vectors are assigned to four classes.

Samples may be classified using clustering (using only input data) or with Pattern Recognition or Fitting (with input and target data).

The data set consists of 1000 samples.

"simpleclusterInputs" is a 2x1000 matrix of values.

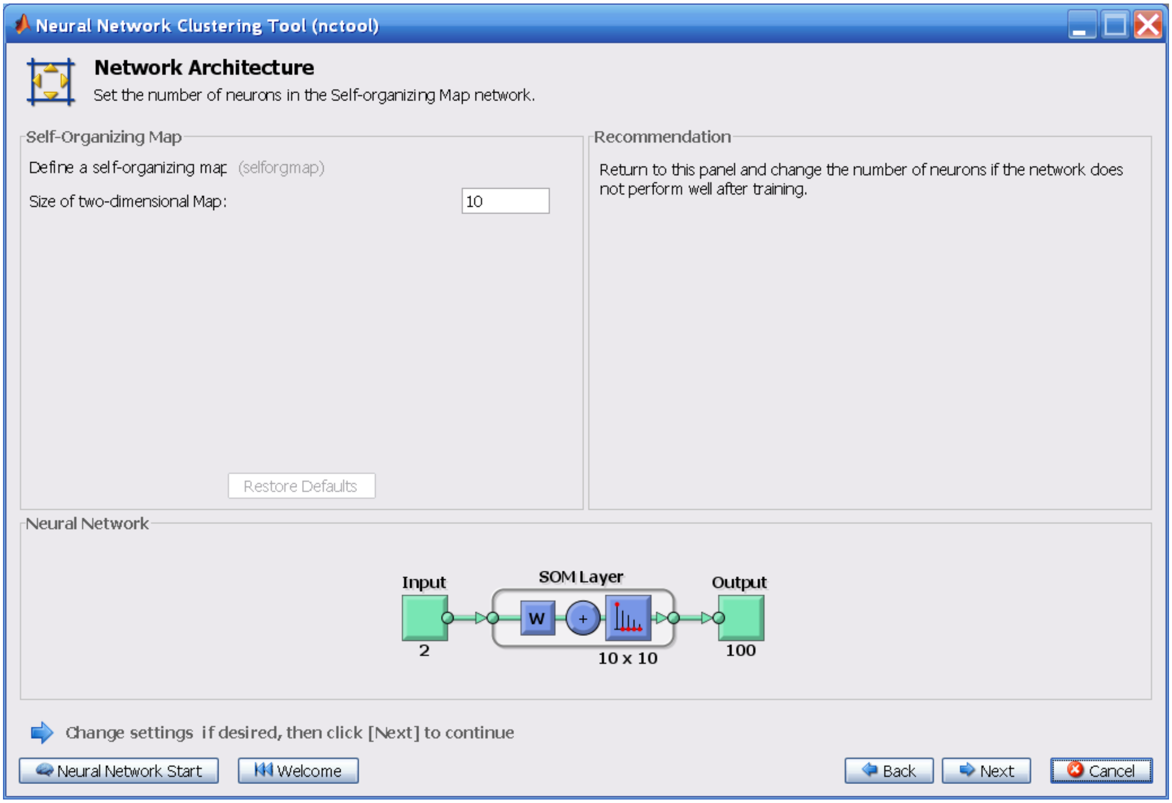
"simpleclusterTargets" is an 4x1000 matrix, where each ith column indicates which category the ith iris belongs to with a 1 in one element (and zeros in the other elements).

This simple example can be solved with the Neural Network Pattern Recognition Tool (nprtool) or Clustering Tool (nctool).

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” on page 7-9 and Chapter 14 of [HDB96].) 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.



Neural Network Clustering Tool (nctool)

Network Architecture

Set the number of neurons in the Self-organizing Map network.

Self-Organizing Map

Define a self-organizing map (selforgmap)

Size of two-dimensional Map: 10

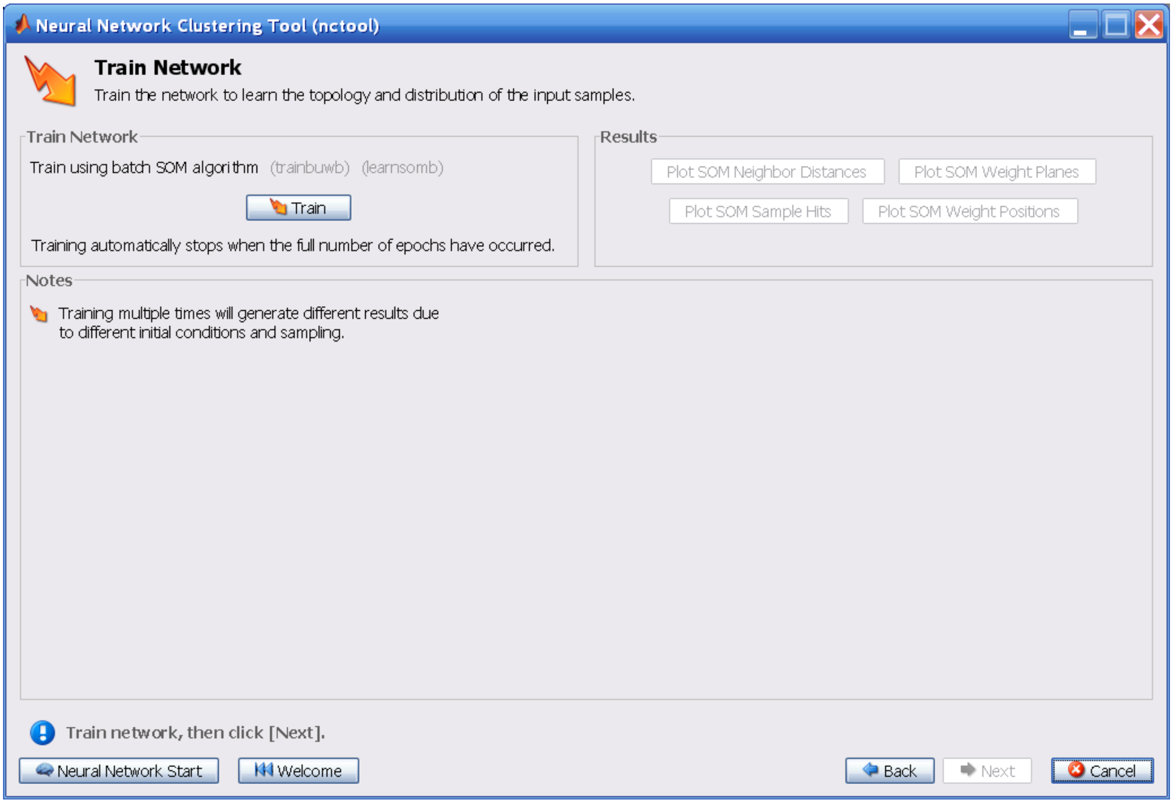
Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

Neural Network

Change settings if desired, then click [Next] to continue

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



Neural Network Clustering Tool (nctool)

Train Network

Train the network to learn the topology and distribution of the input samples.

Train using batch SOM algorithm (trainbuwb) (learnsomb)

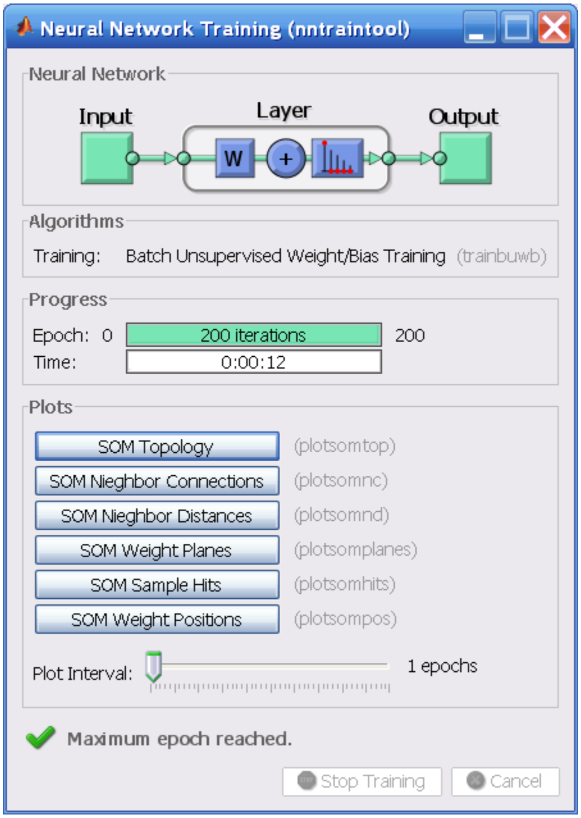
Training automatically stops when the full number of epochs have occurred.

Results
Plot SOM Neighbor Distances
Plot SOM Weight Planes
Plot SOM Sample Hits
Plot SOM Weight Positions
Notes

Training multiple times will generate different results due to different initial conditions and sampling.

Train network, then click [Next].

8 Click **Train**



Neural Network Training (nntraintool)

Neural Network

Algorithms

Training: Batch Unsupervised Weight/Bias Training (trainbuwb)

Progress

Epoch: 0 200 iterations 200

Time: 0:00:12

Plots

(plotsomtop)

(plotsomnc)

(plotsomnd)

(plotsomplanes)

(plotsomhits)

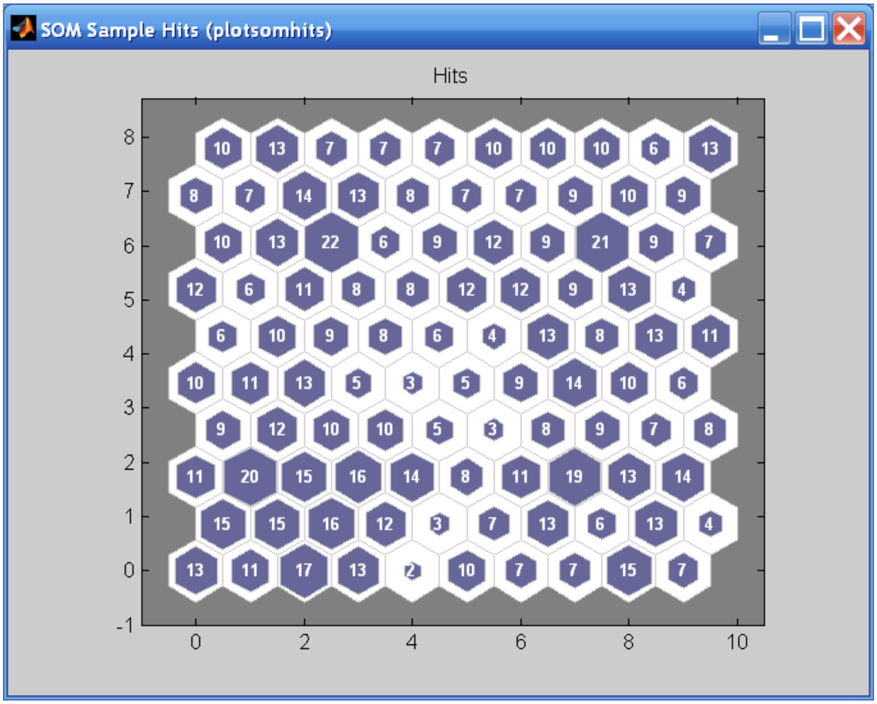
(plotsompos)

Plot Interval: 1 epochs

✓ Maximum epoch reached.

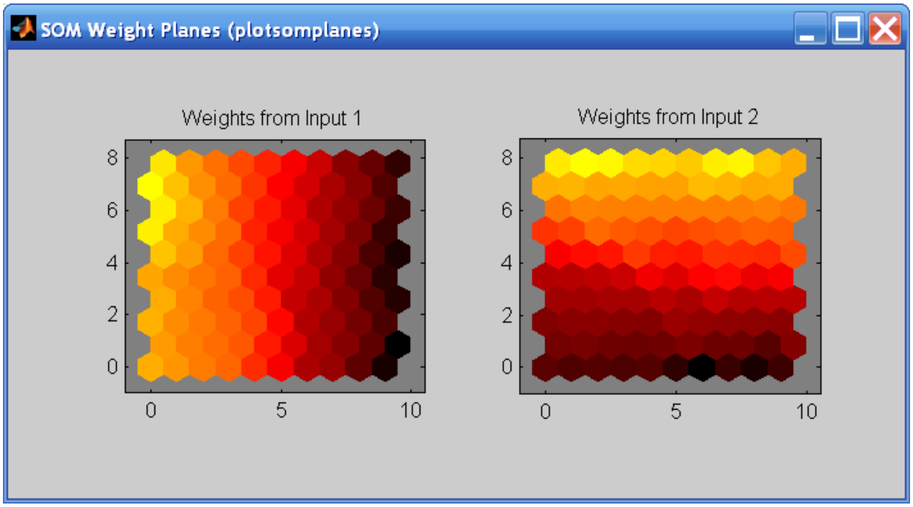
The training runs for the maximum number of epochs, which is 200.

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**.



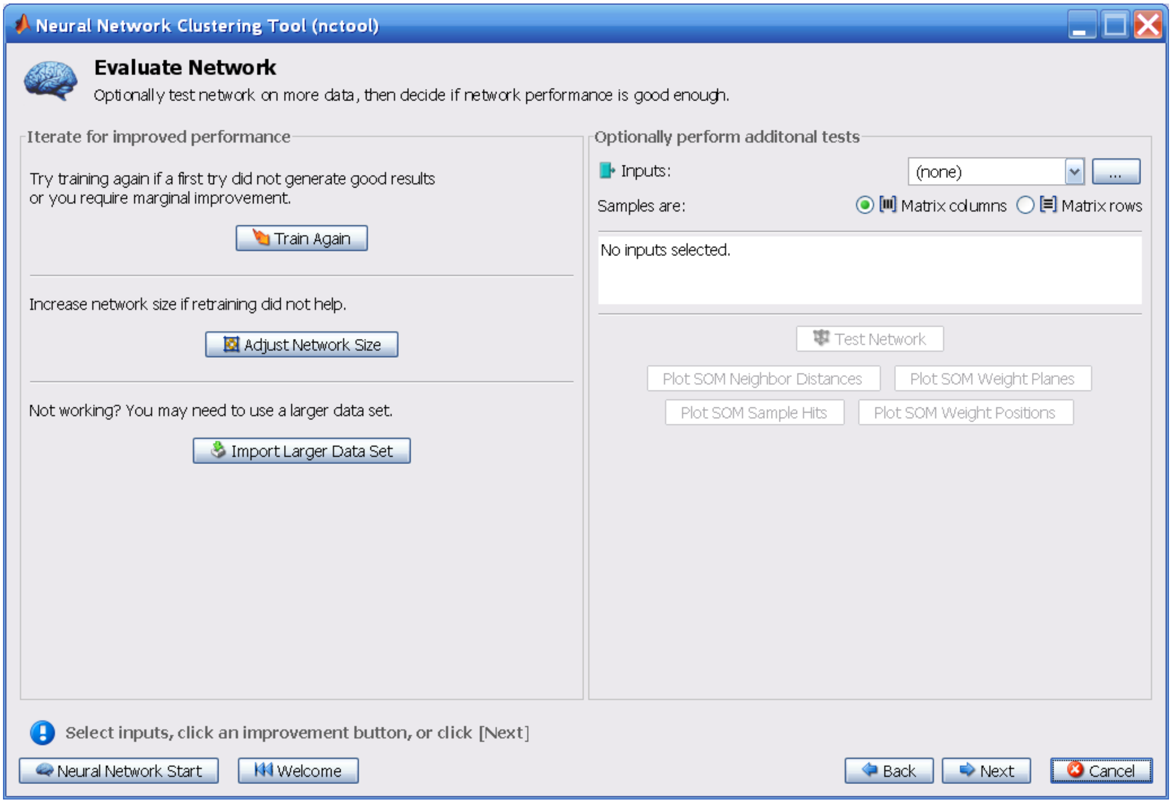
The default topology of the SOM is hexagonal. This figure shows the neuron locations in the topology, and indicates how many of the training data are associated with each of the neurons (cluster centers). The topology is a 10-by-10 grid, so there are 100 neurons. The maximum number of hits associated 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.



Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Try training again if a first try did not generate good results or you require marginal improvement.

Increase network size if retraining did not help.

Not working? You may need to use a larger data set.

No inputs selected.

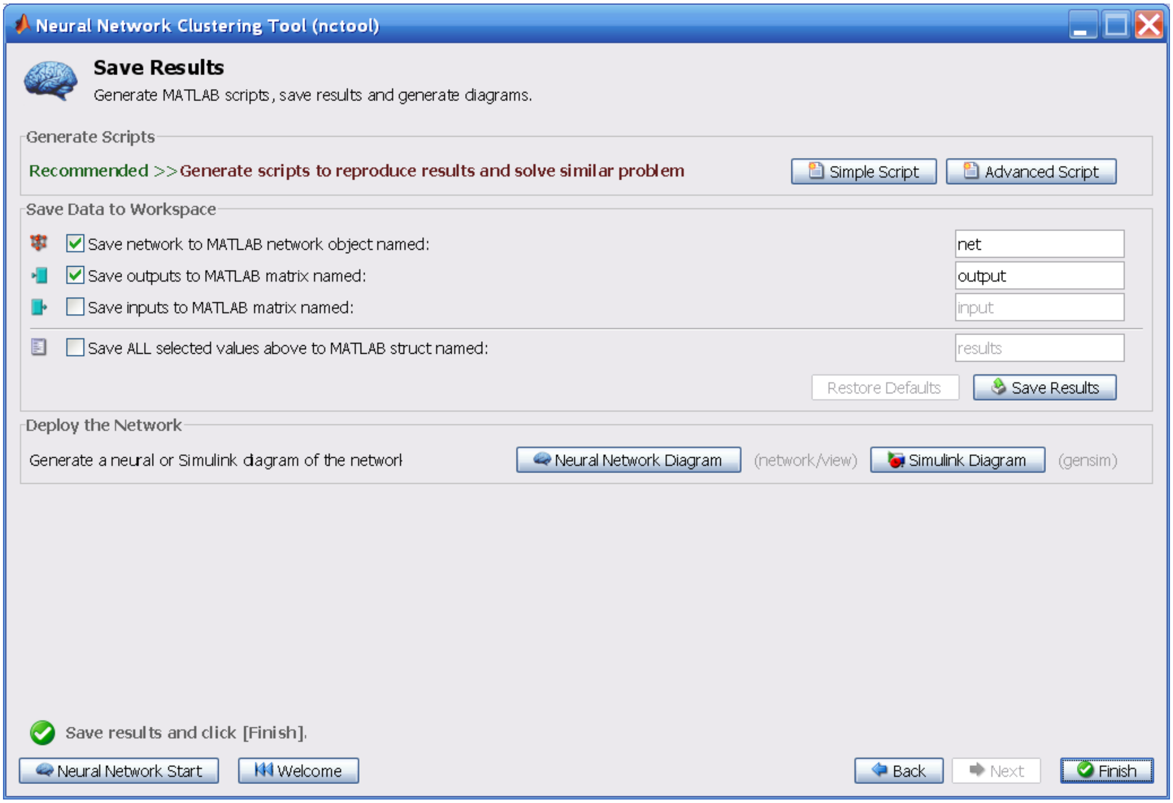
Select inputs, click an improvement button, or click [Next]

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 the buttons on this screen to save your results.



Neural Network Clustering Tool (nctool)

Save Results

Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >> Generate scripts to reproduce results and solve similar problem

Save Data to Workspace

☑ Save network to MATLAB network object named: net

☑ Save outputs to MATLAB matrix named: output

□ Save inputs to MATLAB matrix named: input

□ Save ALL selected values above to MATLAB struct named: results

Deploy the Network

Generate a neural or Simulink diagram of the networl

Save results and click [Finish].

- 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 1-60, 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.

14 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.

0 The script assumes that the input vectors are already loaded into the workspace. To demonstrate 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;
```

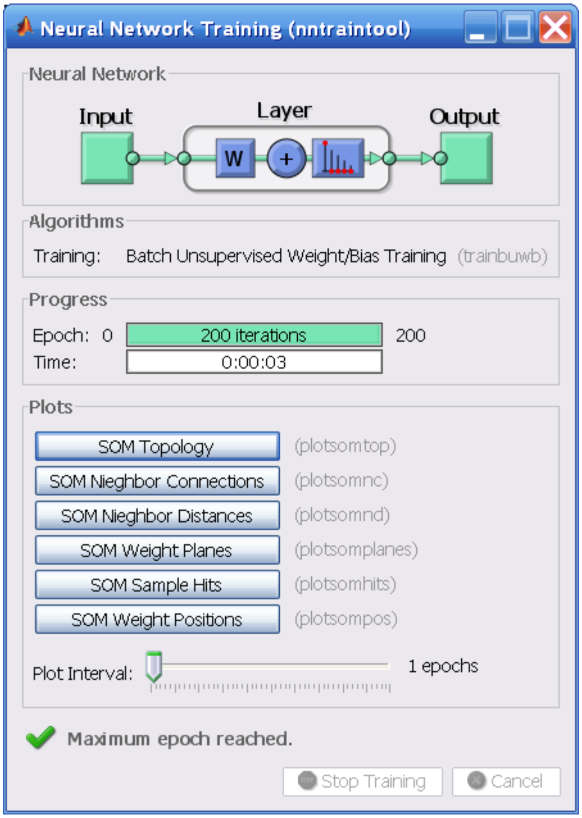
1 Create a network. For this example, you use a self-organizing map (SOM). This network has one layer, with the neurons organized in a grid. (For more information, see "Self-Organizing Feature Maps" on page 7-9 and Chapter 14 of [HDB96].) When creating the network with `selforgmap`, you specify the number of rows and columns in the grid:

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

2 Train the network. The SOM network uses the default batch SOM algorithm for training.

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

3 During training, the training window opens and displays the training progress. To interrupt training at any point, click **Stop Training**.



Training: Batch Unsupervised Weight/Bias Training (trainbuwb)

Epoch: 0 200 iterations 200

Time: 0:00:03

Plot Interval: 1 epochs

✓ Maximum epoch reached.

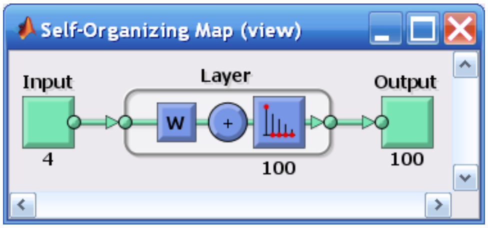
4 Test the network. After the network has been trained, you can use it to compute the network outputs.

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

5 View the network diagram.

```
view(net)
```

This creates the following diagram of the network.

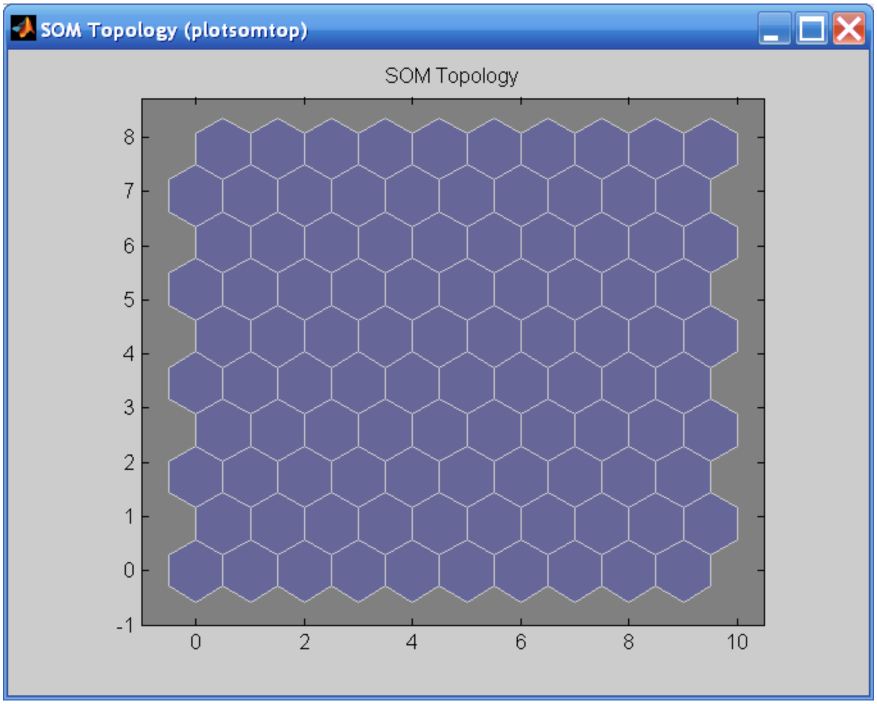


6 For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional 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)
```

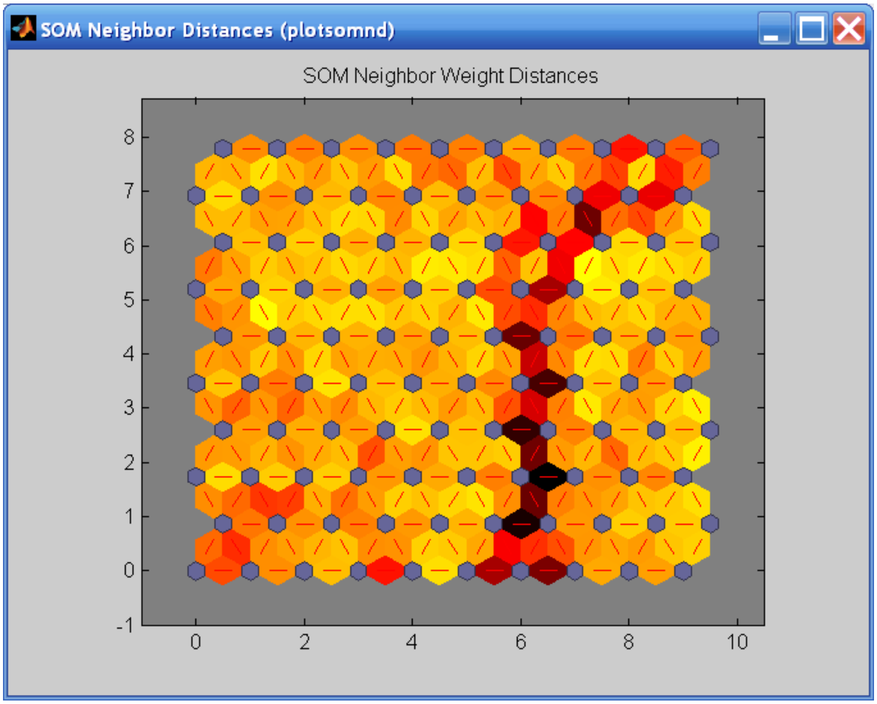
In this figure, each of the hexagons represents a neuron. The grid is 10-by-10, so there are a total of 100 neurons in this network. There are four elements in each input vector, so the input space is four-dimensional. The weight vectors (cluster centers) fall within this space.

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*).



7 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.


SOM Neighbor Distances (plotsomnd)
SOM Neighbor Weight Distances
8
7
6
5
4
3
2
1
0
-1
0
2
4
6
8
10

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.

Time Series Prediction

Dynamic neural networks, such as those described in Chapter 4, “Dynamic Networks,” are good at time series prediction.

Suppose, for instance, that you have data from a pH neutralization process [McHs72]. 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 1-67.
- Use command-line functions, as described in “Using Command-Line Functions” on page 1-80.

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” on page 2-23 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:

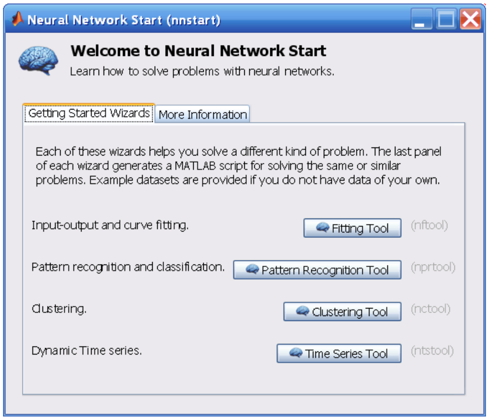
```
targets = {1 2 3 4 5};
```

The next section demonstrates 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
```



Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Each of these wizards helps you solve a different kind of problem. The last panel of each wizard generates a MATLAB script for solving the same or similar problems. Example datasets are provided if you do not have data of your own.

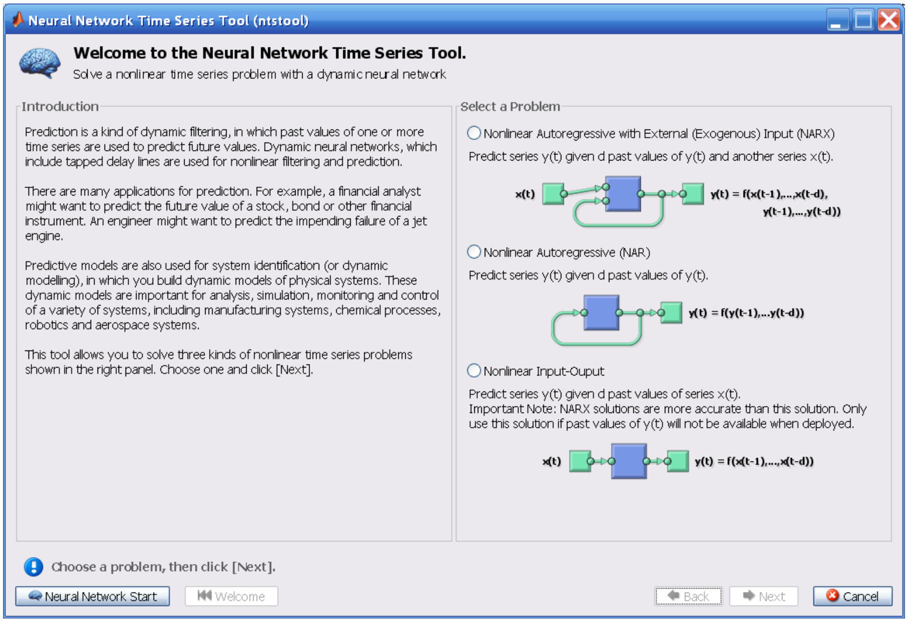
Input-output and curve fitting.

Pattern recognition and classification.

Clustering.

Dynamic Time series.

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


Neural Network Time Series Tool (ntstool)
Introduction
Select a Problem
x(t)
y(t) = f(x(t-1),...,x(t-d),
y(t-1),...,y(t-d))
y(t) = f(y(t-1),...y(t-d))
x(t)
y(t) = f(x(t-1),...,x(t-d))
Neural Network Start
Welcome
Back
Next
Cancel

Welcome to the Neural Network Time Series Tool.

Solve a nonlinear time series problem with a dynamic neural network

Prediction is a kind of dynamic filtering, in which past values of one or more time series are used to predict future values. Dynamic neural networks, which include tapped delay lines are used for nonlinear filtering and prediction.

There are many applications for prediction. For example, a financial analyst might want to predict the future value of a stock, bond or other financial instrument. An engineer might want to predict the impending failure of a jet engine.

Predictive models are also used for system identification (or dynamic modelling), in which you build dynamic models of physical systems. These dynamic models are important for analysis, simulation, monitoring and control of a variety of systems, including manufacturing systems, chemical processes, robotics and aerospace systems.

This tool allows you to solve three kinds of nonlinear time series problems shown in the right panel. Choose one and click [Next].

○ Nonlinear Autoregressive with External (Exogenous) Input (NARX)

Predict series y(t) given d past values of y(t) and another series x(t).

○ Nonlinear Autoregressive (NAR)

Predict series y(t) given d past values of y(t).

○ Nonlinear Input-Ouput

Predict series y(t) given d past values of series x(t).
Important Note: NARX solutions are more accurate than this solution. Only use this solution if past values of y(t) will not be available when deployed.

Choose a problem, then click [Next].

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)" on page 4-21), and can be written as follows:

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

  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:

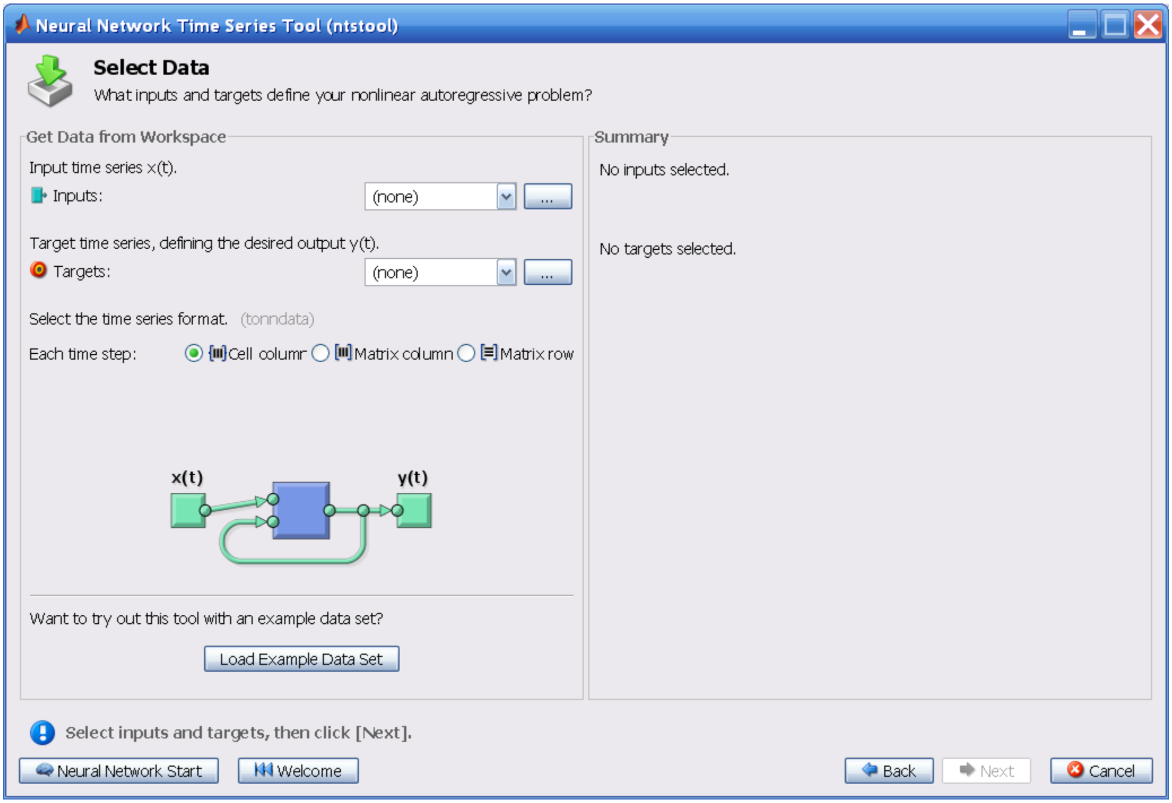
  $$y(t) = f(y(t-1), \ldots, y(t-d))$$

  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:

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

  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 demonstration, select the NARX model and click **Next** to proceed.



Neural Network Time Series Tool (ntstool)

Select Data

What inputs and targets define your nonlinear autoregressive problem?

Get Data from Workspace

Input time series x(t).

Inputs: (none)

Target time series, defining the desired output y(t).

Targets: (none)

Select the time series format. (tonndata)

Each time step: ● Cell columr ○ Matrix column ○ Matrix row

Summary

No inputs selected.

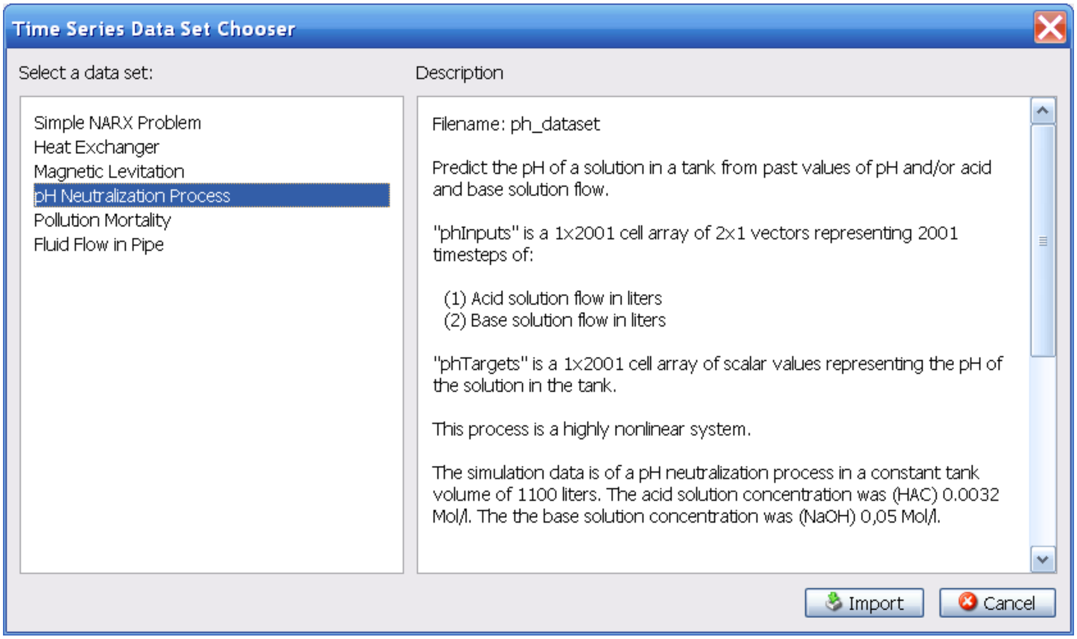
No targets selected.

Want to try out this tool with an example data set?

Select inputs and targets, then click [Next].

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.



Filename: ph_dataset

Predict the pH of a solution in a tank from past values of pH and/or acid and base solution flow.

"phInputs" is a 1x2001 cell array of 2x1 vectors representing 2001 timesteps of:

(1) Acid solution flow in liters
(2) Base solution flow in liters

"phTargets" is a 1x2001 cell array of scalar values representing the pH of the solution in the tank.

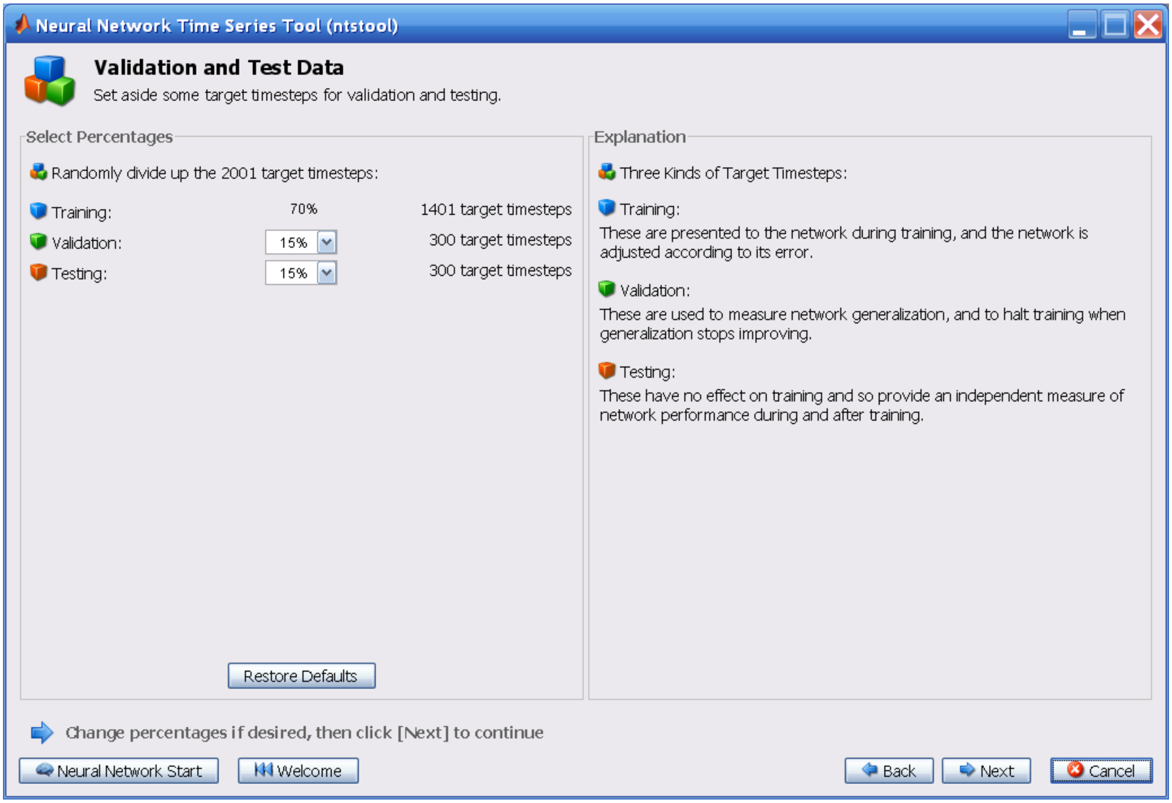
This process is a highly nonlinear system.

The simulation data is of a pH neutralization process in a constant tank volume of 1100 liters. The acid solution concentration was (HAC) 0.0032 Mol/l. The the base solution concentration was (NaOH) 0,05 Mol/l.

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.



Neural Network Time Series Tool (ntstool)

Validation and Test Data

Set aside some target timesteps for validation and testing.

Select Percentages

Randomly divide up the 2001 target timesteps:

Training:	70%	1401 target timesteps
Validation:	15%	300 target timesteps
Testing:	15%	300 target timesteps

Explanation

Three Kinds of Target Timesteps:

Training:
These are presented to the network during training, and the network is adjusted according to its error.

Validation:
These are used to measure network generalization, and to halt training when generalization stops improving.

Testing:
These have no effect on training and so provide an independent measure of network performance during and after training.

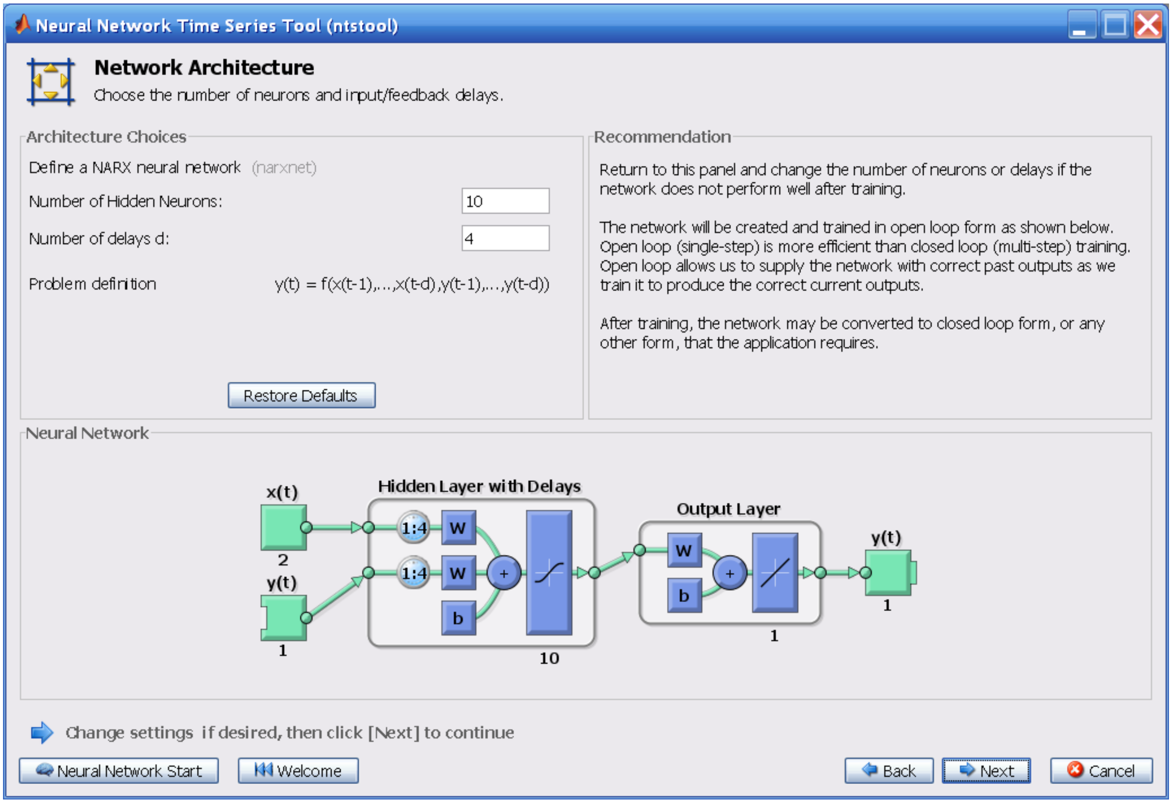
Change percentages if desired, then click [Next] to continue

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” on page 3-10 for more discussion of the data division process.)

7 Click **Next**.



Neural Network Time Series Tool (ntstool)

Network Architecture

Choose the number of neurons and input/feedback delays.

Architecture Choices

Define a NARX neural network (narxnet)

Number of Hidden Neurons: 10

Number of delays d: 4

Problem definition y(t) = f(x(t-1),...,x(t-d),y(t-1),...,y(t-d))

Restore Defaults

Recommendation

Return to this panel and change the number of neurons or delays if the network does not perform well after training.

The network will be created and trained in open loop form as shown below. Open loop (single-step) is more efficient than closed loop (multi-step) training. Open loop allows us to supply the network with correct past outputs as we train it to produce the correct current outputs.

After training, the network may be converted to closed loop form, or any other form, that the application requires.

Neural Network

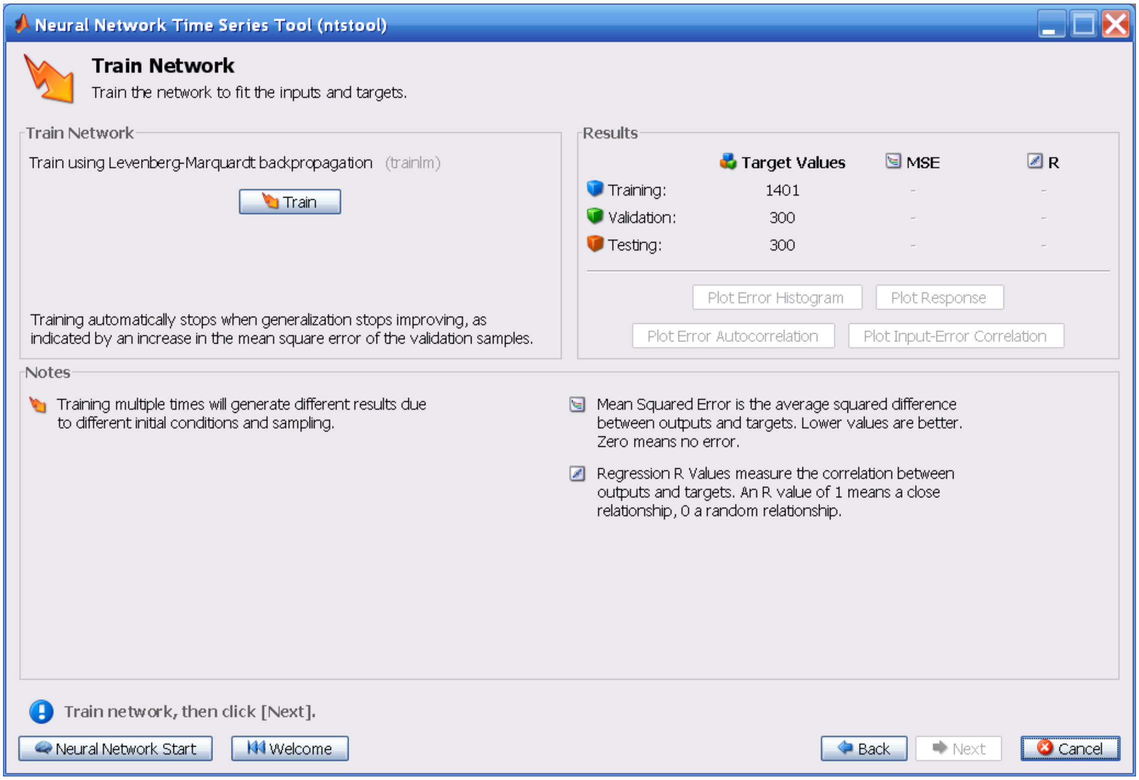
Change settings if desired, then click [Next] to continue

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 $y(t\text{-}1)$, $y(t\text{-}2)$, ..., $y(t\text{-}\mathrm{d})$. 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)” in Chapter 4.

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**.



Neural Network Time Series Tool (ntstool)

Train Network

Train the network to fit the inputs and targets.

Train Network

Train using Levenberg-Marquardt backpropagation (trainlm)

Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Results

	Target Values	MSE	R
Training:	1401	-	-
Validation:	300	-	-
Testing:	300	-	-

Notes

Training multiple times will generate different results due to different initial conditions and sampling.

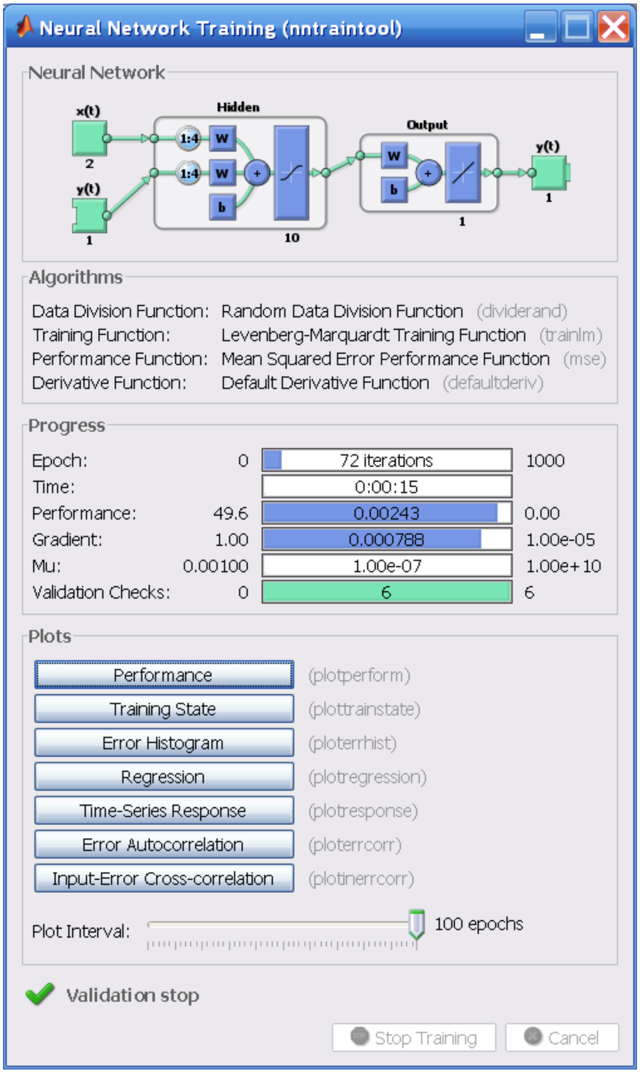
Mean Squared Error is the average squared difference between outputs and targets. Lower values are better. Zero means no error.

Regression R Values measure the correlation between outputs and targets. An R value of 1 means a close relationship, 0 a random relationship.

Train network, then click [Next].

9 Click **Train.**

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



Neural Network Training (nntraintool)

Neural Network

Algorithms

Data Division Function: Random Data Division Function (dividerand)
Training Function: Levenberg-Marquardt Training Function (trainlm)
Performance Function: Mean Squared Error Performance Function (mse)
Derivative Function: Default Derivative Function (defaultderiv)

Progress

Epoch:	0	72 iterations	1000
Time:		0:00:15	
Performance:	49.6	0.00243	0.00
Gradient:	1.00	0.000788	1.00e-05
Mu:	0.00100	1.00e-07	1.00e+10
Validation Checks:	0	6	6

Plots

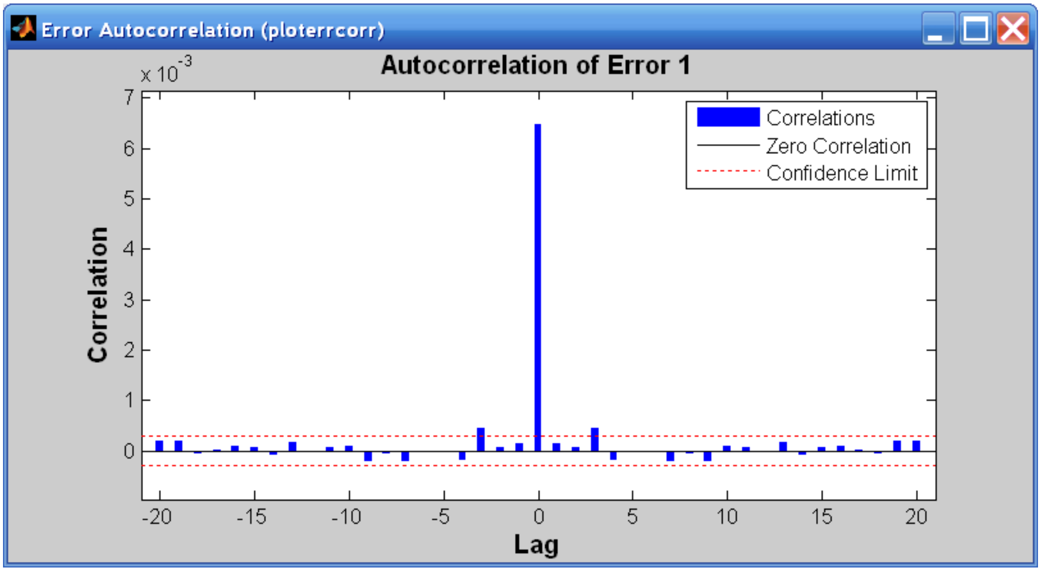
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)
Time-Series Response (plotresponse)
Error Autocorrelation (ploterrcorr)
Input-Error Cross-correlation (plotinerrcorr)

Plot Interval: 100 epochs

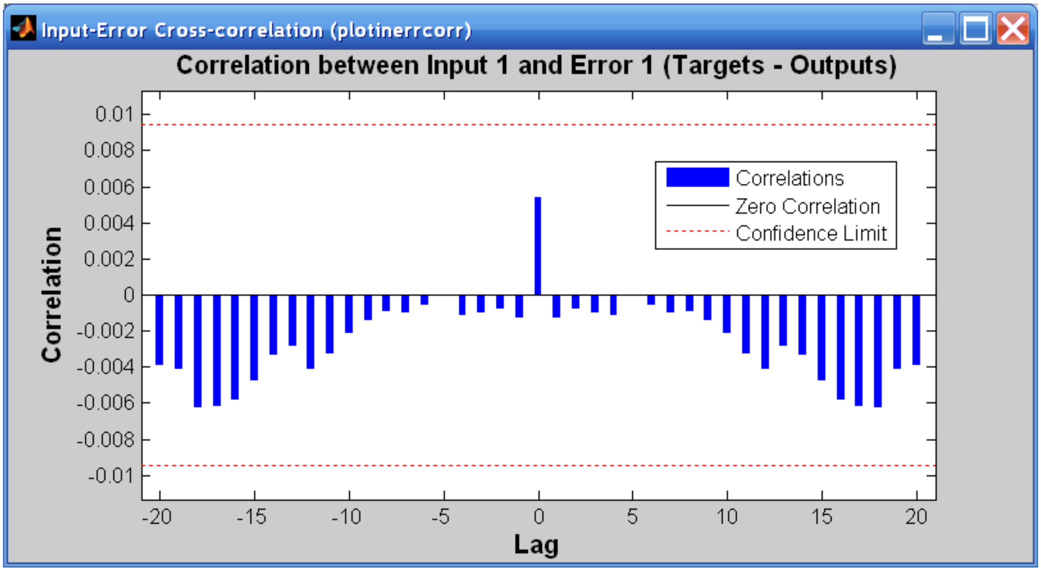
✓ Validation stop

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 `ntstool`. This will change the initial weights and biases of the network, and may produce an improved network after retraining.

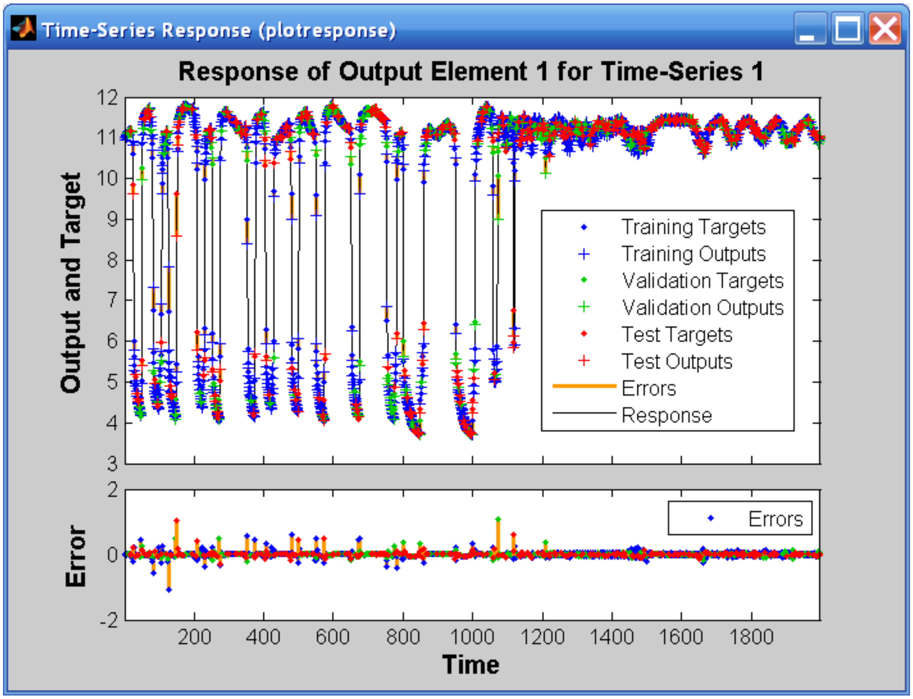


11 View the input-error cross-correlation function to obtain additional verification of network performance. Under the **Plots** pane, click **Input-Error Cross-correlation**.

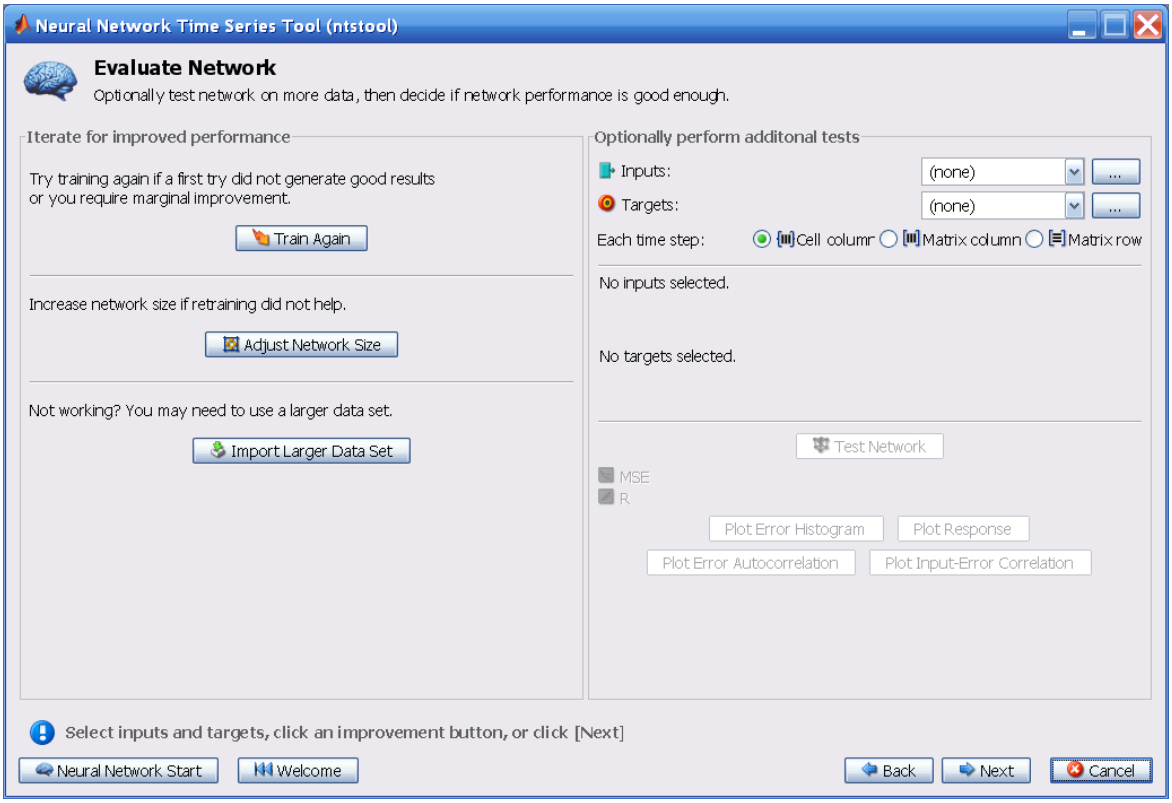


This input-error cross-correlation function illustrates how the errors are correlated with the input sequence $x(t)$. For a perfect prediction model, all of the correlations should be zero. If the input is correlated with the error, then it should be possible to improve the prediction, perhaps by increasing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



Neural Network Time Series Tool (ntstool)

Evaluate Network

Optionally test network on more data, then decide if network performance is good enough.

Iterate for improved performance

Try training again if a first try did not generate good results or you require marginal improvement.

Increase network size if retraining did not help.

Not working? You may need to use a larger data set.

Import Larger Data Set
Optionally perform additonal tests

Inputs: (none)

Targets: (none)

Each time step: ● Cell columr ○ Matrix column ○ Matrix row

No inputs selected.

No targets selected.

Select inputs and targets, click an improvement button, or click [Next]

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

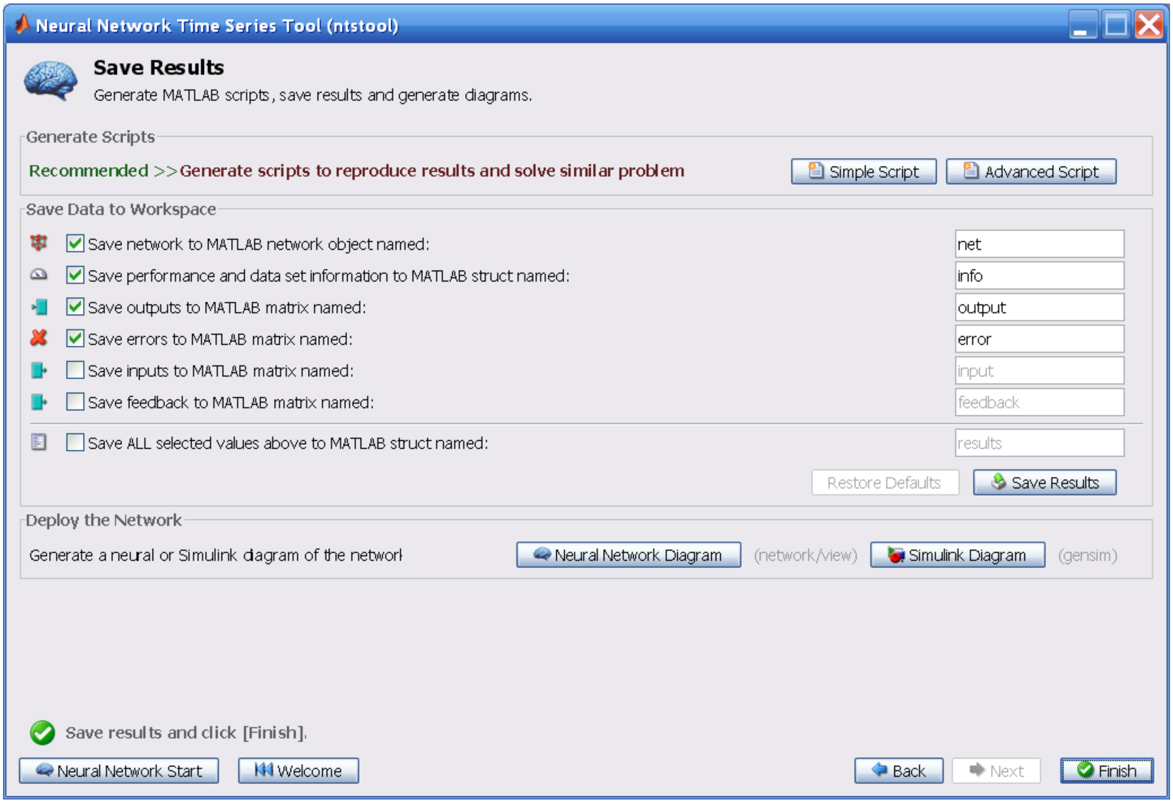
If you are dissatisfied with the network's performance on the original or new data, you can do any of the following:

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



Neural Network Time Series Tool (ntstool)

Save Results

Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

Recommended >> Generate scripts to reproduce results and solve similar problem

Save Data to Workspace

☑ Save network to MATLAB network object named: net

☑ Save performance and data set information to MATLAB struct named: info

☑ Save outputs to MATLAB matrix named: output

☑ Save errors to MATLAB matrix named: error

☐ Save inputs to MATLAB matrix named: input

☐ Save feedback to MATLAB matrix named: feedback

☐ Save ALL selected values above to MATLAB struct named: results

Deploy the Network

Generate a neural or Simulink diagram of the networl

Save results and click [Finish].

- 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 1-80, 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.

16 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] = ...
    preparets(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 outout 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.

0 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
X = pHInputs;
T = pHTargets;
```

1 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.

2 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 - `preparets`. 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] = ...
    preparets(net,inputSeries,{},targetSeries);
```

3 Set up the division of data.

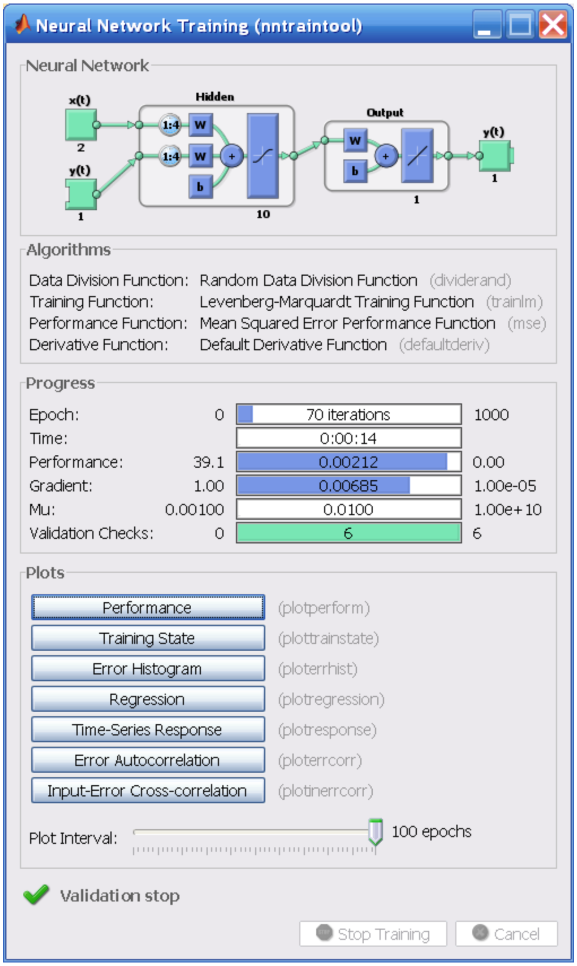
```
      net.divideParam.trainRatio = 70/100;
      net.divideParam.valRatio = 15/100;
      net.divideParam.testRatio = 15/100;
```

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.

4 Train the network. The network uses the default Levenberg-Marquardt algorithm (`trainlm`) for training. 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**.



Algorithms

Data Division Function: Random Data Division Function (dividerand)
Training Function: Levenberg-Marquardt Training Function (trainlm)
Performance Function: Mean Squared Error Performance Function (mse)
Derivative Function: Default Derivative Function (defaultderiv)

Progress

Epoch:	0	70 iterations	1000
Time:		0:00:14	
Performance:	39.1	0.00212	0.00
Gradient:	1.00	0.00685	1.00e-05
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

Plots

(plotperform)
(plottrainstate)
(ploterrhist)
(plotregression)
(plotresponse)
(ploterrcorr)
(plotinerrcorr)

Plot Interval: 100 epochs

✓ Validation stop

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

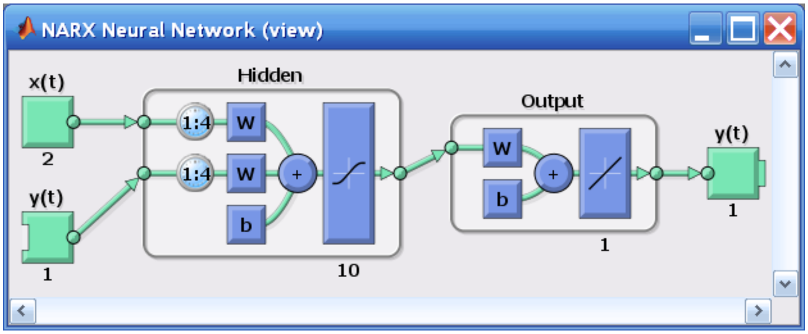
5 Test the network. After the network has been trained, you can use it to compute the network outputs. The following code calculates the network outputs, errors and overall performance. Note that to simulate a network with tapped delay lines, you need to assign the initial values for these delayed signals.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

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

6 View the network diagram.

```
view(net)
```

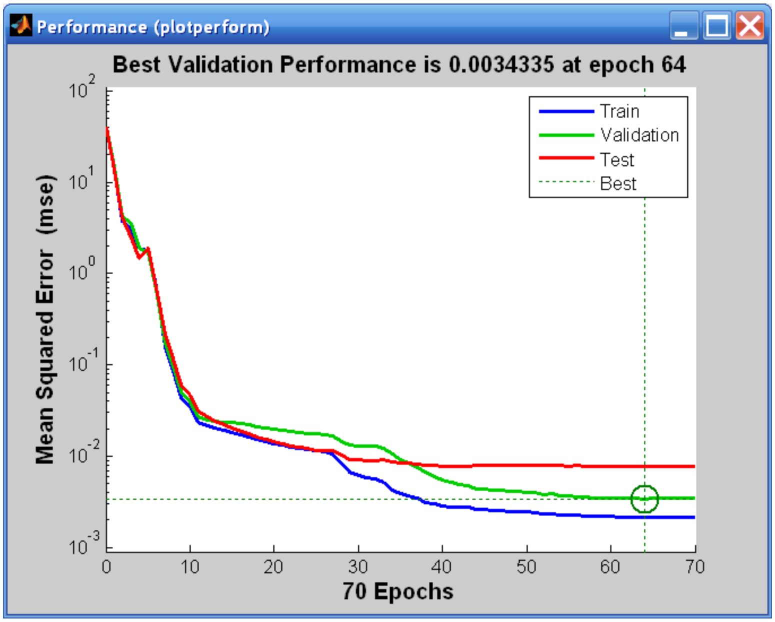
This creates the following diagram of the network.



7 Plot the performance training record to check for potential overfitting.

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

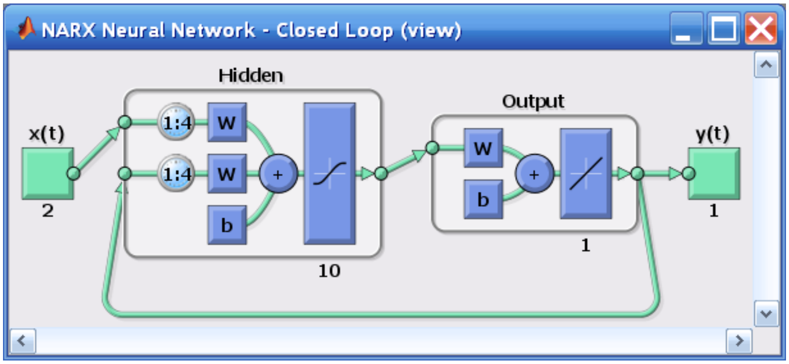
This creates the following figure, which demonstrates that training, validation and testing errors all decreased until iteration 64. It does not appear that any overfitting has occurred, since neither testing or validation error increased before iteration 64.



8 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

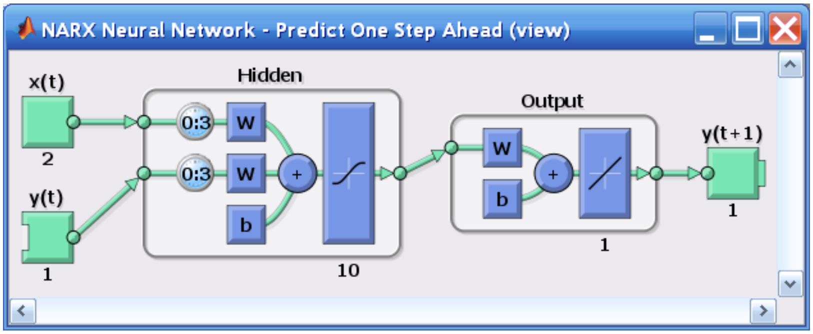
```
netc = closeloop(net);
netc.name = [net.name ' - Closed Loop'];
view(netc)
[xc,xic,aic,tc] = preparets(netc,inputSeries,{},targetSeries);
yc = netc(xc,xic,aic);
perfc = perform(netc,tc,yc)
```

The following figure shows the closed loop network.



9 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)
```



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)” on page 3-13).
- 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” on page 3-15).

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.

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
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

  See also nndemos, nntextdemos, nntextbook.
```

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_datasets;
```

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
```

2

Network Objects, Data and Training Styles

Introduction

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

0 collect data

1 create the network,

2 configure the network,

3 initialize the weights and biases,

4 train the network,

5 validate the network and

6 use the network.

This chapter discusses the basic ideas behind steps 1, 2, 4 and 6. The details of these steps will come in later chapters, as will discussions of steps 3 and 5, since the fine points are specific to the type of network that you are using. (Step 0 generally occurs outside the framework of the Neural Network Toolbox™ software, but it will be discussed in Chapter 3, “Multilayer Networks and Backpropagation Training.”)

The Neural Network Toolbox™ software uses the network object to store all of the information that defines a neural network. This chapter 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 chapter 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.

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 Chapter 1, “Getting Started”. 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 chapter, and the ones that follow, concentrate on command-line operations.

The GUIs described in Chapter 1, “Getting Started” can automatically generate M-files with the command-line implementation of the GUI operations. This provides a nice introduction to the use of the command-line functionality.

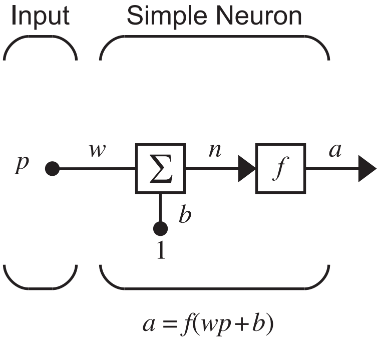
A third level of toolbox use is customization of the toolbox. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox. The fourth level of toolbox usage is the ability to modify any of the M-files contained in the toolbox. Every computational component is written in MATLAB® code and is fully accessible.

The first level of toolbox use (through the GUIs) was described in Chapter 1, “Getting Started,” which also introduced command-line operations. The next seven chapters will discuss the command-line operations in more detail. The customization of the toolbox is described in Chapter 9, “Advanced Topics”.

Neuron Model

Simple Neuron

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



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, $|w\text{-}p|$) are sometimes used. (See “Weight Functions” on page 12-23 for a list of weight functions.) 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. (See “Net Input Functions” on page 12-11 for a list of net input functions.) Chapter 6, “Radial Basis Networks,” 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 2-5. See also “Transfer Functions” on page 12-21.

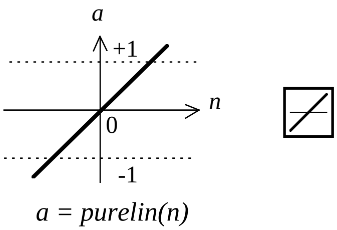
Note that *w* and *b* are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, you can train the network to do a particular job by adjusting the weight or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

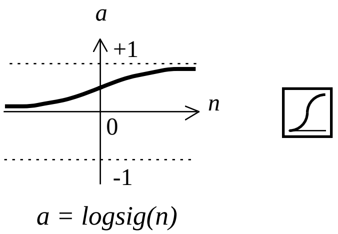
The following figure illustrates the linear transfer function.



Linear Transfer Function

Neurons of this type are used in the final layer of multilayer networks that are used as function approximators. This is demonstrated in Chapter 3, "Multilayer Networks and Backpropagation Training."

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.



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 listing of transfer functions and their icons, see “Transfer Functions” on page 12-21. You can also specify your own transfer functions.

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

Neuron with Vector Input

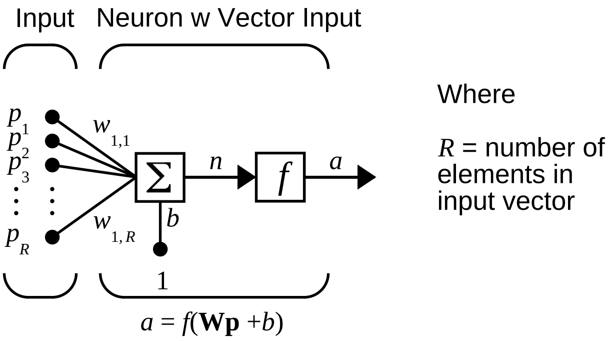
The simple neuron can be extended to handle inputs that are vectors. A neuron with a single R-element input vector is shown below. Here the individual input elements

$$p_1, p_2, \dots p_R$$

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 **Wp**, the dot product of the (single row) matrix **W** and the vector **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 Chapter 6, “Radial Basis Networks.”)



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 Chapter 6, “Radial Basis Networks.”) The net input n is the argument of the transfer function f.

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

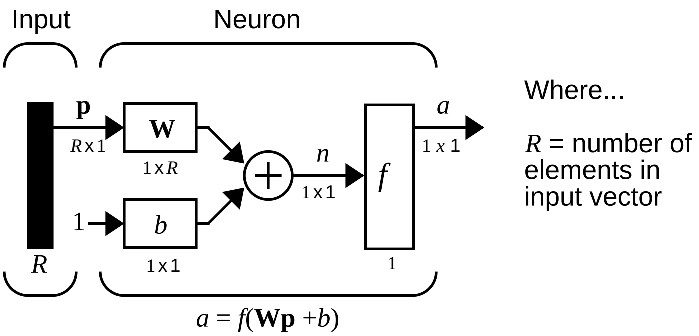
This expression can, of course, be written in MATLAB® code as

```
n = W*p + 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.



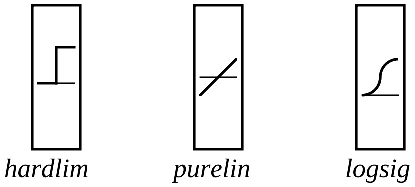
Here the input vector **p** is represented by the solid dark vertical bar at the left. The dimensions of **p** are shown below the symbol **p** in the figure as *R*x1. (Note that a capital letter, such as *R* in the previous sentence, is used when referring to the *size* of a vector.) Thus, **p** is a vector of *R* input elements. These inputs postmultiply the single-row, *R*-column matrix **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 **Wp**. 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 **Wp**), the bias *b*, and the transfer function *f*. The array of inputs, vector **p**, is not included in or called a layer.

As with the "Simple Neuron" on page 2-4, 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 2-5, 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.



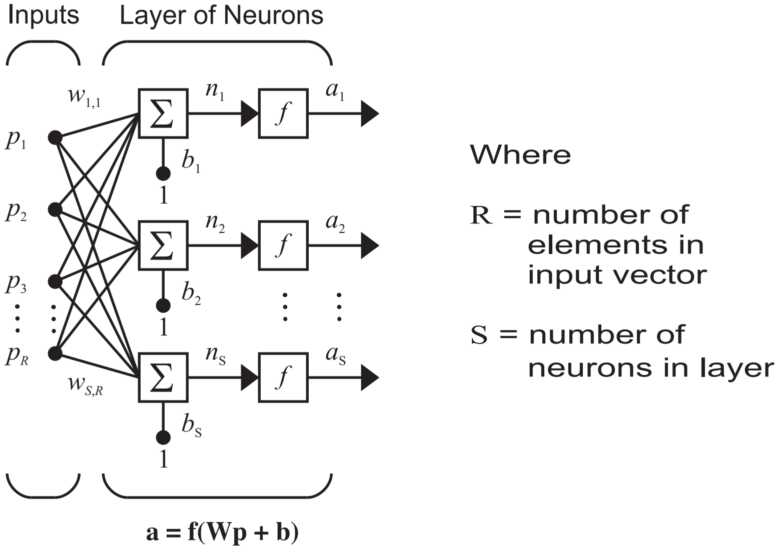
You can experiment with a two-element neuron by running the demonstration program `nnd2n2`.

Network Architectures

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.

A Layer of Neurons

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



In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The ith 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 **n**. Finally, the neuron layer outputs form a column vector **a**. The expression for **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.

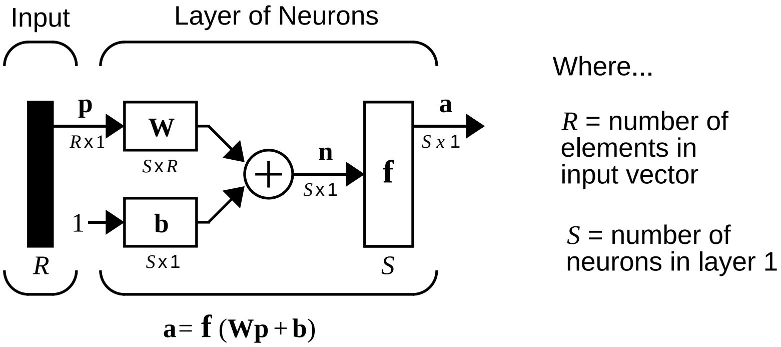
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**.

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

Note that the row indices on the elements of matrix **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.

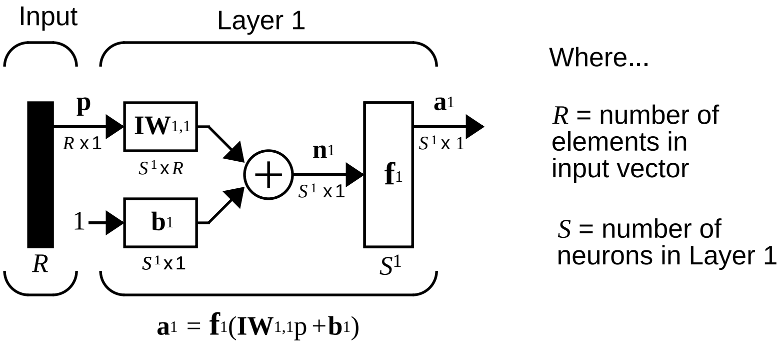


Here **p** is an R length input vector, **W** is an SxR matrix, and **a** and **b** are S length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function blocks.

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 below.

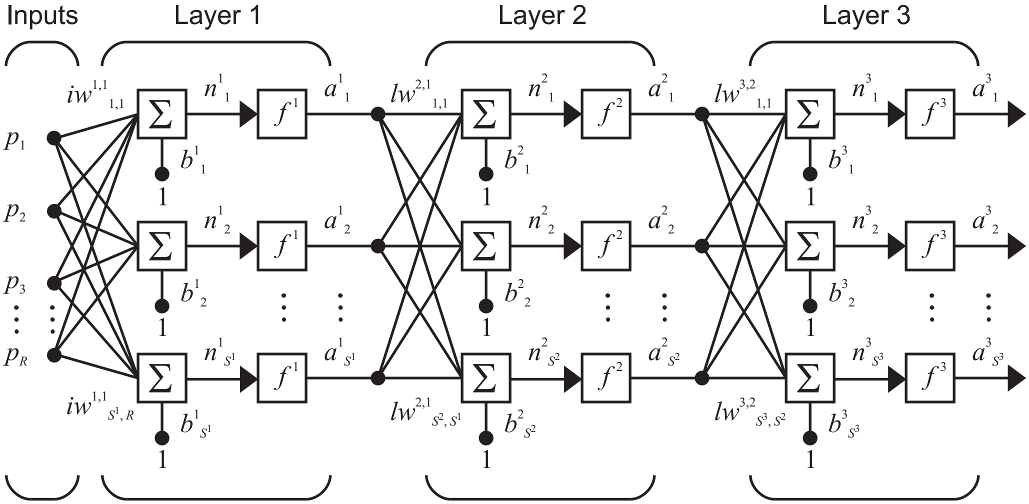


As you can see, the weight matrix connected to the input vector **p** is labeled as an input weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated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

"Multiple Layers of Neurons" 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 **W**, a bias vector **b**, and an output vector **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 below, and in the equations at the bottom of the figure.



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2+\mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

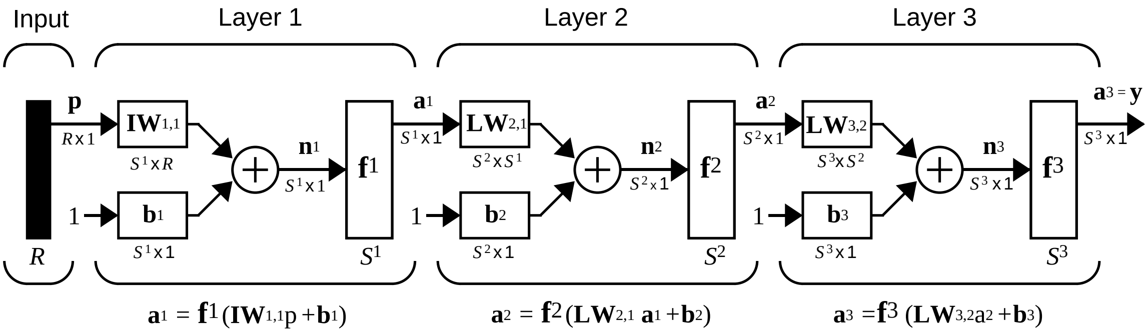
The network shown above has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the bias for each neuron.

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^2xS^1 weight matrix 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-...-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.



$$\mathbf{a}^3 = \mathbf{f}^3\,(\mathbf{LW}_{3,2}\,\mathbf{f}^2\,(\mathbf{LW}_{2,1}\mathrm{f}^1\,(\mathbf{IW}_{1,1}\mathbf{p} + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{b}_3 = \mathbf{y}$$

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 Chapter 3, "Multilayer Networks and Backpropagation Training."

Here it is assumed that the output of the third layer, $\mathbf{a}^3$, 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 “Preprocessing and Postprocessing” in Chapter 3.

Introduction to the Network Object

The easiest way to create a neural network is to use one of the network creation functions (see “New Networks Functions” on page 12-14 for a full list of these functions). To investigate how this is done, you can create a simple, two-layer feedforward network, using the command `feedforwardnet`:

```
net = feedforwardnet
```

This command will display the following

```
net =

Neural Network

              name: 'Feed-Forward Neural Network'
    efficiencyMode: 'speed'
 efficiencyOptions: .cacheDelayedInputs, .flattenTime,
                    .memoryReduction
          userdata: (your custom info)

    dimensions:

         numInputs: 1
         numLayers: 2
        numOutputs: 1
    numInputDelays: 0
    numLayerDelays: 0
 numFeedbackDelays: 0
 numWeightElements: 20
        sampleTime: 1

    connections:

       biasConnect: [1; 1]
      inputConnect: [1; 0]
      layerConnect: [0 0; 1 0]
     outputConnect: [0 1]

    subobjects:
        inputs: 1x1 cell array of 1 nnInput
        layers: 2x1 cell array of 2 nnLayers
       outputs: 1x2 cell array of 1 nnOutput
        biases: 2x1 cell array of 2 nnBiass
  inputWeights: 2x1 cell array of 1 nnWeight
  layerWeights: 2x2 cell array of 1 nnWeight

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}
     plotParam: 2x2 cell array of 1 nnParam
      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, here 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 a lack of connection. For this example, there is a single one in the 2,1 element 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}
```

This will display

```
Neural Network Layer

             name: 'Hidden'
       dimensions: 20
      distanceFcn: (none)
    distanceParam: (none)
        distances: []
          initFcn: 'initnw'
      netInputFcn: 'netsum'
    netInputParam: (none)
        positions: []
            range: [20x2 double]
```

```
             size: 20
      topologyFcn: (none)
      transferFcn: 'tansig'
    transferParam: (none)
         userdata: (your custom info)
```

The number of neurons in this layer is 20, 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}
```

This produces the following response.

```
Neural Network Weight

          delays: 0
         initFcn: 'initzero'
      initConfig: .inputSize
           learn: true
        learnFcn: 'learngdm'
      learnParam: .lr, .mc
            size: [0 20]
       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 20. 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 determined by the number of elements 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 (see “New Networks Functions” on page 12-14). It is usually only when you want to override the system defaults that it is necessary to access the network object directly. Later chapters will demonstrate how this is done for particular networks and training methods.

If you would like to investigate the network object in more detail, you will find that the object listings, such as the one shown above, contains links to help files on each subobject. Just click on the links, and you can selectively investigate those parts of the object that are of interest to you.

Configuration

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: 'initzero'
        initConfig: .inputSize
             learn: true
          learnFcn: 'learngdm'
        learnParam: .lr, .mc
              size: [1 20]
         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'
        processFcns: {'removeconstantrows','mapminmax'}
       processParam: {1x3} containing 2 nnParam
    processSettings: {1x3} containing 3 nnSetting
     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 Chapter 3, "Multilayer Networks and Backpropagation Training," 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 Chapter 3, "Multilayer Networks and Backpropagation Training."

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.

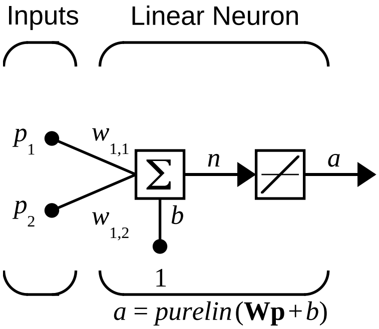
Data Structures

This section 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

$$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix} \text{ and } b = \begin{bmatrix} 0 \end{bmatrix}$$

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:

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

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

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

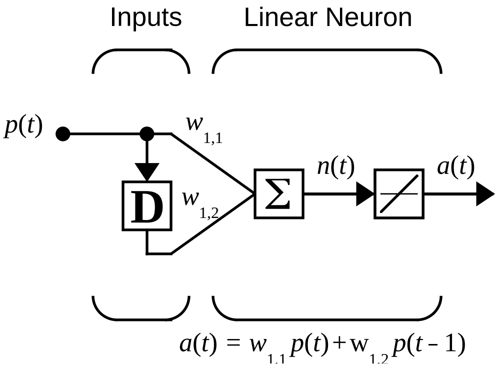
You can now simulate the network:

```
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

$$\mathbf{W} = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

The command is

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

Suppose that the input sequence is

$$\mathbf{p}_1 = \begin{bmatrix} 1 \end{bmatrix},\ \mathbf{p}_2 = \begin{bmatrix} 2 \end{bmatrix},\ \mathbf{p}_3 = \begin{bmatrix} 3 \end{bmatrix},\ \mathbf{p}_4 = \begin{bmatrix} 4 \end{bmatrix}$$

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 2-24, if you use a concurrent set of inputs you have

$$\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix},\quad \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix},\quad \mathbf{p}_3 = \begin{bmatrix}3\end{bmatrix},\quad \mathbf{p}_4 = \begin{bmatrix}4\end{bmatrix}$$

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:

$$\mathbf{p}_1(1) = \begin{bmatrix}1\end{bmatrix},\ \mathbf{p}_1(2) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_1(3) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_1(4) = \begin{bmatrix}4\end{bmatrix}$$

$$\mathbf{p}_2(1) = \begin{bmatrix}4\end{bmatrix},\ \mathbf{p}_2(2) = \begin{bmatrix}3\end{bmatrix},\ \mathbf{p}_2(3) = \begin{bmatrix}2\end{bmatrix},\ \mathbf{p}_2(4) = \begin{bmatrix}1\end{bmatrix}$$

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:

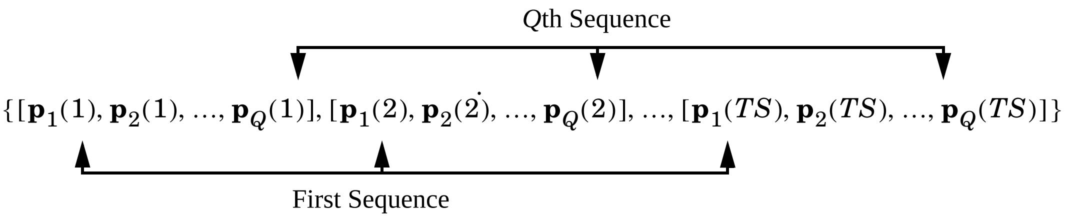
```
A = net(P);
```

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.


Qth Sequence
{[p1(1), p2(1), ..., pQ(1)], [p1(2), p2(2), ..., pQ(2)], ..., [p1(TS), p2(TS), ..., pQ(TS)]}
First Sequence

In this section, you apply sequential and concurrent inputs to dynamic networks. In “Simulation with Concurrent Inputs in a Static Network” on page 2-23, 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 “Training Styles” on page 2-29.

Training Styles

This section 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 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 demonstrates 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

$$\mathbf{t}_1 = \begin{bmatrix}4\end{bmatrix},\ \mathbf{t}_2 = \begin{bmatrix}5\end{bmatrix},\ \mathbf{t}_3 = \begin{bmatrix}7\end{bmatrix},\ \mathbf{t}_4 = \begin{bmatrix}7\end{bmatrix}$$

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 2-23 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,pf] = 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,pf] = 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 `Pi` 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,pf] = 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 adaptation function for the linear network) and learnwh (the default learning function for the weights and biases). trains uses Widrow-Hoff learning.

```
[net,a,e,pf] = 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 adaptation 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')
```

Multilayer Networks and Backpropagation Training

Introduction

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 (see “Focused Time-Delay Neural Network (timedelaynet)” on page 4-12). This chapter demonstrates how you can use the multilayer network. It also illustrates the basic procedures for designing any neural network.

Note The training functions described in this chapter 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 six primary steps:

0. collect data
1. create the network,
2. configure the network,
3. initialize the weights and biases,
4. train the network,
5. validate the network (post-training analysis) and
6. use the network.

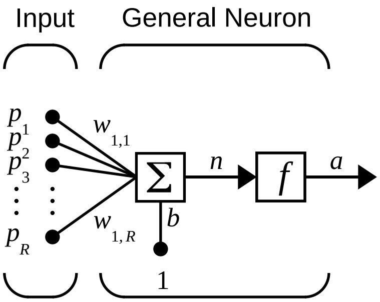
Chapter 2, “Network Objects, Data and Training Styles” introduced steps 1 and 2, and described the basics of steps 4 and 6. The current chapter will demonstrate all six steps of the design process for multilayer networks. It will also discuss step 0, which may happen outside the framework of the Neural Network Toolbox™ software, but which is critical to the success of the design process. The next section describes the architecture of the multilayer feedforward network. This is followed by other sections that describe the six steps of the neural network design process.

Multilayer Neural Network Architecture

This section presents the architecture of the multilayer feedforward neural network.

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.

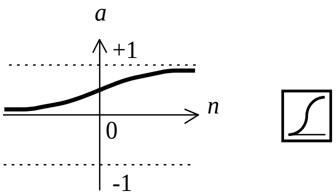


$a = f(\mathbf{Wp} + b)$

Where

R = number of elements in input vector

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

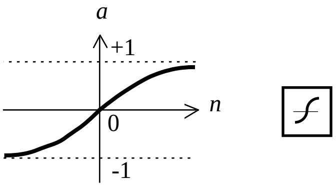


$a = logsig(n)$

Log-Sigmoid Transfer Function

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

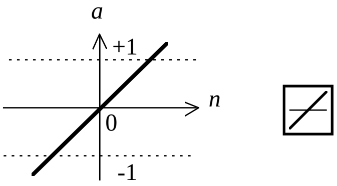
Alternatively, multilayer networks can use the tan-sigmoid transfer function `tansig`.



a = tansig(n)

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.



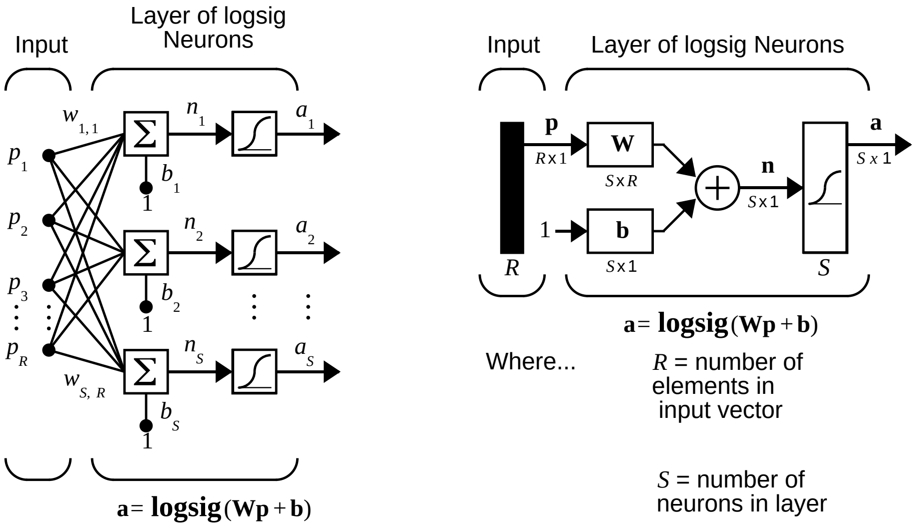
a = purelin(n)

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. See Chapter 9, “Advanced Topics.”

Feedforward Network

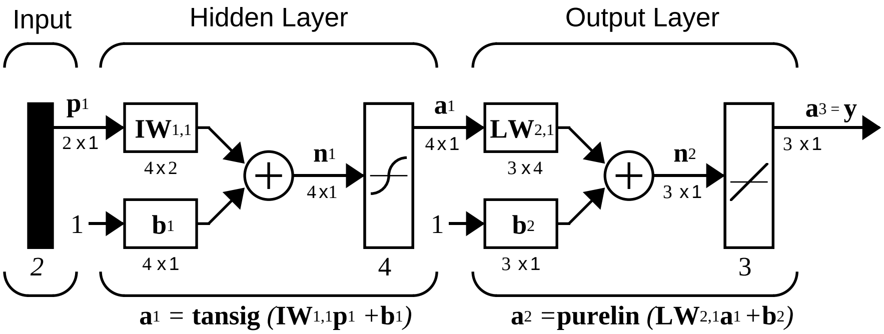
A single-layer network of S `logsig` neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



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).

As noted in Chapter 2, “Network Objects, Data and Training Styles,” 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.

Collect and Prepare the Data

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. The next two subsections describe these two steps.

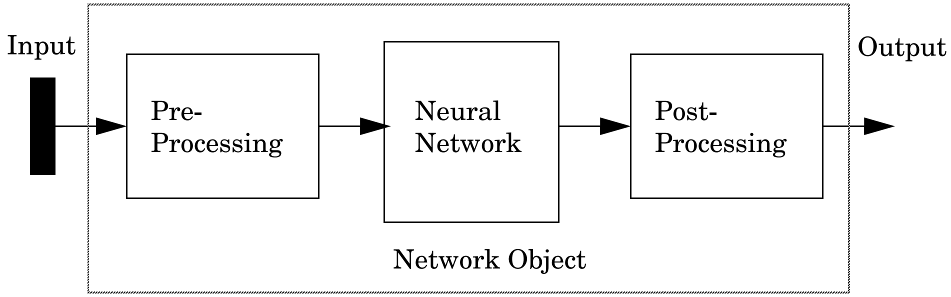
Preprocessing and Postprocessing

Neural network training can be made 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 ($\exp(-3) \cong 0.05$). 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`th input processing function for the network input as follows:

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

You can access or change the parameters of the $\mathtt{i}^{th}$ 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. See the function pages for detailed descriptions. In most cases, you will not need to use them directly, since the preprocessing steps become part of the network object, as was described in Chapter 2, "Network Objects, Data and Training Styles." When you simulate or train the network, the preprocessing and postprocessing will be done automatically.

Function	Algorithm
`mapminmax`	Normalize inputs/targets to fall in the range [-1,1]
`mapstd`	Normalize inputs/targets to have zero mean and unity variance
`processpca`	Extract principal components from the input vector
`fixunknowns`	Process unknown inputs
`removeconstantrows`	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.

Dividing the Data

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 "Improving Generalization" on page 9-34.

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
`dividerand`	Divide the data randomly (default)
`divideblock`	Divide the data into contiguous blocks
`divideint`	Divide the data using an interleaved selection
`divideind`	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 the Network

After the data has be 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, as was shown in Chapter 2, “Network Objects, Data and Training Styles.” 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. The section “Initializing Weights (init)” on page 3-13 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. They are introduced in Chapter 4, “Dynamic Networks”.

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);
```

For specifics on how the weights are initialized, see Chapter 9, “Advanced Topics.”

Train the Network

Once 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 “Error Weighting” on page 4-38.) 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 chapter describes batch mode training with the `train` command. Incremental training with the `adapt` command is discussed in “Incremental Training with adapt” on page 2-29 and in Chapter 8, “Adaptive Filters and Adaptive Training”. 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

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

where $\mathbf{x}_k$ is a vector of current weights and biases, $\mathbf{g}_k$ is the current gradient, and α_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 which use gradient- or Jacobian-based methods, is shown in the following table. For details on specific algorithms, see their function pages. See also Chapters 11 and 12 of [HDB96] for a detailed description of several of these techniques.

Function	Algorithm
`trainlm`	Levenberg-Marquardt
`trainbr`	Bayesian Regularization
`trainbfg`	BFGS Quasi-Newton
`trainrp`	Resilient Backpropagation
`trainscg`	Scaled Conjugate Gradient
`traincgb`	Conjugate Gradient with Powell/Beale Restarts
`traincgf`	Fletcher-Powell Conjugate Gradient
`traincgp`	Polak-Ribiére Conjugate Gradient
`trainoss`	One Step Secant
`traingdx`	Variable Learning Rate Gradient Descent

Function	Algorithm
`traingdm`	Gradient Descent with Momentum
`traingd`	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 "Speed and Memory Comparison for Training Multilayer Networks" 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. (See Chapter 4, "Dynamic Networks" and Chapter 6, "Radial Basis Networks" for examples of other networks that can use the same training functions that are shown in the table above.) 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, as described in Chapter 9, "Advanced Topics," and then train them using any of the training functions in the table above. The gradients and Jacobians will be automatically computed for you.

Efficiency and Memory Reduction

There are some network parameters that are helpful when training large networks or using large data sets. For example, the parameter

```
net.effficiency.memoryReduction
```

can be used to reduce the amount of memory that you use while training or simulating the network. If this parameter is set to 1 (the default value), the maximum memory is used, and the fastest training times will be achieved. If this parameter is set to 2, then the data is divided into two parts. All calculations (like gradients and Jacobians) are done first on part one, and then later on part two. Any intermediate variables used in part 1 are released before the part 2 calculations are done. This can save significant memory, especially for the `trainlm` training function. If `memoryReduction` is set to N, then the data is divided into N parts, which are computed separately. The larger the value of N, the larger the reduction in memory use, although the amount of reduction diminishes as N is increased.

There is a drawback to using memory reduction. A computational overhead is associated with computing the Jacobian and gradient in submatrices. If you have enough memory available, then it is better to leave `memoryReduction` set to 1 and to compute the full Jacobian or gradient in one step. If you have a large training set, and you are running out of memory, then you should set `memoryReduction` to 2 and try again. If you still run out of memory, continue to increase `memoryReduction`.

Generalization

Properly trained multilayer networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an accurate ouput, if the new input is similar to inputs used in the training set. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs. There are two features of the Neural Network Toolbox™ software that are designed to improve network generalization: regularization and early stopping. These features and their use are discussed in detail in “Improving Generalization” on page 9-34. A few comments on using these techniques are given in the following.

The default generalization feature for the multilayer feedforward network is early stopping. Data are automatically divided into training, validation and test sets, as described in “Dividing the Data” on page 3-10. The error on the validation set is monitored during training, and the training is stopped when the validation increases over net.trainParam.max_fail iterations. If you wish to disable early stopping, you can assign no data to the validation set. This can be done by setting net.divideParam.valRatio to zero.

An alternative method for improving generalization is regularization. Regularization can be done automatically by using the Bayesian regularization training function trainbr. This can be done by setting net.trainFcn to `trainbr'. This will also automatically move any data in the validation set to the training set.

Training Example

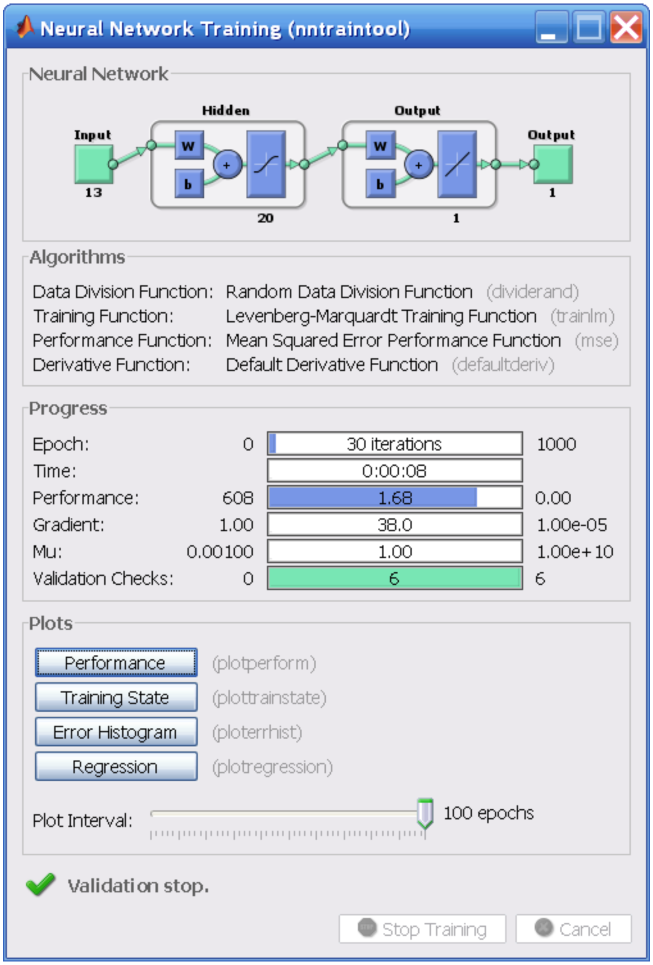
To demonstrate the training process, execute the following commands:

```
load house_dataset
net = feedforwardnet;
[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 wish 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.)



Algorithms

Data Division Function: Random Data Division Function (dividerand)
Training Function: Levenberg-Marquardt Training Function (trainlm)
Performance Function: Mean Squared Error Performance Function (mse)
Derivative Function: Default Derivative Function (defaultderiv)

Progress

Epoch:	0	30 iterations	1000
Time:		0:00:08	
Performance:	608	1.68	0.00
Gradient:	1.00	38.0	1.00e-05
Mu:	0.00100	1.00	1.00e+10
Validation Checks:	0	6	6

Plots

(plotperform)
(plottrainstate)
(ploterrhist)
(plotregression)

Plot Interval: 100 epochs

✓ Validation stop.

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

Parameter	Stopping Criteria
`min_grad`	Minimum Gradient Magnitude
`max_fail`	Maximum Number of Validation Increases
`time`	Maximum Training Time
`goal`	Minimum Performance Value
`epochs`	Maximum Number of Training Epochs (Iterations)

The training will also stop if you click the “Stop Training” button in the training window. You may want to do this if the performance function fails to decrease significantly over many iterations. It is always possible to continue the training by re-issuing 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 the next section.

Post-Training Analysis (Network Validation)

When the training is complete, you will want to check the network performance and determine if any changes need to be made to the training process, the network architecture or the data sets. The first thing to do is to check the training record, tr, which was the second argument returned from the training function.

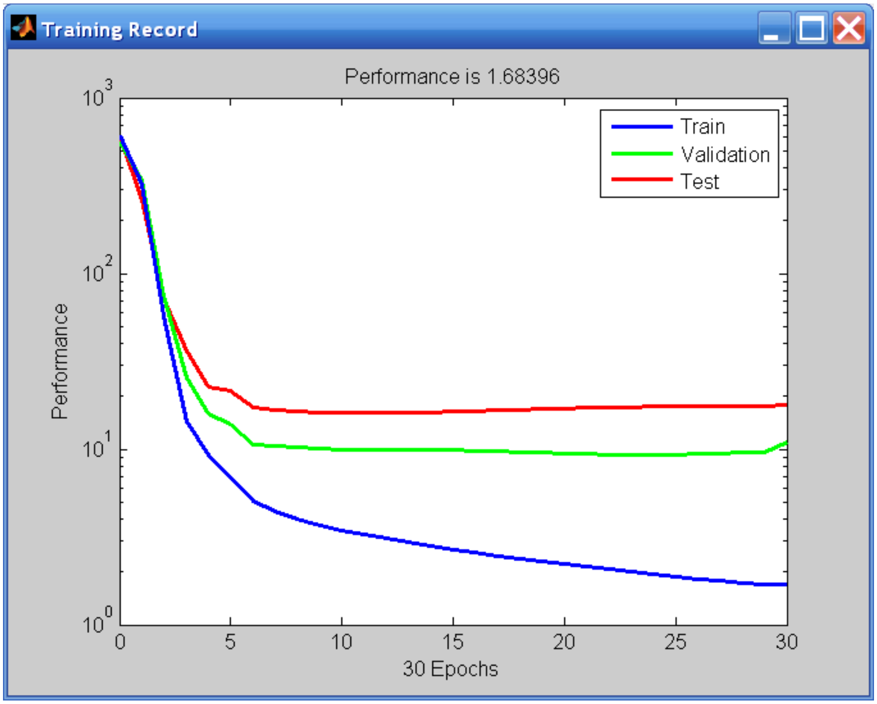
```
tr =

                  trainFcn: 'trainlm'
                trainParam: [1x1 nnParam]
                 performFcn: 'mse'
              performParam: [1x1 nnParam]
                  derivFcn: 'defaultderiv'
                 divideFcn: 'dividerand'
                divideMode: 'sample'
               divideParam: [1x1 nnParam]
                  trainInd: [1x354 double]
                    valInd: [1x76 double]
                   testInd: [1x76 double]
                      stop: 'Validation stop.'
                num_epochs: 30
                 trainMask: {[1x506 double]}
                   valMask: {[1x506 double]}
                  testMask: {[1x506 double]}
                best_epoch: 24
                      goal: 0
                    states: {1x8 cell}
                     epoch: [1x31 double]
                      time: [1x31 double]
                      perf: [1x31 double]
                     vperf: [1x31 double]
                     tperf: [1x31 double]
                        mu: [1x31 double]
                  gradient: [1x31 double]
                  val_fail: [1x31 double]
       physicalMemoryTotal: [1x31 double]
   physicalMemoryAvailable: [1x31 double]
        physicalMemoryUsed: [1x31 double]
 physicalMemoryPercentUsed: [1x31 double]
```

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 wish 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, as in

```
plotperf(tr)
```

This produces the following figure. As indicated by `tr.best_epoch`, the iteration at which the validation performance reached a minimum was 24. The training continued for 6 more iterations before the training stopped.



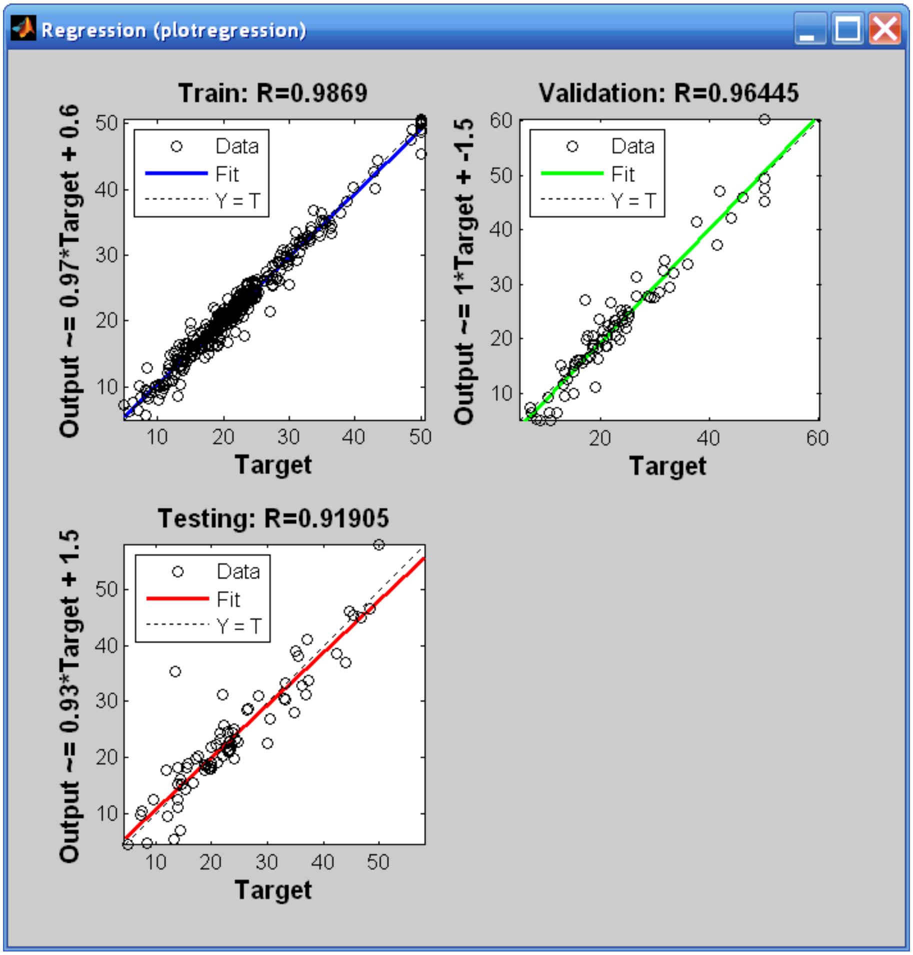
This figure doesn't indicate any major problems with the training. The validation and test curves are very similar. If the test curve had increased significantly before the validation curve increased, then it is possible that some overfitting might have occurred.

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 result is shown in the following figure. The three axes represent the training, validation and testing data. The dashed line in each axis 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 your initialize a feed-forward 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.

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 wish to find the network response to the 5th 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);
```

Automatic Code Generation

It is often easiest to learn how to use the Neural Network Toolbox™ software by starting with some example code and modifying it to suit your problem. It is very simple to create example code by using the GUIs described in Chapter 1, "Getting Started." In particular, to generate some sample code to reproduce the function fitting examples shown in this chapter, you can run the neural fitting GUI, `nftool`. Select the house pricing data from the GUI, and after you have trained the network, click the "Advanced Script" button on the final panel of the GUI. This will automatically generate code that will demonstrate most of the options that are available to you when following the general network design process for function fitting problems. You can customize the generated script to fit your needs.

If you are interested in using a multilayer neural network for pattern recognition, use the pattern recognition GUI, `nprtool`. It will lead you through a similar set of design steps for pattern recognition problems, and can then generate example code demonstrating the many options that are available for pattern recognition networks.

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 `trainrp`.

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 “Improving Generalization” on page 9-34. This topic is also discussed starting on page 11-21 of [HDB96].

4

Dynamic Networks

Introduction

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.

The training of dynamic networks is very similar to the training of static feedforward networks, as discussed in Chapter 3, “Multilayer Networks and Backpropagation Training.” As described in that chapter, the work flow for the general neural network design process has six primary steps:

0 collect data

1 create the network,

2 configure the network,

3 initialize the weights and biases,

4 train the network,

5 validate the network (post-training analysis) and

6 use the network.

These design steps, and the training methods discussed in Chapter 3, “Multilayer Networks and Backpropagation Training,” can also be used for dynamic networks. The main differences in the design process occur because the inputs to the dynamic networks are time sequences. (See “Simulation with Sequential Inputs in a Dynamic Network” on page 2-24 and “Batch Training with Dynamic Networks” on page 2-33 for previous discussions of simulation and training of dynamic networks.) This results in some additional initialization procedures prior to training or simulating a dynamic network. There are also special validation procedures that can be used for dynamic networks. (These were discussed in “Time Series Prediction” on page 1-66.)

This chapter will begin by explaining how dynamic networks operate and by giving examples of applications for dynamic networks. Then you will be introduced to the general framework for representing dynamic networks in the toolbox. This will allow you to design your own specialized dynamic networks, which can then be trained using existing toolbox training functions. Next, the chapter describes several standard dynamic network architectures that you can create with a single command. Each will be demonstrated with a practical application. Finally, the chapter provides an example of creating and training a custom network.

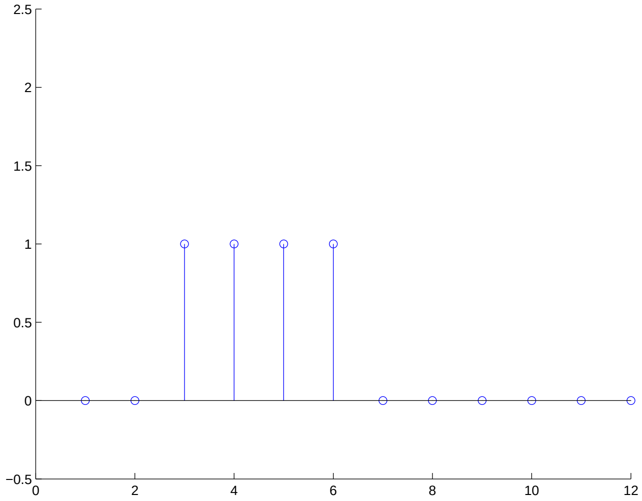
Examples of Dynamic Networks

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 the section on applying sequential inputs to a dynamic network on page 2-24.)

The following command creates a pulse input sequence and plots it:

```
p = {0 0 1 1 1 1 0 0 0 0 0 0};

stem(cell2mat(p))
```

The next figure show the resulting pulse.

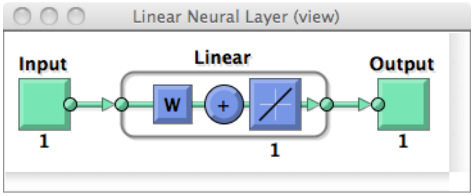


Now create a static network and find the network response to the pulse sequence. The following commands create a simple linear network with one layer, one neuron, no bias, and a weight of 2:

```
net = linearlayer;
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = 2;
```

To view the network, use the following command:

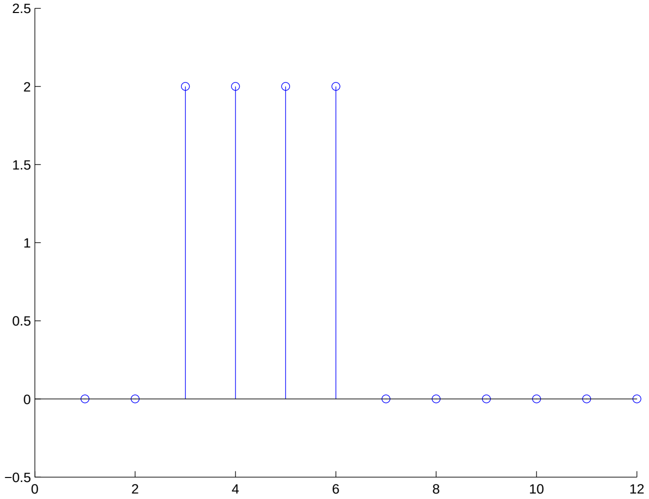
```
view(net)
```



You can now simulate the network response to the pulse input and plot it:

```
a = net(p);
stem(cell2mat(a))
```

The result is shown in the following figure. 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.

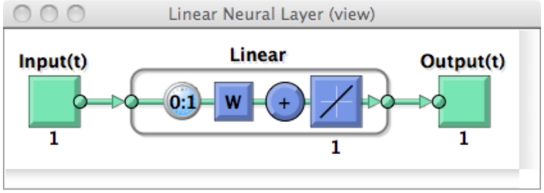


Now create a dynamic network, but one that does not have any feedback connections (a nonrecurrent network). You can use the same network used on page 2-24, which was a linear network with a tapped delay line on the input:

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

To view the network, use the following command:

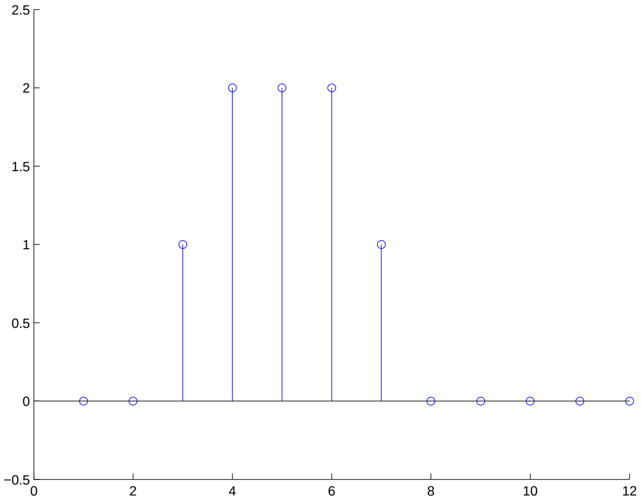
```
view(net)
```



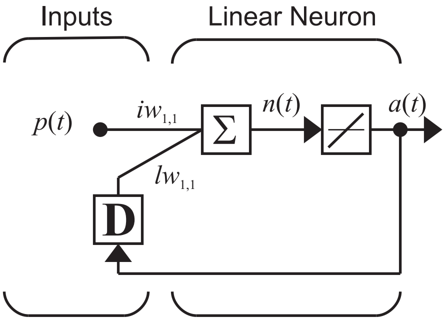
You can again simulate the network response to the pulse input and plot it:

```
a = net(p);
stem(cell2mat(a))
```

The response of the dynamic network, shown in the following figure, 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.

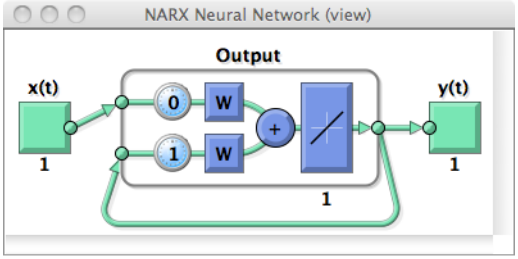


$$a(t) = iw_{1,1}p(t)+lw_{1,1}a(t\text{-}1)$$

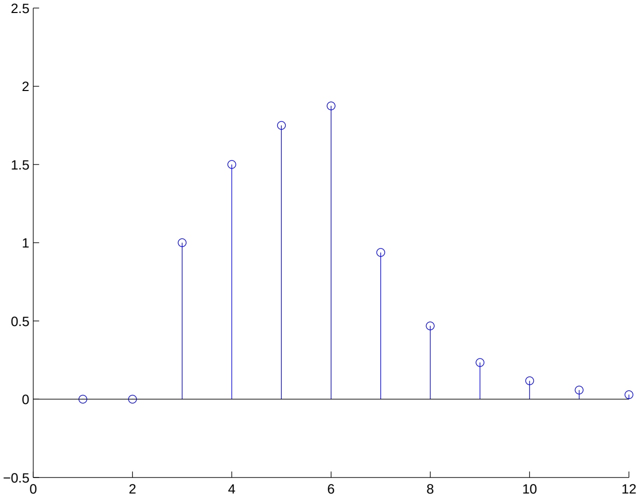
You can create the network, view it and simulate it with the following commands. The `narxnet` command is discussed in "NARX Network (narxnet, closeloop)" on page 4-21.

```
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)
a = net(p);
stem(cell2mat(a))
```

The resulting network diagram is shown below.



The following figure is the plot of the network response.



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 Chapter 5, “Control Systems.” Dynamic networks are also well suited for filtering. You will see the use of some linear dynamic networks for filtering in Chapter 8, “Adaptive Filters and Adaptive Training,” and some of those ideas are extended in this chapter, using nonlinear dynamic networks.

Dynamic Network Structures

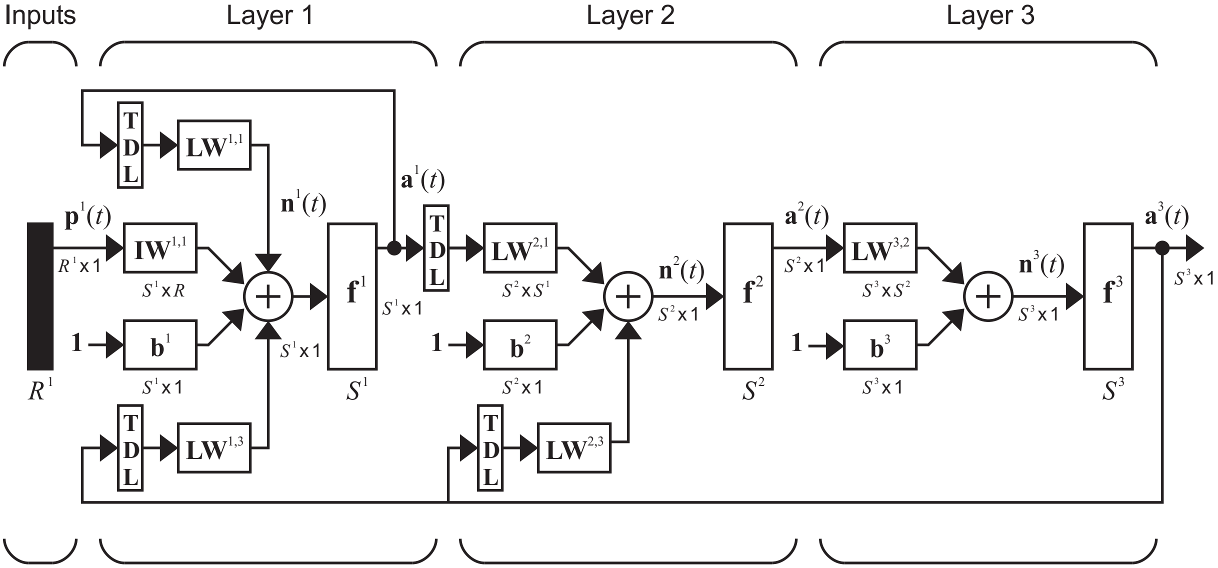
The Neural Network Toolbox™ software is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with the toolbox. Here is a basic description of the LDDN.

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 $\mathbf{IW}^{i,j}$ (`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 $\mathbf{LW}^{i,j}$ (`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 chapter 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 chapter can be created using the generic network command, as explained in Chapter 9, “Advanced Topics.”

Dynamic Network Training

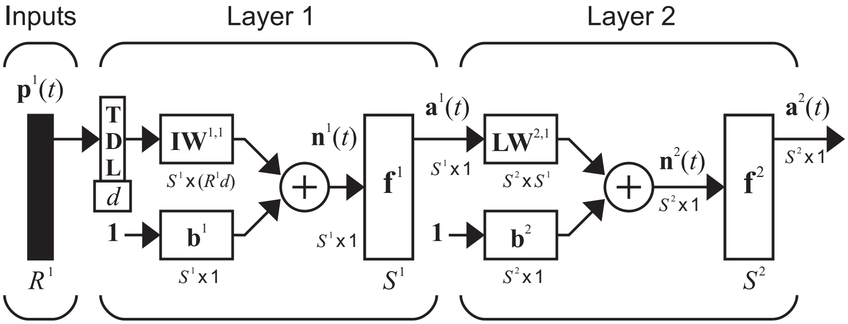
Dynamic networks are trained in the Neural Network Toolbox™ software using the same gradient-based algorithms that were described in Chapter 3, “Multilayer Networks and Backpropagation Training.” You can select from any of the training functions that were presented in that chapter. Examples are provided 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 the simple recurrent network shown on page 4-7. 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 $a(t \quad 1)$, 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 chapter demonstrate 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.

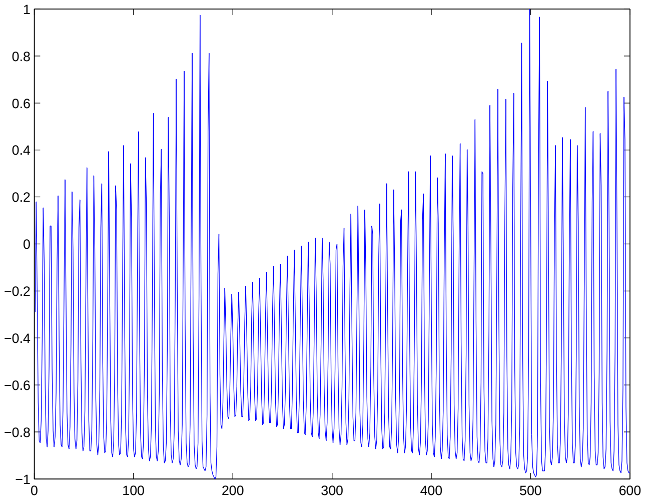
Focused Time-Delay Neural Network (timedelaynet)

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 demonstrates 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 = '';
```

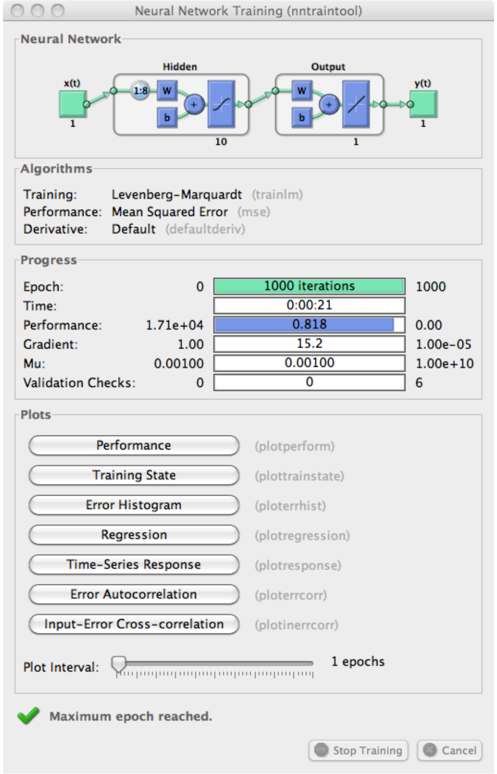
Arrange the network inputs and targets for training. Because the network has a tapped delay line with a maximum delay of 8, begin by predicting the ninth value of the time series. You also need to load the tapped delay line with the eight initial values of the time series (contained in the variable `Pi`):

```
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 will appear.



Algorithms

Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Derivative: Default (defaultderiv)

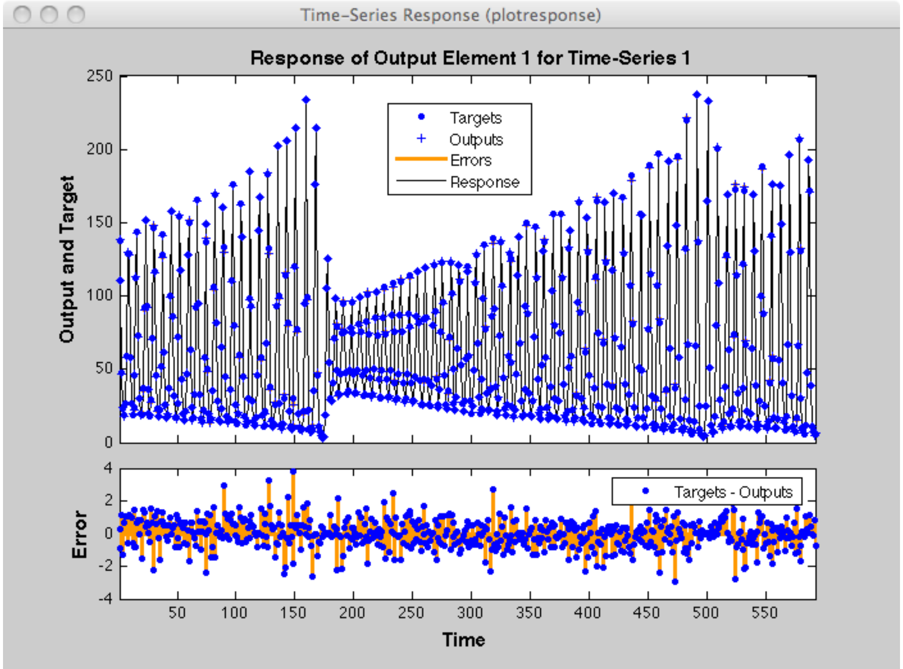
Progress

Epoch:	0	1000 iterations	1000
Time:		0:00:21	
Performance:	1.71e+04	0.818	0.00
Gradient:	1.00	15.2	1.00e-05
Mu:	0.00100	0.00100	1.00e+10
Validation Checks:	0	0	6

Plot Interval: 1 epochs

✓ Maximum epoch reached.

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 will appear.



Now simulate the network and determine the prediction error.

```
yp = ftdnn_net(p,Pi);
e = gsubtract(yp,t);
rmse = sqrt(mse(e))

rmse =
    0.9047
```

(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 0.9407 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 chapter.

Preparing Data (preparets)

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 on page 4-13.) There is a toolbox function that facilitates the data preparation for dynamic (time series) networks - `preparets`. 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)
```

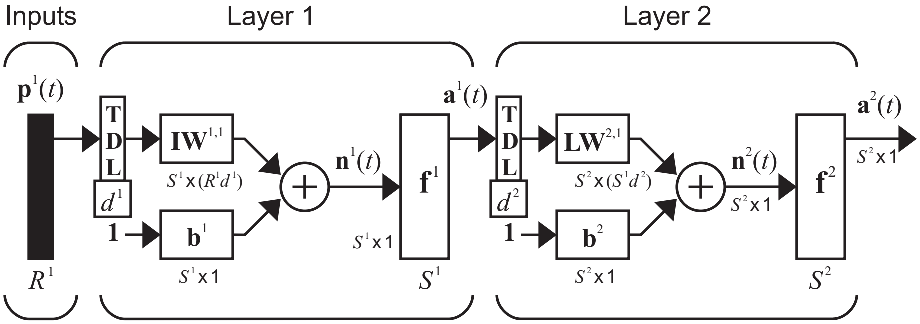
The input arguments for `preparets` are the network object (`net`), the external (non-feedback) input to the network (`inputs`), the non-feedback target (`targets`), the feedback target (`feedback`), and the error weights (`EW`) (see “Error Weighting” on page 4-38). The difference between external and feedback signals will become clearer when the NARX network is described in “NARX Network (narxnet, closeloop)” on page 4-21. 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`).

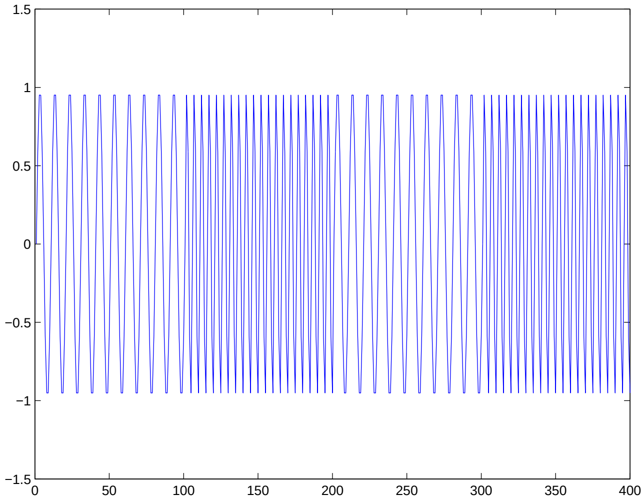
Using preparets eliminates the need to manually shift inputs and targets and load tapped delay lines. This is especially useful for more complex networks.

Distributed Time-Delay Neural Network (newdtdnn)

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 figure below 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.



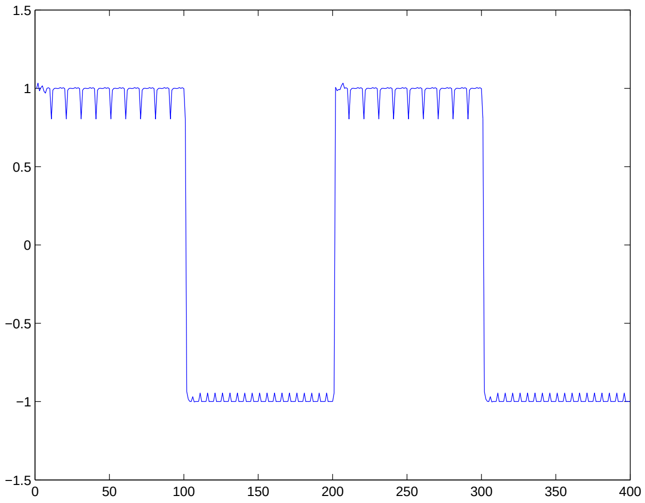
The following code creates this signal and a target network output. The target output is 1 when the input is at the low frequency and -1 when the input is at the high frequency.

```
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 Chapter 3, “Multilayer Networks and Backpropagation Training.”)

```
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);
yp = cell2mat(yp);
plotresponse(t,yp);
```

The following figure shows the trained network output. 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.

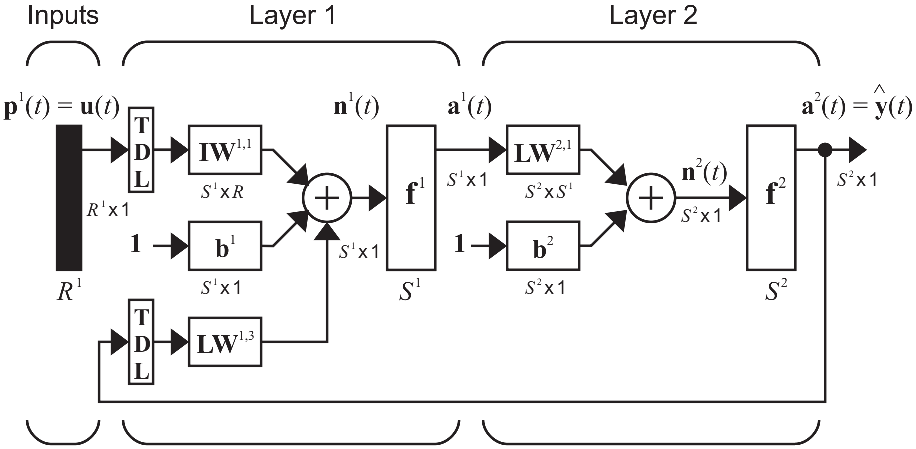
NARX Network (narxnet, closeloop)

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

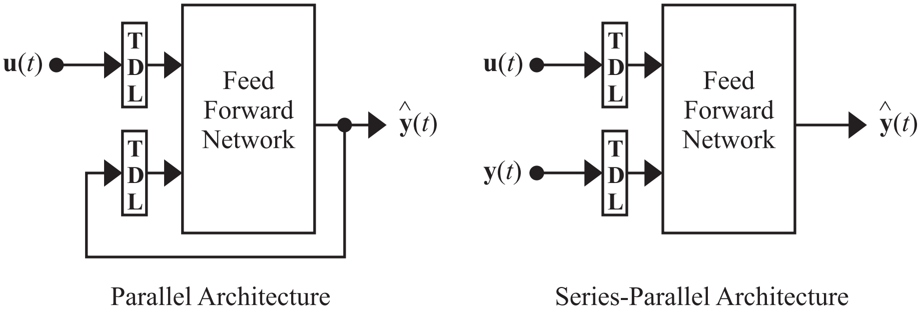
$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u))$$

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 demonstrated in another important application, the modeling of nonlinear dynamic systems.

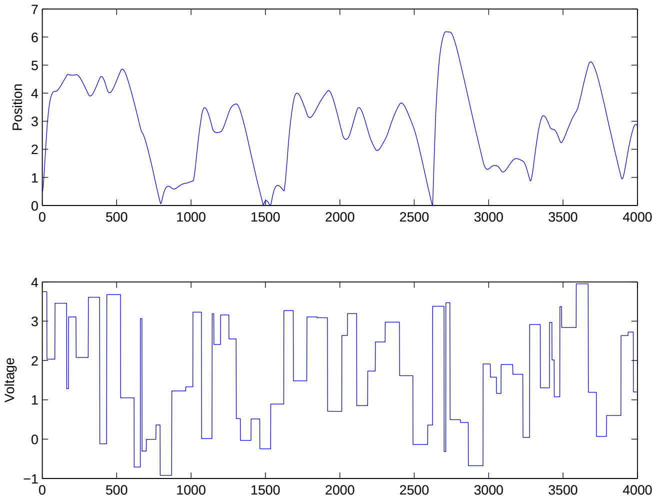
Before demonstrating the training of the NARX network, an important configuration that is useful in training needs explanation. You can consider the output of the NARX network to be an estimate of the output of some nonlinear dynamic system that you are trying to model. The output is fed back to the 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 demonstrates the use of the series-parallel architecture for training an NARX network to model a dynamic system.

The example of the NARX network is the magnetic levitation system described beginning 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 an 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, so p is a cell array with two rows:

```
load magdata
[u,us] = mapminmax(u);
[y,ys] = mapminmax(y);
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 preparets:

```
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,{},y);
```

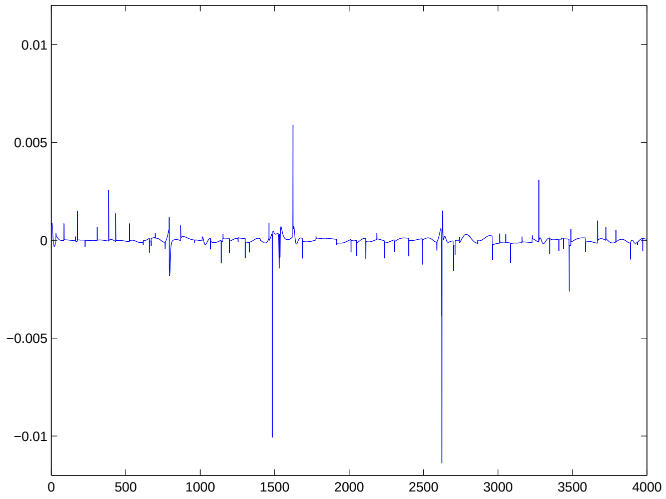
(Notice that the y sequence is considered a feedback signal, which is an input that is also an output (target). Later, when you close the loop, the appropriate output will be connected to the appropriate input.) Now you are ready to train the network.

```
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)
```

The result is displayed in the following plot. 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 demonstrated.

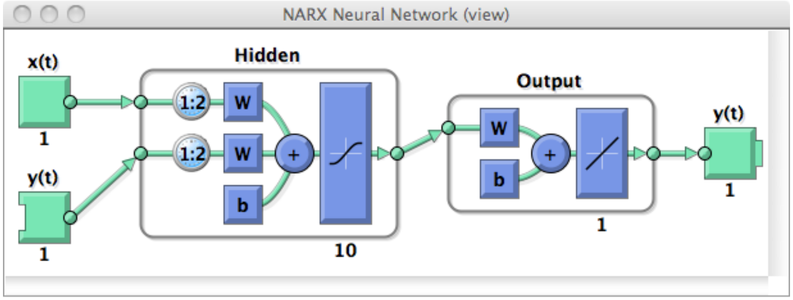


There is a toolbox function (`closeloop`) for converting NARX (and other) networks from the series-parallel configuration (open loop), which is useful for training, to the parallel configuration (closed loop), which is useful for multi-step-ahead prediction. The following command illustrates how to convert the network that you just trained to parallel form.

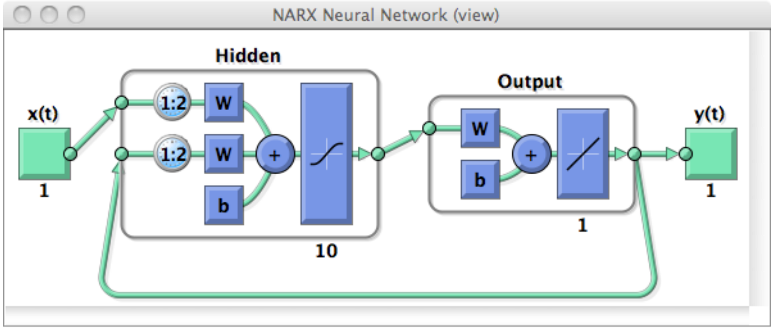
```
narx_net_closed = closeloop(narx_net);
```

To see the differences between the two networks, you can use the view command:

```
view(narx_net)
```



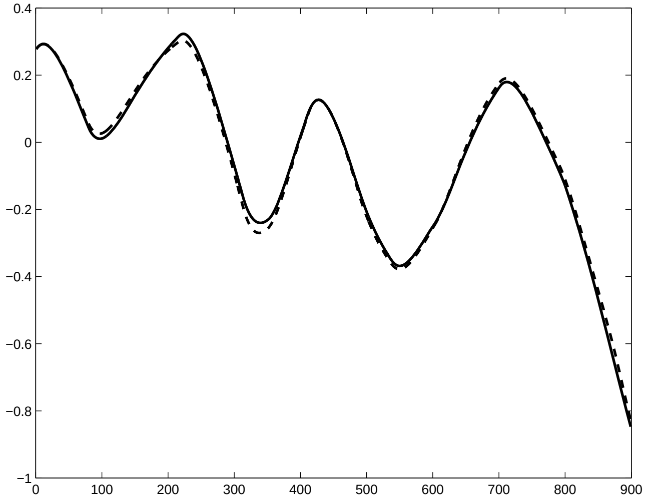
```
view(narx_net_closed)
```



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);
plot([cell2mat(yp1)' cell2mat(t1)'])
```

The following figure illustrates the iterated prediction. The solid line is the actual position of the magnet, and the dashed 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.

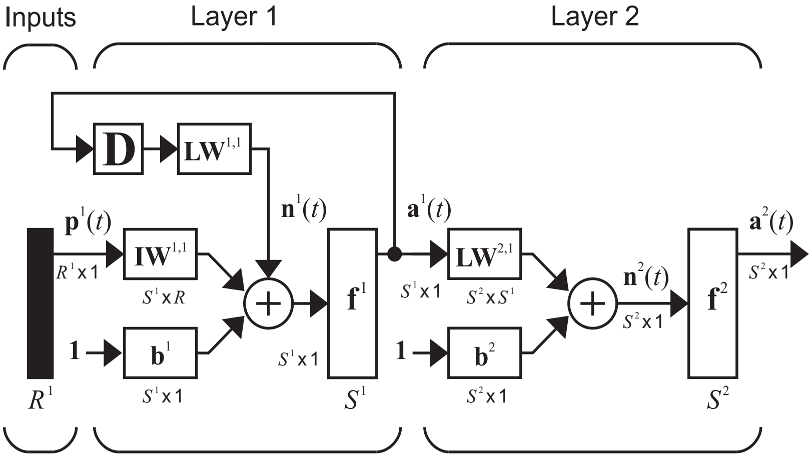


In order for the parallel response (iterated prediction) to be accurate, it is important that the network be trained so that the errors in the series-parallel configuration (one-step-ahead prediction) are very small.

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.

Layer-Recurrent Network (layrecnet)

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 Chapter 3, "Multilayer Networks and Backpropagation Training." The following figure illustrates a two-layer LRN.



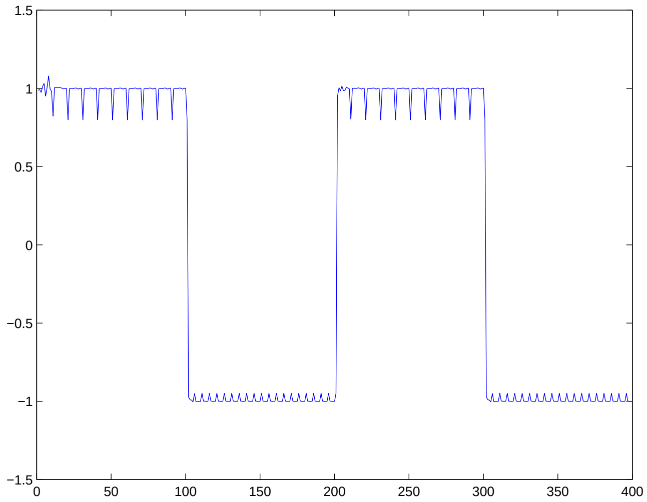
The LRN configurations are used in many filtering and modeling applications discussed already. To demonstrate its operation, the "phoneme" detection problem discussed on page 4-18 is used. Here is the code to load the data and to create and train the network:

```
load phoneme
p = con2seq(y);
t = con2seq(t);
lrn_net = newlrn(p,t,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 following plot demonstrates that the network was able to detect the "phonemes." The response is very similar to the one obtained using the TDNN.



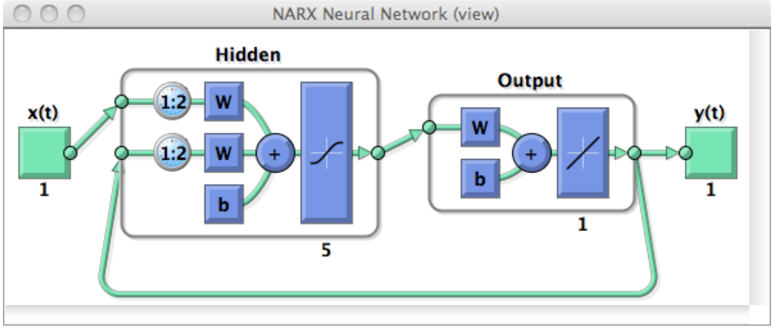
Training Custom Networks

So far, this chapter 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 Chapter 3, “Multilayer Networks and Backpropagation Training,” so long as the components of the network are differentiable. This section will give an example of how to create and train a custom architecture. The custom architecture we will use is the model reference adaptive control (MRAC) system that is described in detail in “Model Reference Control” on page 5-23.

As you can see in “Model Reference Control” 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. We will begin by training a NARX network that will become the plant model subnetwork. For this example, we will use the robot arm to represent the plant, as described in “Model Reference Control” 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;
y = con2seq(y);
u = con2seq(u);
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 following 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.

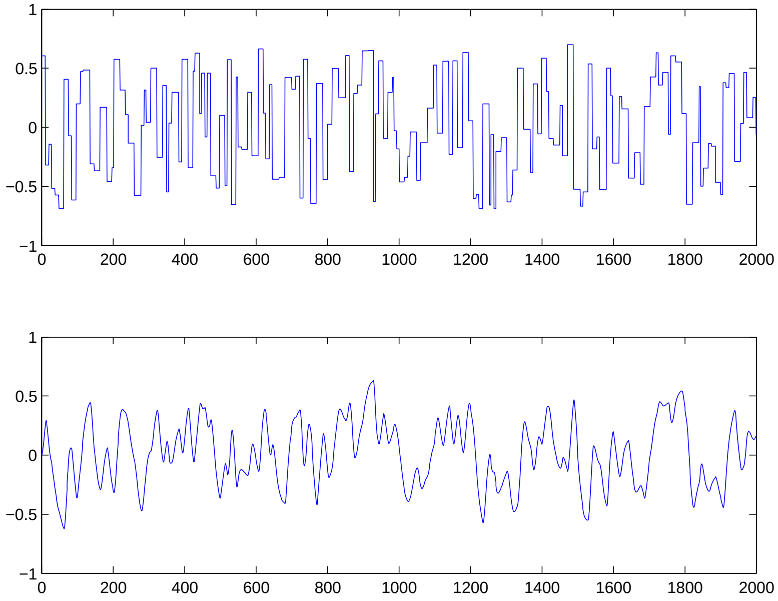
```
mrac_net = feedforwardnet([S1 1 S1]);
mrac_net.layerConnect = [0 1 0 1;1 0 0 0;0 1 0 1;0 0 1 0];
mrac_net.outputs{4}.feedbackMode = 'closed';
mrac_net.layers{2}.transferFcn = 'purelin';
mrac_net.layerWeights{3,4}.delays = 1:2;
mrac_net.layerWeights{3,2}.delays = 1:2;
mrac_net.layerWeights{3,2}.learn = 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, we 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,refin,ind,refout);
refin = con2seq(refin);
refout = con2seq(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 “Using the Model Reference Controller Block” on page 5-25 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.

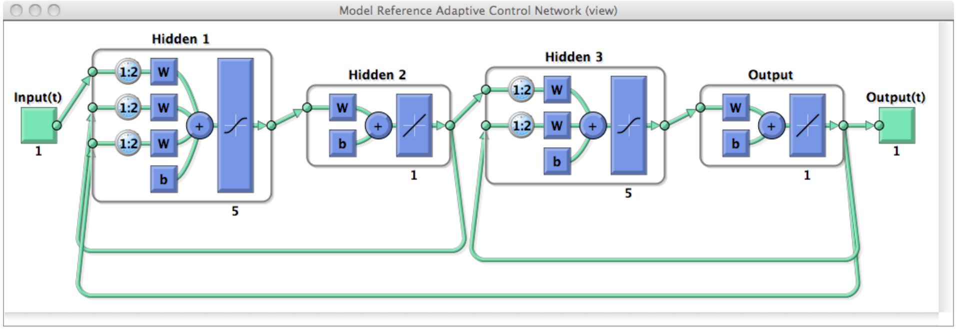
```
mrac_net.LW{2,1} = zeros(size(mrac_net.LW{2,1}));
mrac_net.b{2} = 0;
```

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] = ...
            preparets(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 Chapter 3, “Multilayer Networks and Backpropagation Training,” 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 that 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.

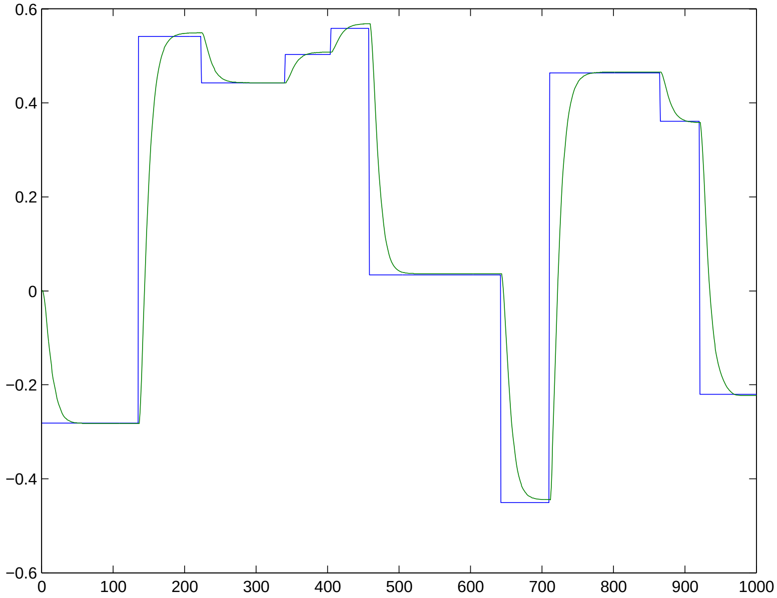
After the network has been trained, you can test the operation by applying a test input to the MRAC network. The following code creates a `skyline` input function, which is a series of steps of random height and width, and applies it to the trained MRAC network.

```
testin = skyline(1000,50,200,-.7,.7);
testinseq = con2seq(testin);
testoutseq = mrac_net(testinseq);
testout = cell2mat(testoutseq);
figure;plot([testin' testout'])
```

From the figure below, 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.


0.6
0.4
0.2
0
−0.2
−0.4
−0.6
0
100
200
300
400
500
600
700
800
900
1000

Multiple Sequences, Time Series Utilities and Error Weighting

There are a number of utility functions available in the toolbox for manipulating time series data sets. This section describes some of these functions, as well as a technique for weighting errors.

Multiple Sequences

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 “Data Structures” on page 2-23.

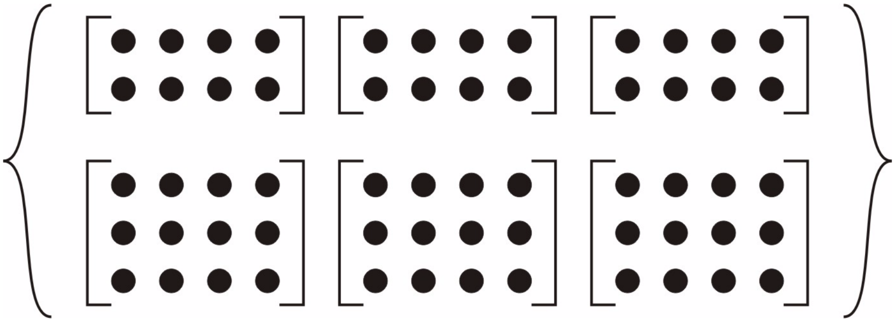
When training a network with a concurrent set of sequences, it is required that each sequence be of the same length. If this is not the case, then the shorter sequence inputs and targets should be padded with NaNs, in order to make all sequences the same length. The targets that are assigned values of 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);
```

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 Name	Operation
`gadd`	Add neural network (nn) data.
`gdivide`	Divide nn data.
`getelements`	Selects indicated elements from nn data.
`getsamples`	Selects indicated samples from nn data.
`getsignals`	Selects indicated signals from nn data.
`gettimesteps`	Selects indicated time steps from nn data.
`gmultiply`	Multiply nn data.
`gnegate`	Take the negative of nn data.

Function Name	Operation
`gsubtract`	Subtract nn data.
`nndata`	Creates an nn data object of specified size, where values are assigned randomly or to a constant.
`nnsize`	Returns number of elements, samples, time steps and signals in an nn data object.
`numelements`	Returns the number of elements in nn data.
`numsamples`	Returns the number of samples in nn data.
`numsignals`	Returns the number of signals in nn data.
`numtimesteps`	Returns the number of time steps in nn data.
`setelements`	Sets specified elements of nn data.
`setsamples`	Sets specified samples of nn data.
`setsignals`	Sets specified signals of nn data.
`settimesteps`	Sets 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 “Time Series Prediction” on page 1-66.

Function Name	Operation
`ploterrcorr`	Plot the autocorrelation function of the error.
`plotinerrcorr`	Plot the crosscorrelation between the error and the input.
`plotresponse`	Plot network output and target versus time.

Error Weighting

In the default mean square error performance function (see Train the Network (p. 3-14)), each squared error contributes the same amount to the performance function, as in

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

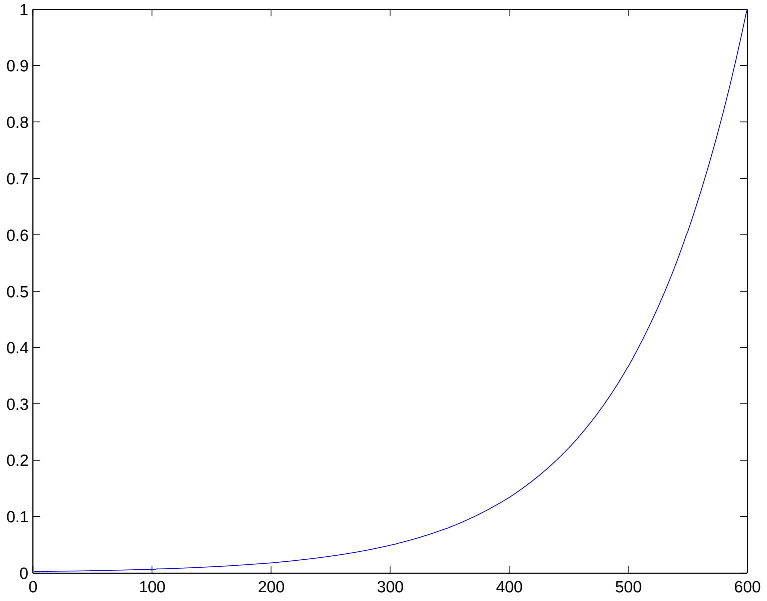
However, the toolbox allows you to weight each squared error individually, as in

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

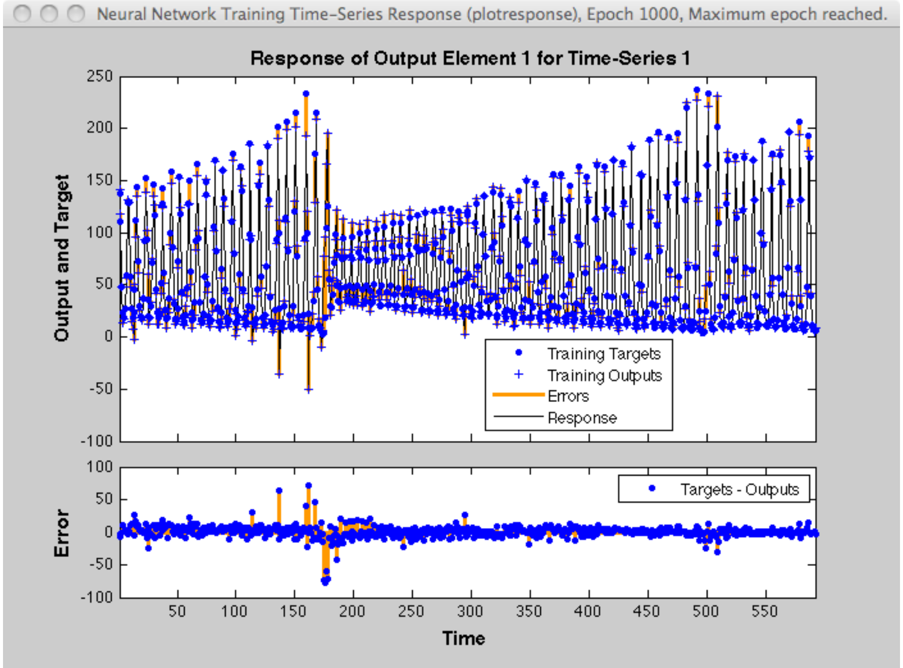
The error weighting object needs to have the same dimensions as the target data. In this way, errors can be weighted according to time step, sample number, signal number or element number. The following is an example of weighting the errors at the end of a time sequence more heavily than errors at the 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(w)
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 following 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 on page 4-15, 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.



Neural Network Training Time-Series Response (plotresponse), Epoch 1000, Maximum epoch reached.

5

Control Systems

Introduction

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 chapter introduces three popular neural network architectures for prediction and control that have been implemented in the Neural Network Toolbox™ software:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

This chapter presents brief descriptions of each of these architectures and demonstrates how you can use them.

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 chapter, 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 of this chapter discuss model predictive control, NARMA-L2 control, and model reference control. Each section consists of a brief description of the control concept, followed by a demonstration 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, using any of the training algorithms discussed in Chapter 3, "Multilayer Networks and Backpropagation Training." (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.

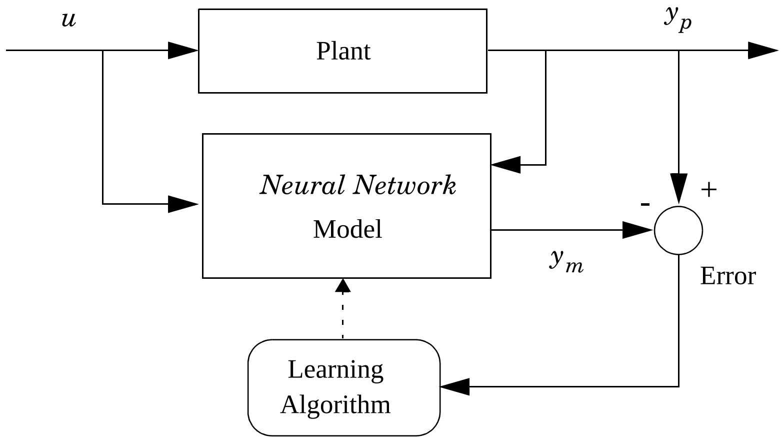
NN Predictive Control

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.)

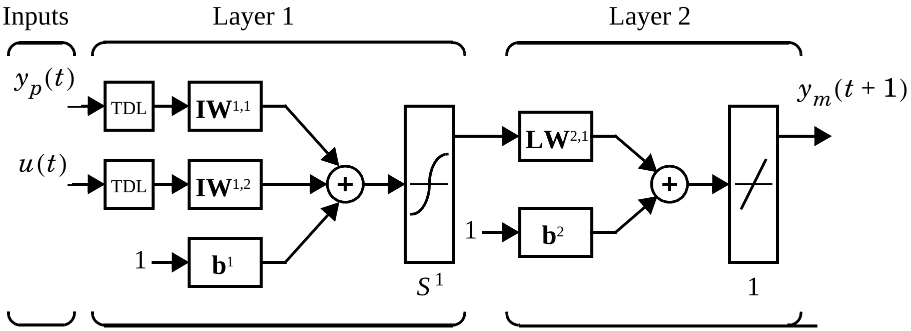
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that is implemented in the Simulink® environment.

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 Chapter 3, "Multilayer Networks and Backpropagation Training," for network training. This process is discussed in more detail later in this chapter.

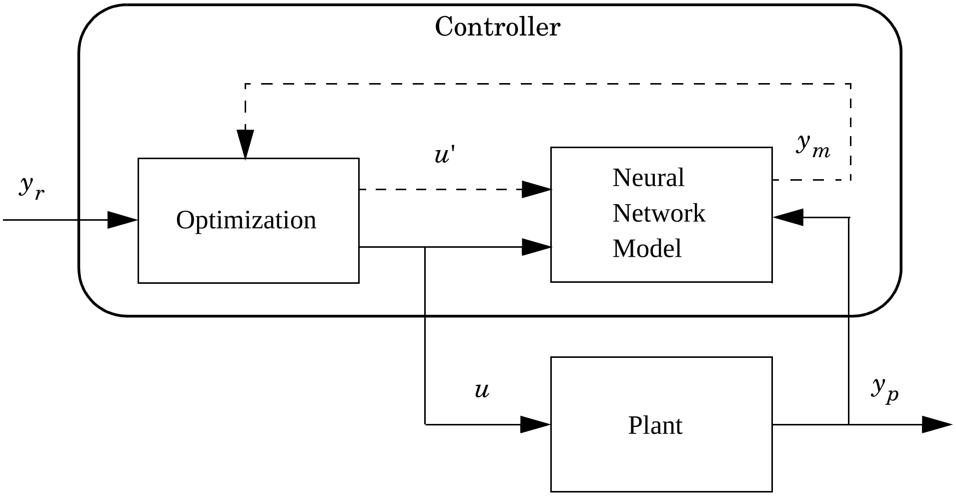
Predictive Control

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 and 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 ρ 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.

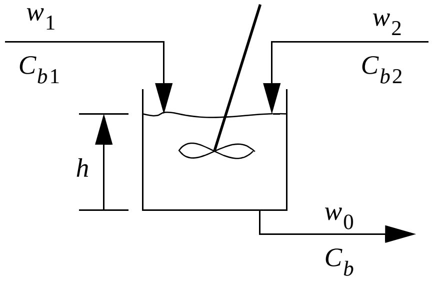


Using the NN Predictive Controller Block

This section demonstrates how the NN Predictive Controller block is used. The first step is to copy the NN Predictive Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox software to demonstrate the predictive controller. This demo 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



$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

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 demonstration, set $w_2(t)$ = 0.1. The level of the tank $h(t)$ is not controlled for this experiment.

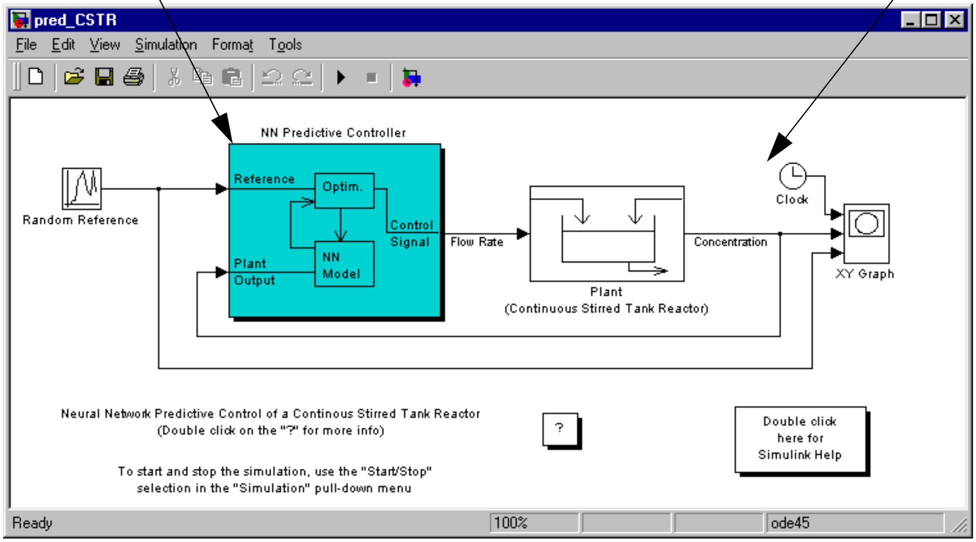
To run this demo, follow these steps:

1 Start MATLAB®.

2 Run the demo model by typing `predcstr` in the MATLAB Command Window. This command starts Simulink and creates the following model window. The NN Predictive Controller block is already in the model.

This NN Predictive Controller block was copied from the Neural Network Toolbox blockset to this model window. The **Control Signal** was connected to the input of the plant model. The output of the plant model was connected to **Plant Output**. The reference signal was connected to **Reference**.

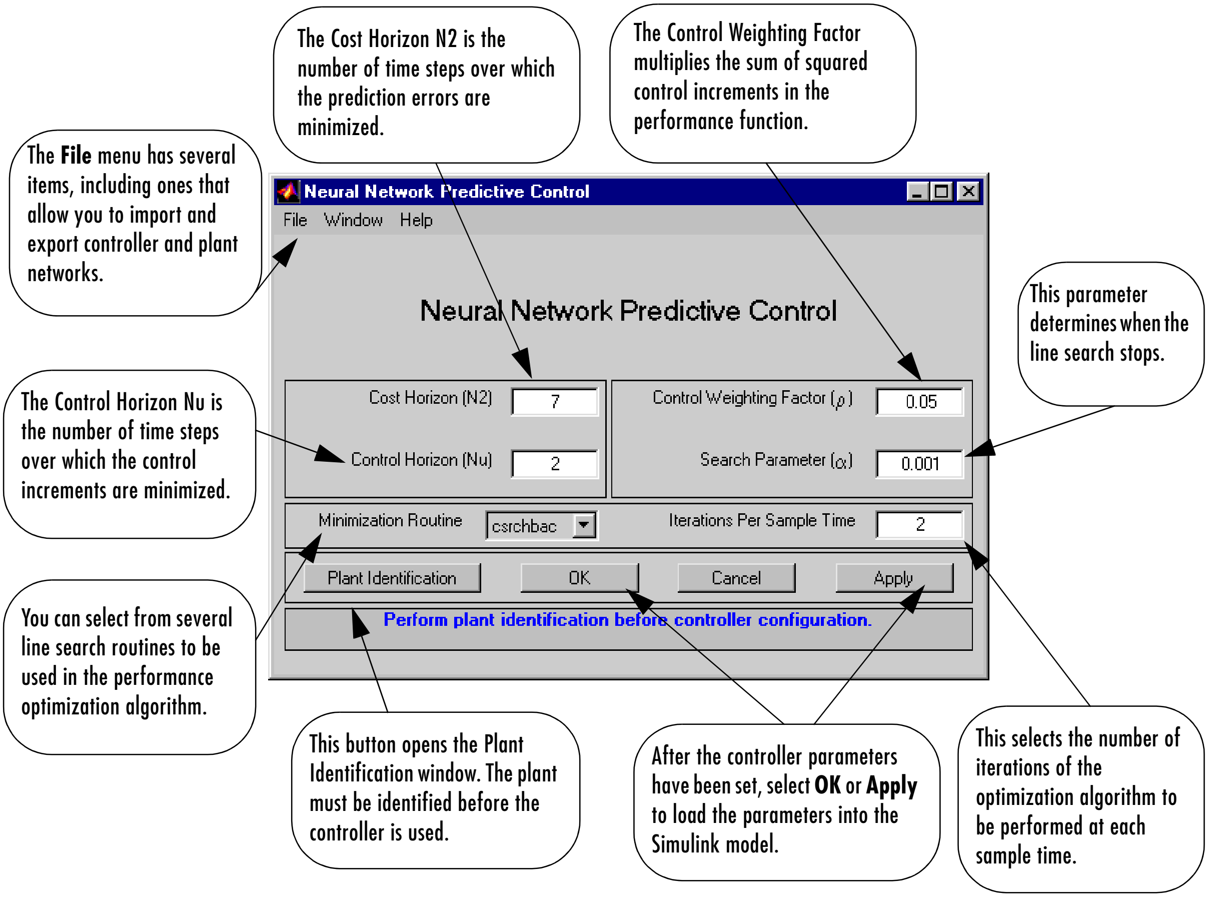
This block contains the Simulink CSTR plant model.



Neural Network Predictive Control of a Continous Stirred Tank Reactor
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

3 Double-click the NN Predictive Controller block. This brings up 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 ρ, described earlier, is also defined in this window. The parameter α 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 Chapter 3, “Multilayer Networks and Backpropagation Training.”



Neural Network Predictive Control

Neural Network Predictive Control

Cost Horizon (N2) 7

Control Horizon (Nu) 2

Control Weighting Factor (ρ) 0.05

Search Parameter (α) 0.001

Minimization Routine csrchbac

Iterations Per Sample Time 2

Perform plant identification before controller configuration.

The Cost Horizon N2 is the number of time steps over which the prediction errors are minimized.

The Control Weighting Factor multiplies the sum of squared control increments in the performance function.

The **File** menu has several items, including ones that allow you to import and export controller and plant networks.

This parameter determines when the line search stops.

The Control Horizon Nu is the number of time steps over which the control increments are minimized.

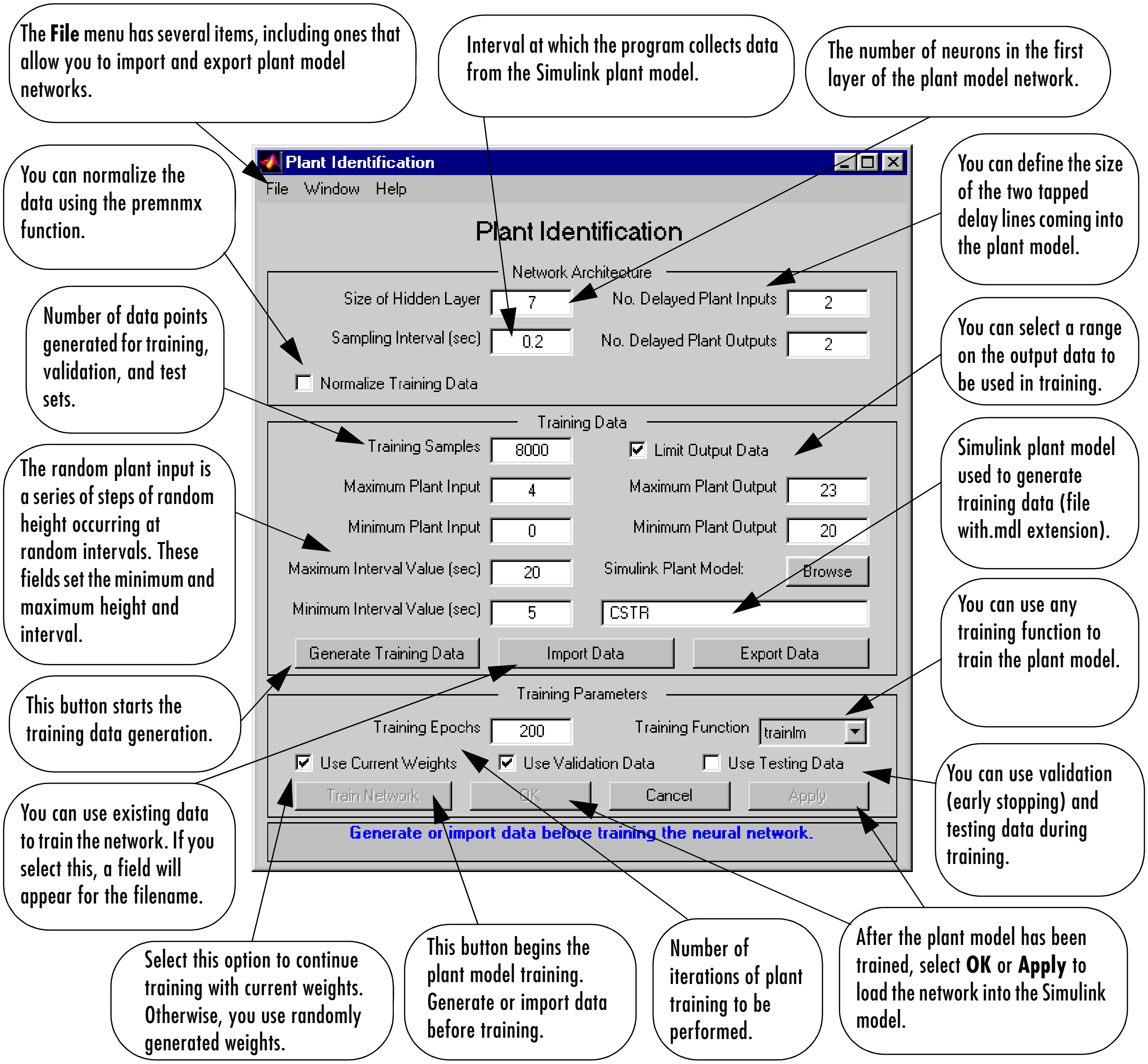
You can select from several line search routines to be used in the performance optimization algorithm.

This button opens the Plant Identification window. The plant must be identified before the controller is used.

After the controller parameters have been set, select **OK** or **Apply** to load the parameters into the Simulink model.

This selects the number of iterations of the optimization algorithm to be performed at each sample time.

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 Chapter 3, “Multilayer Networks and Backpropagation Training,” to train the neural network plant model.



Plant Identification

Network Architecture

Size of Hidden Layer 7

No. Delayed Plant Inputs 2

Sampling Interval (sec) 0.2

No. Delayed Plant Outputs 2

□ Normalize Training Data

Training Data

Training Samples 8000

☑ Limit Output Data

Maximum Plant Input 4

Maximum Plant Output 23

Minimum Plant Input 0

Minimum Plant Output 20

Maximum Interval Value (sec) 20

Simulink Plant Model: CSTR

Minimum Interval Value (sec) 5

Training Parameters

Training Epochs 200

Training Function trainlm

☑ Use Current Weights

☑ Use Validation Data

□ Use Testing Data

Generate or import data before training the neural network.

The **File** menu has several items, including ones that allow you to import and export plant model networks.

Interval at which the program collects data from the Simulink plant model.

The number of neurons in the first layer of the plant model network.

You can normalize the data using the premnmx function.

You can define the size of the two tapped delay lines coming into the plant model.

Number of data points generated for training, validation, and test sets.

You can select a range on the output data to be used in training.

The random plant input is a series of steps of random height occurring at random intervals. These fields set the minimum and maximum height and interval.

Simulink plant model used to generate training data (file with.mdl extension).

You can use any training function to train the plant model.

This button starts the training data generation.

You can use validation (early stopping) and testing data during training.

You can use existing data to train the network. If you select this, a field will appear for the filename.

Select this option to continue training with current weights. Otherwise, you use randomly generated weights.

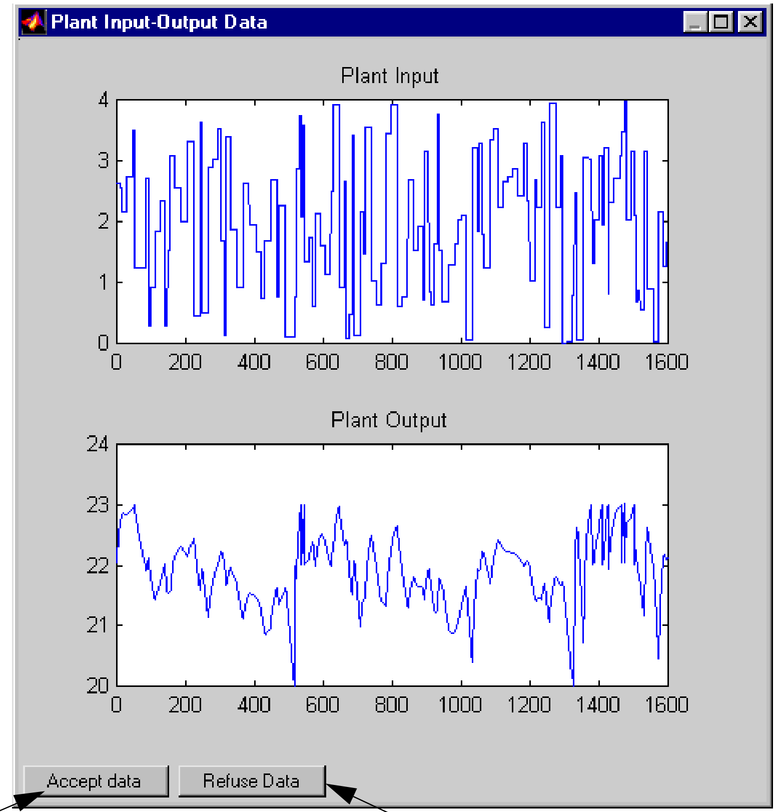
This button begins the plant model training. Generate or import data before training.

Number of iterations of plant training to be performed.

After the plant model has been trained, select **OK** or **Apply** to load the network into the Simulink model.

5 Select the **Generate Training Data** button. The program generates training data by applying a series of random step inputs to the Simulink plant model.

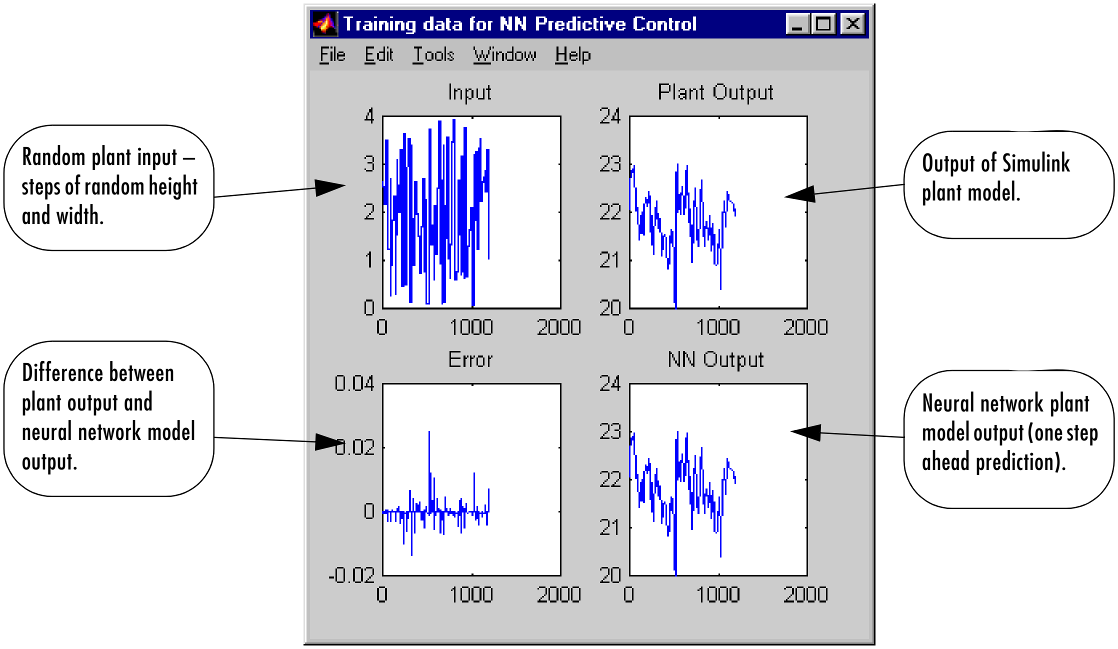
The potential training data is then displayed in a figure similar to the following.



Accept the data if it is sufficiently representative of future plant activity. Then plant training begins.

If you refuse the training data, you return to the Plant Identification window and restart the training.

6 Select **Accept Data**, and then select **Train Network** from 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 Chapter 3, “Multilayer Networks and Backpropagation Training.” 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.)



Random plant input – steps of random height and width.

Output of Simulink plant model.

Difference between plant output and neural network model output.

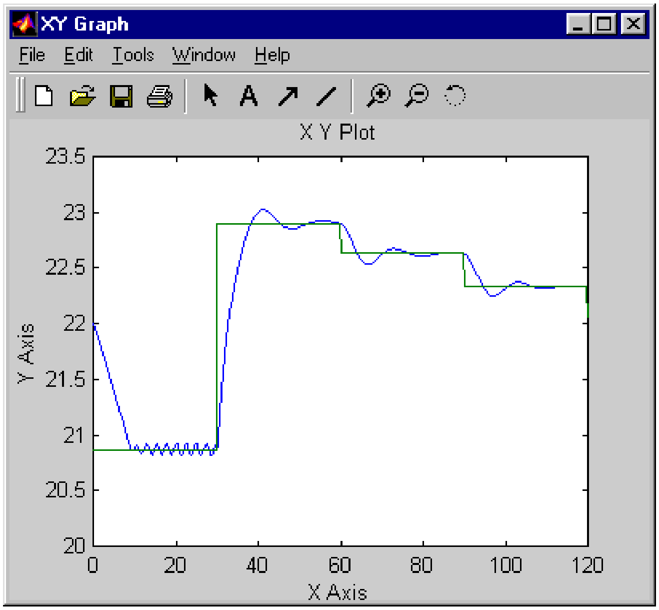
Neural network plant model output (one step ahead prediction).

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 demonstration, 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 model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File
Edit
Tools
Window
Help
X Y Plot
23.5
23
22.5
22
21.5
21
20.5
20
Y Axis
0
20
40
60
80
100
120
X Axis

NARMA-L2 (Feedback Linearization) Control

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 demonstrating 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 a demonstration 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:

$$y(k+d) = N[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)]$$

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 $y(k + d) = y_r(k + d)$, the next step is to develop a nonlinear controller of the form

$$u(k) = G[y(k), y(k-1), \ldots, y(k-n+1), y_r(k+d), u(k-1), \ldots, u(k-m+1)]$$

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:

$$\begin{aligned}\hat{y}(k+d) &= f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \\ &\quad + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \cdot u(k)\end{aligned}$$

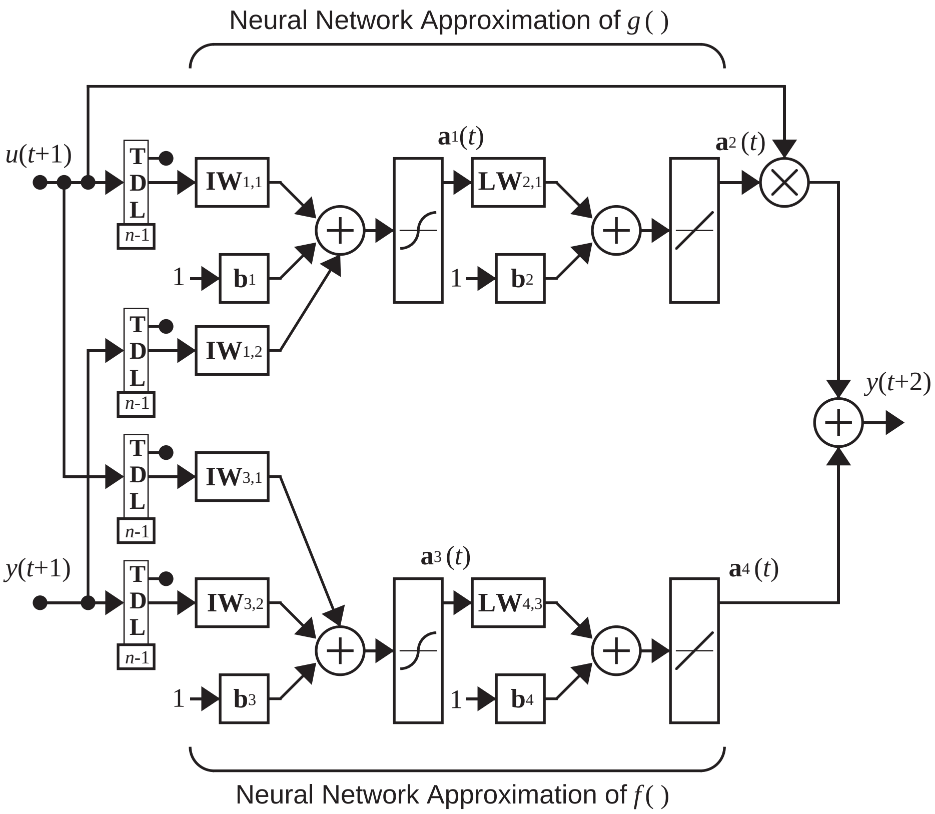
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k + d) = y_r(k + d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}$$

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

$$\begin{aligned}y(k+d) &= f[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)] \\ &\quad + g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)] \cdot u(k+1)\end{aligned}$$

where $d \geq 2$. The following figure shows the structure of a neural network representation.

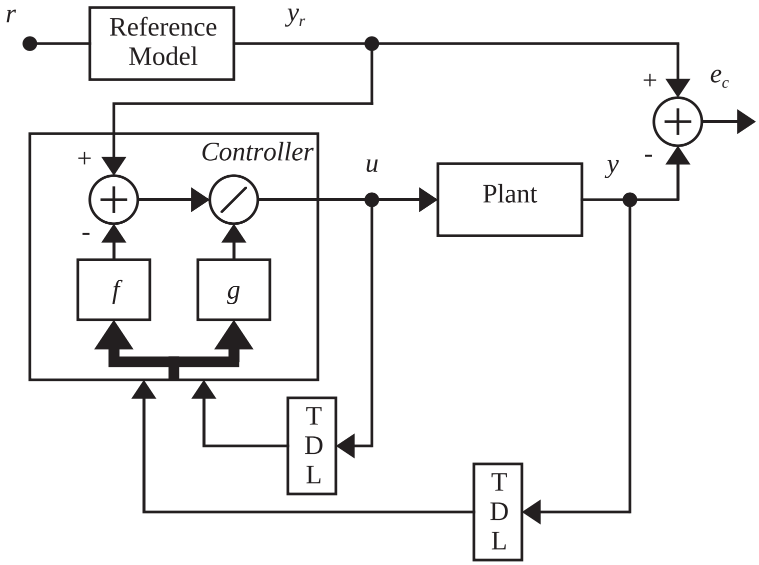


NARMA-L2 Controller

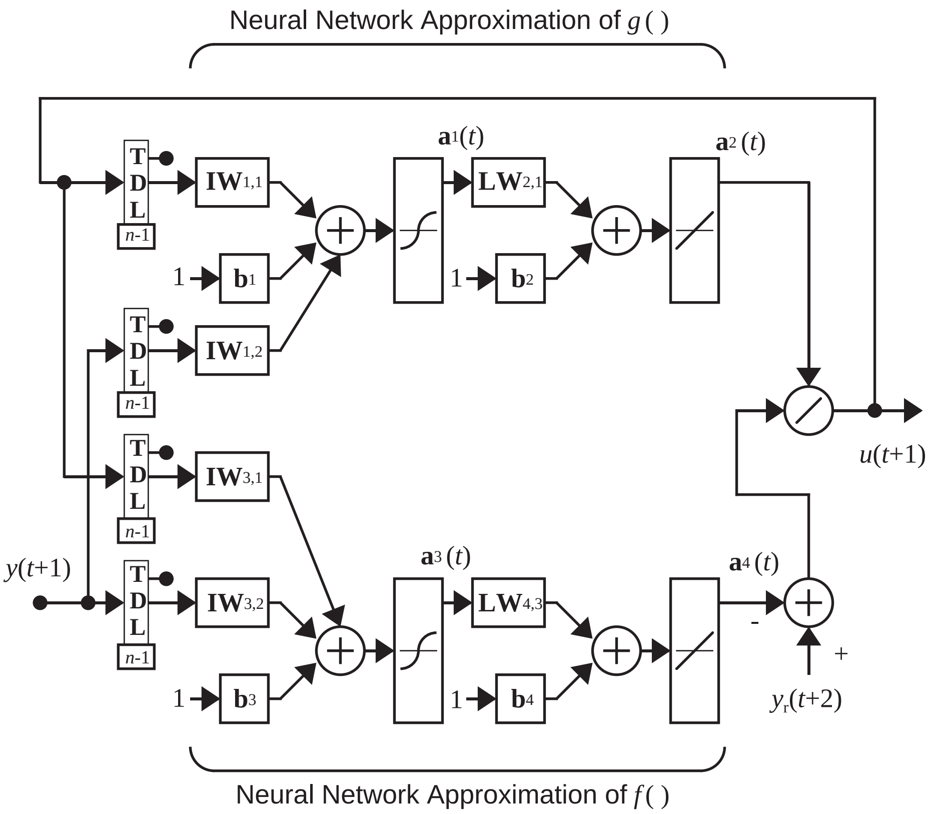
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$

which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller.



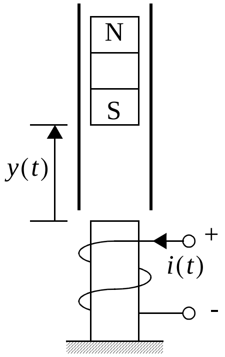
This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.



Using the NARMA-L2 Controller Block

This section demonstrates how the NARMA-L2 controller is trained. The first step is to copy the NARMA-L2 Controller block from the Neural Network Toolbox blockset to your model window. See your Simulink® documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox software to demonstrate the NARMA-L2 controller. In this demo, 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

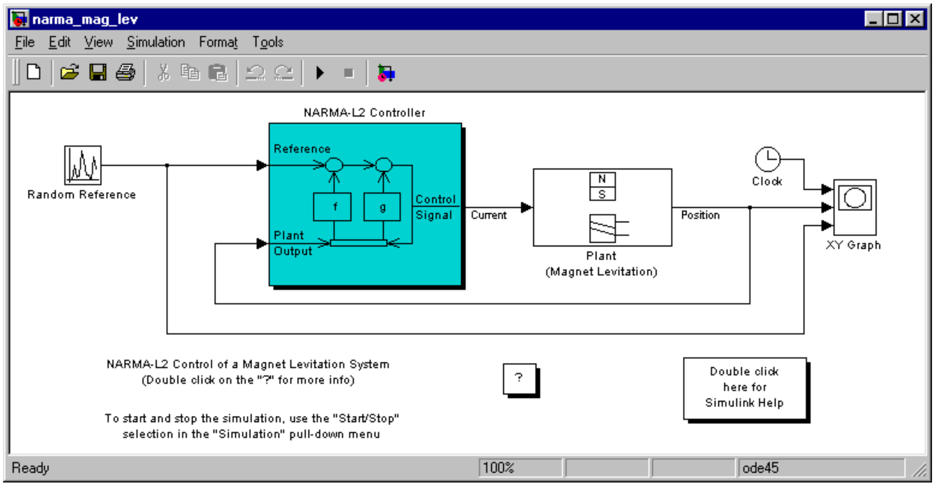
$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt}$$

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 β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α 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 demo, follow these steps:

1 Start MATLAB®.

2 Run the demo model by typing `narmamaglev` in the MATLAB Command Window. This command starts Simulink and creates the following model window. The NARMA-L2 Control block is already in the model.

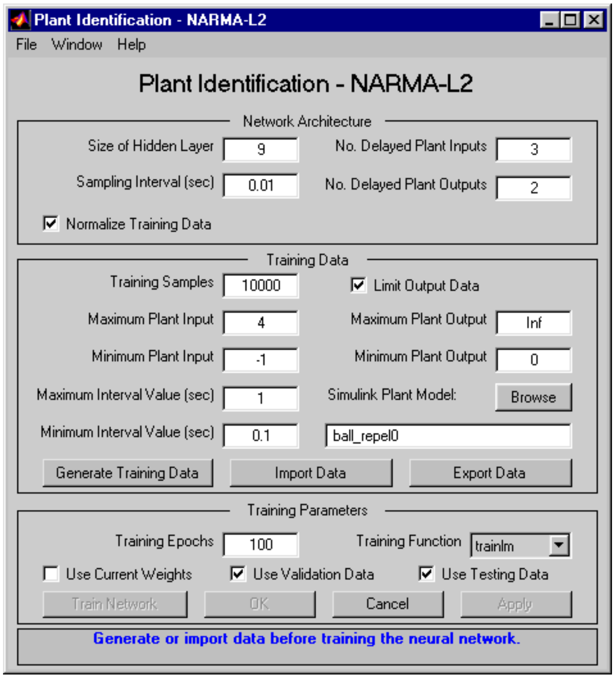


NARMA-L2 Control of a Magnet Levitation System
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

Double click here for Simulink Help

3 Double-click the NARMA-L2 Controller block. This brings up 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.



Plant Identification - NARMA-L2

Network Architecture

Size of Hidden Layer: 9
No. Delayed Plant Inputs: 3
Sampling Interval (sec): 0.01
No. Delayed Plant Outputs 2
☑ Normalize Training Data

Training Data

Training Samples: 10000
☑ Limit Output Data
Maximum Plant Input: 4
Maximum Plant Output: Inf
Minimum Plant Input: -1
Minimum Plant Output: 0
Maximum Interval Value (sec): 1
Simulink Plant Model: ball_repel0
Minimum Interval Value (sec): 0.1

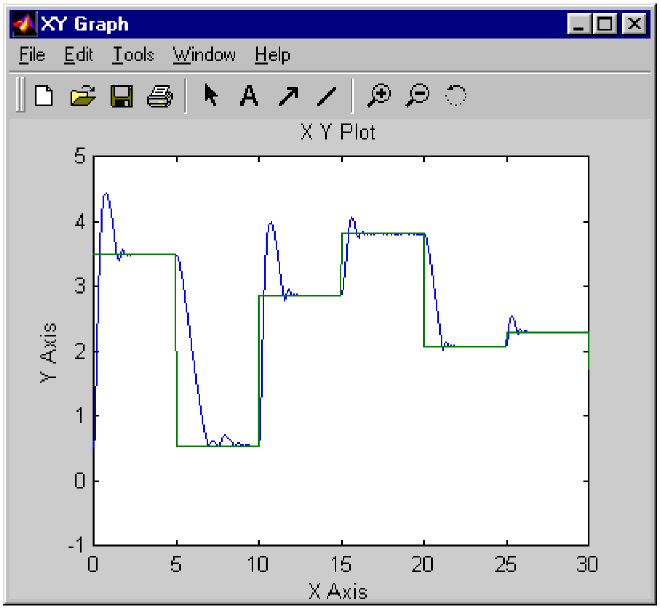
Training Parameters

Training Epochs: 100
Training Function: trainlm
☐ Use Current Weights
☑ Use Validation Data
☑ Use Testing Data

Generate or import data before training the neural network.

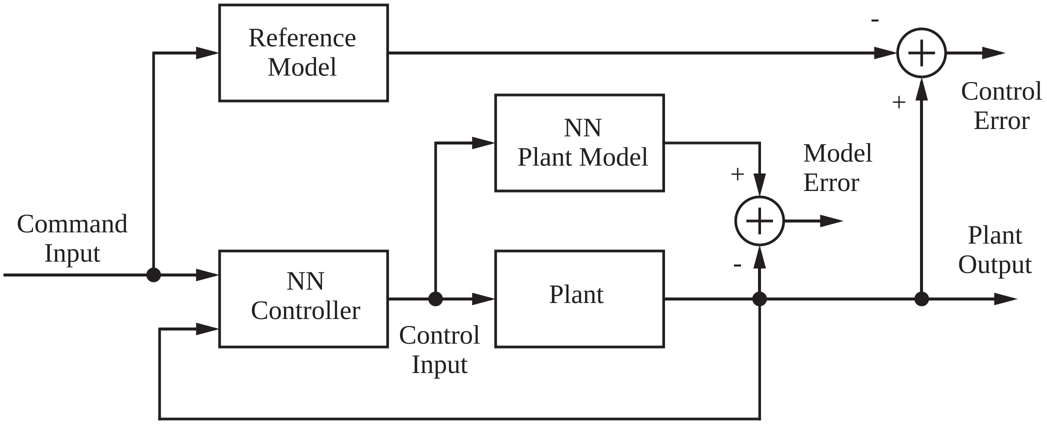
4 This window works the same as the other Plant Identification windows, so the training process is not repeated. Instead, simulate the NARMA-L2 controller.

5 Return to the Simulink model and start the simulation by choosing the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
5
4
3
2
1
0
-1
Y Axis
0
5
10
15
20
25
30
X Axis

Model Reference Control

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.



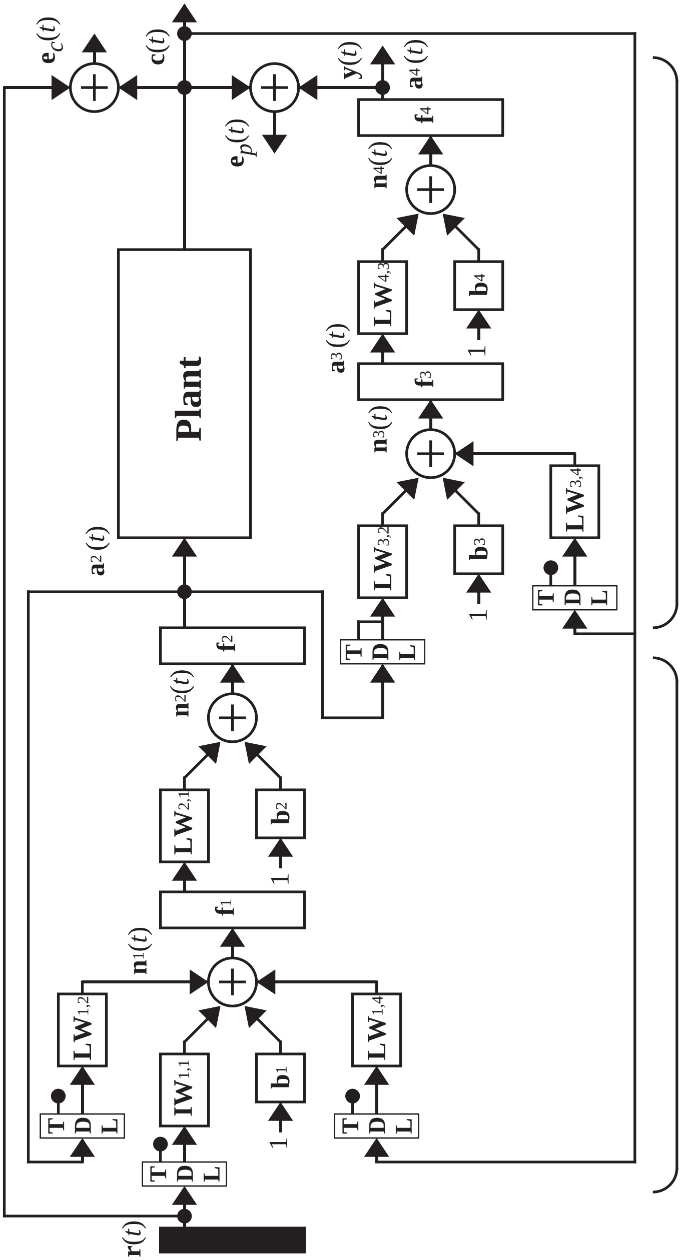
The figure on the following page shows the details of the neural network plant model and the neural network controller as they are implemented in the Neural Network Toolbox™ software. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets 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 demonstrates how you can set the parameters.

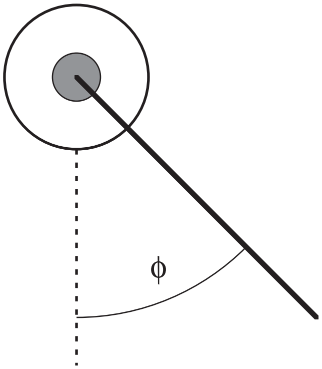


r(t)
T D L
IW1,1
LW1,2
LW1,4
b1
1
n1(t)
f1
LW2,1
b2
n2(t)
f2
a2(t)
Plant
c(t)
ec(t)
ep(t)
LW3,2
LW3,4
b3
n3(t)
f3
a3(t)
LW4,3
b4
n4(t)
f4
a4(t)
y(t)

Using the Model Reference Controller Block

This section demonstrates 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 your model window. See your Simulink® documentation if you are not sure how to do this. This step is skipped in the following demonstration.

A demo model is provided with the Neural Network Toolbox software to demonstrate the model reference controller. In this demo, 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 ϕ 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

$$\frac{d^2y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

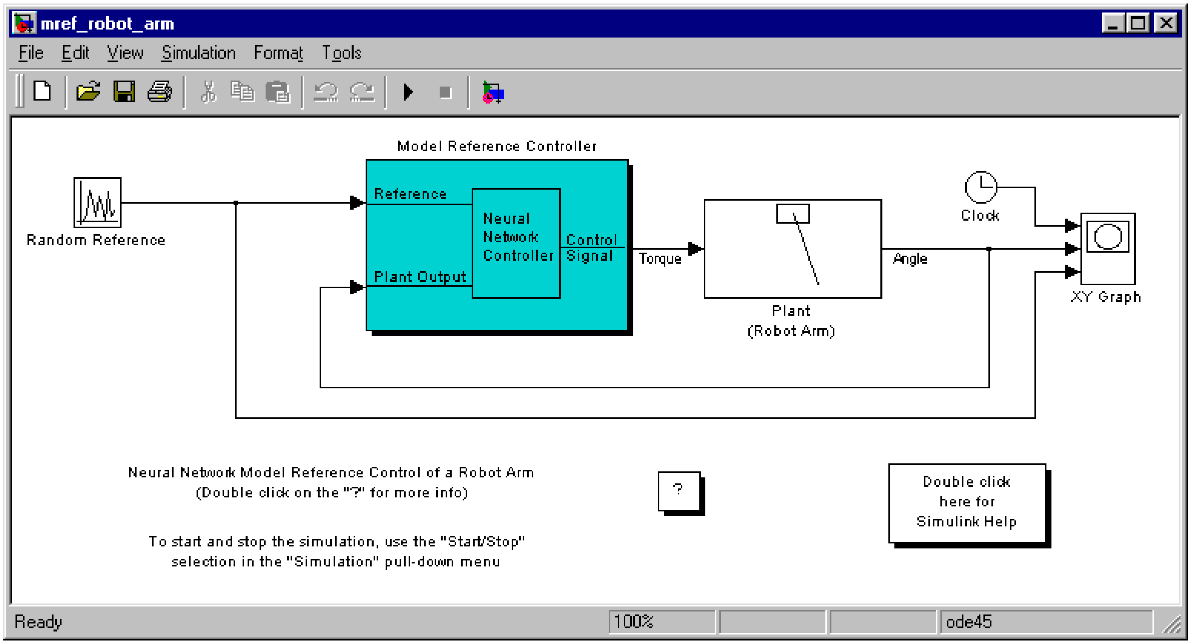
where y_r is the output of the reference model, and r is the input reference signal.

This demo 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 demo, follow these steps.

1 Start MATLAB®.

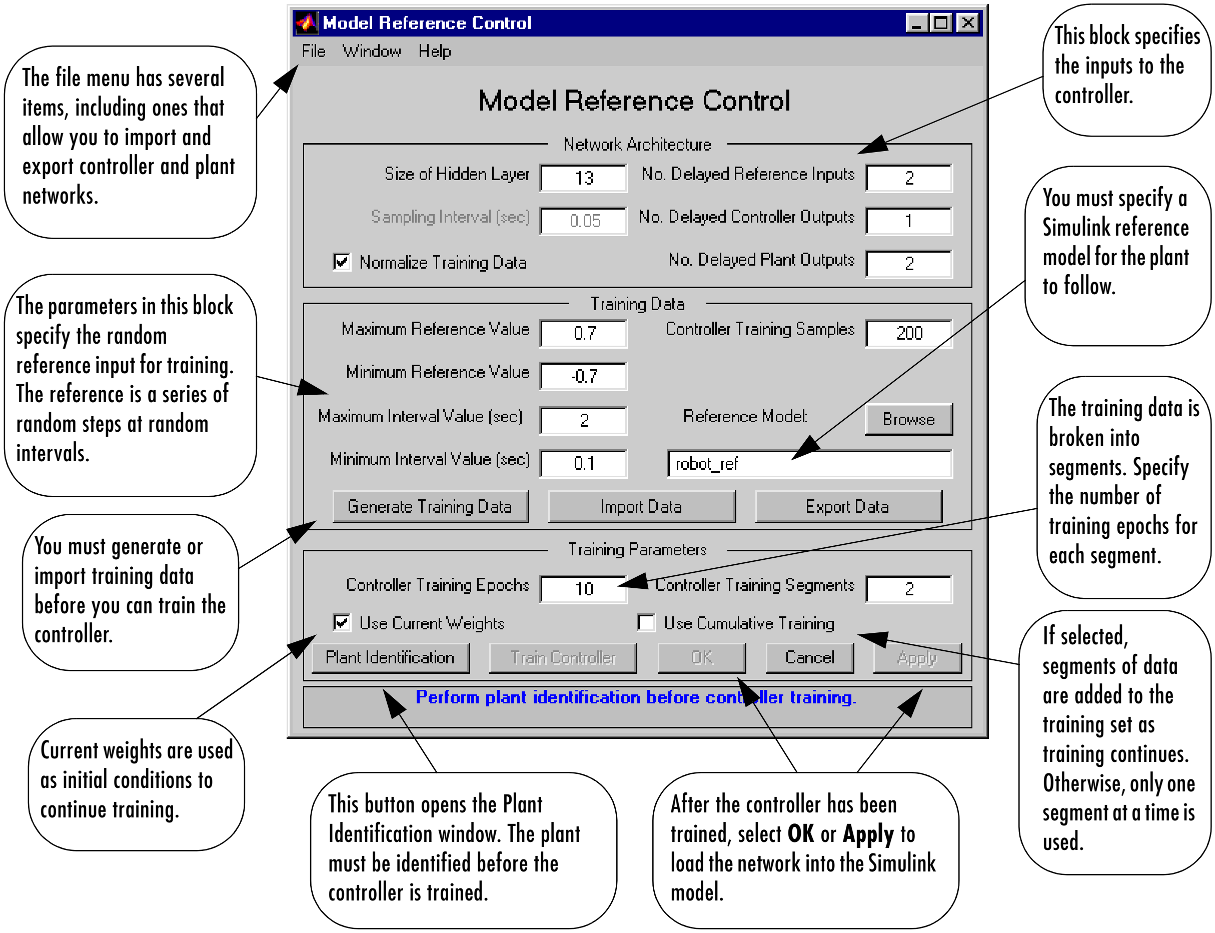
2 Run the demo model by typing `mrefrobotarm` in the MATLAB Command Window. This command starts Simulink and creates the following model window. The Model Reference Control block is already in the model.


mref_robot_arm
File Edit View Simulation Format Tools
Model Reference Controller
Reference
Neural Network Controller
Control Signal
Plant Output
Random Reference
Torque
Plant (Robot Arm)
Angle
Clock
XY Graph
?
Double click here for Simulink Help
Ready
100%
ode45

Neural Network Model Reference Control of a Robot Arm
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

3 Double-click the Model Reference Control block. This brings up the following window for training the model reference controller.



Model Reference Control

File Window Help

Model Reference Control

Network Architecture

Size of Hidden Layer: 13

No. Delayed Reference Inputs: 2

Sampling Interval (sec): 0.05

No. Delayed Controller Outputs: 1

☑ Normalize Training Data

No. Delayed Plant Outputs: 2

Training Data

Maximum Reference Value: 0.7

Controller Training Samples: 200

Minimum Reference Value: -0.7

Maximum Interval Value (sec): 2

Reference Model: robot_ref

Minimum Interval Value (sec): 0.1

Training Parameters

Controller Training Epochs: 10

Controller Training Segments: 2

☑ Use Current Weights

☐ Use Cumulative Training

Perform plant identification before controller training.

The file menu has several items, including ones that allow you to import and export controller and plant networks.

This block specifies the inputs to the controller.

You must specify a Simulink reference model for the plant to follow.

The parameters in this block specify the random reference input for training. The reference is a series of random steps at random intervals.

The training data is broken into segments. Specify the number of training epochs for each segment.

You must generate or import training data before you can train the controller.

If selected, segments of data are added to the training set as training continues. Otherwise, only one segment at a time is used.

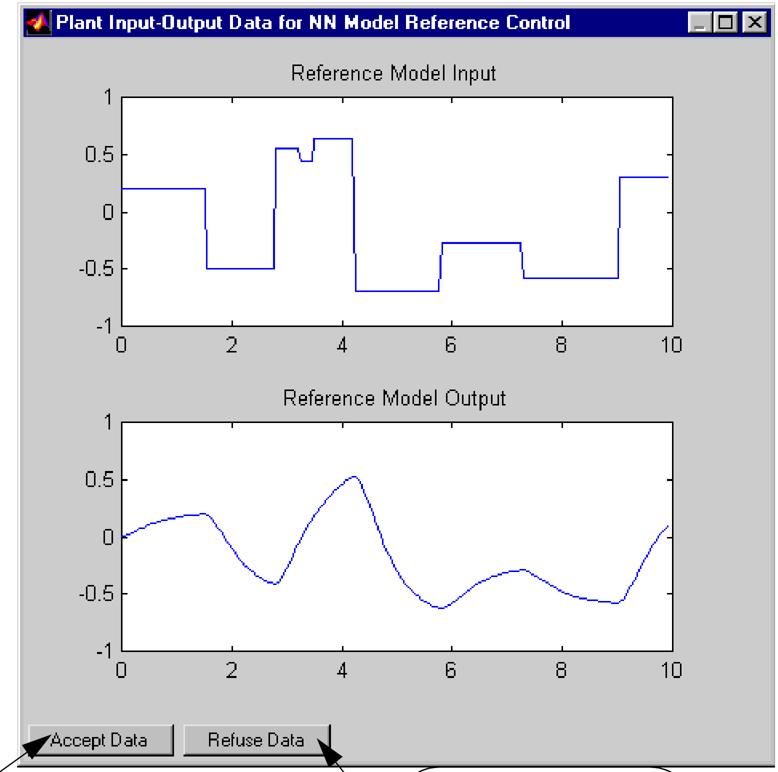
Current weights are used as initial conditions to continue training.

This button opens the Plant Identification window. The plant must be identified before the controller is trained.

After the controller has been trained, select **OK** or **Apply** to load the network into the Simulink model.

4 The next step would normally be to select **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.

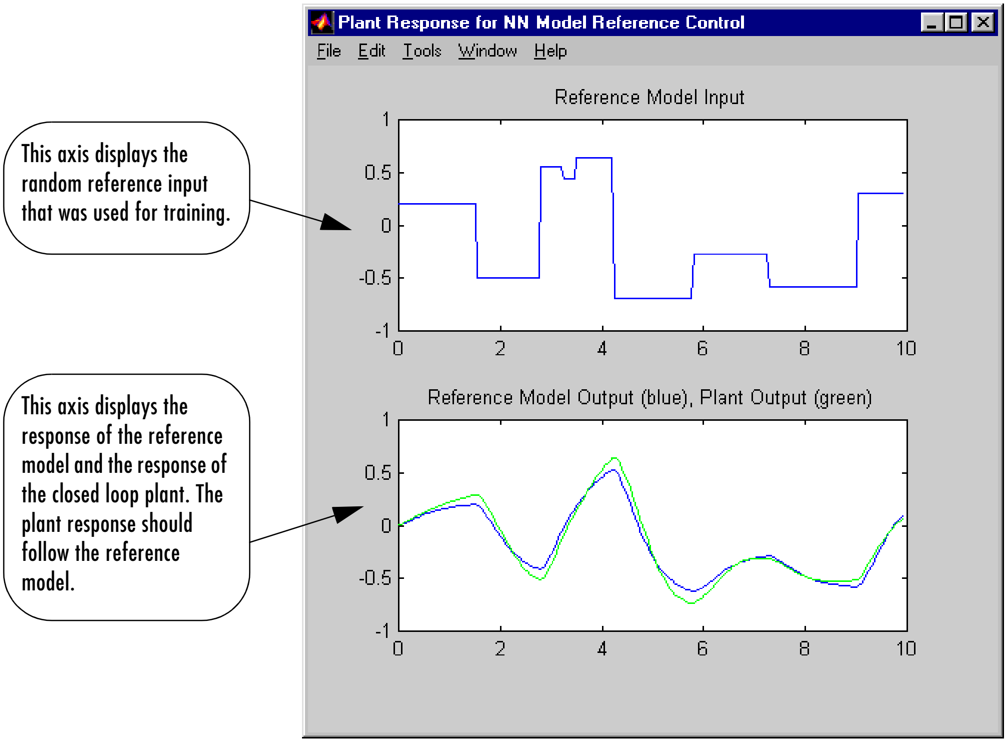
5 Select **Generate Data**. The program starts generating the data for training the controller. After the data is generated, the following window appears.



Select this if the training data shows enough variation to adequately train the controller.

If the data is not adequate, select this button and then go back to the controller window and select **Generate Data** again.

6 Select **Accept Data**. Return to the **Model Reference Control** window and select **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.

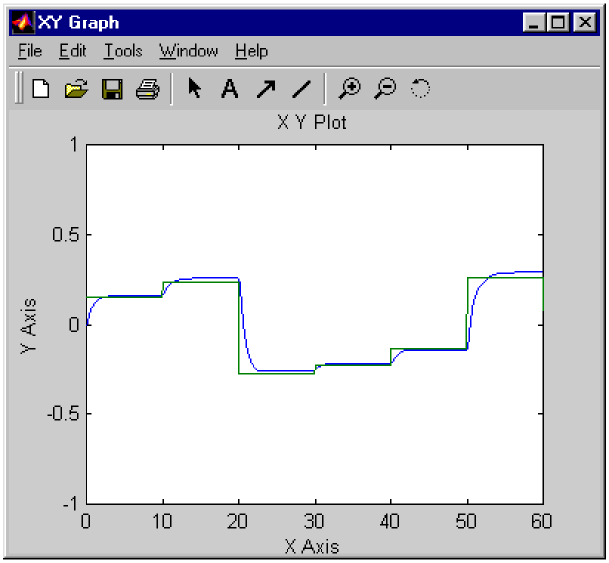

Plant Response for NN Model Reference Control
File Edit Tools Window Help
Reference Model Input
1
0.5
0
-0.5
-1
0 2 4 6 8 10
Reference Model Output (blue), Plant Output (green)
1
0.5
0
-0.5
-1
0 2 4 6 8 10

This axis displays the random reference input that was used for training.

This axis displays the response of the reference model and the response of the closed loop plant. The plant response should follow the reference model.

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 demonstration, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.

8 Return to the Simulink model and start the simulation by selecting the **Start** command from the **Simulation** menu. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File
Edit
Tools
Window
Help
X Y Plot
1
0.5
0
-0.5
-1
Y Axis
0
10
20
30
40
50
60
X Axis

Importing and Exporting

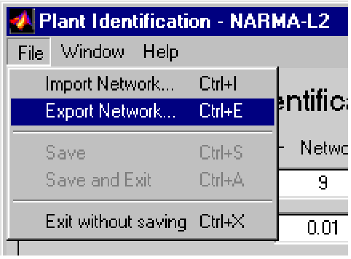
You can save networks and training data to the workspace or to a disk file. The following two sections demonstrate how you can do this.

Importing and Exporting Networks

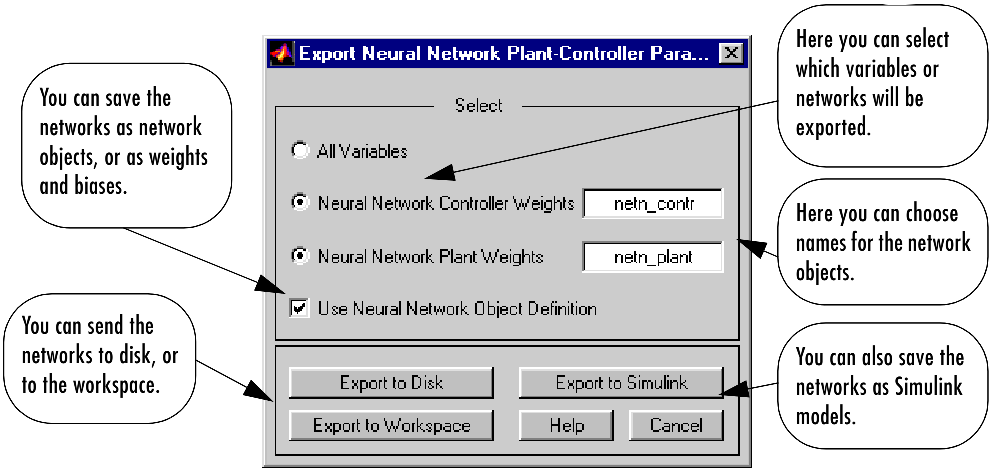
The controller and plant model networks that you develop are stored within Simulink® controller blocks. At some point you might want to transfer the networks into other applications, or you might want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following demonstration leads you through the export and import processes. (The NARMA-L2 window is used for this demonstration, but the same procedure applies to all the controllers.)

1 Repeat the first three steps of the NARMA-L2 demonstration "Using the NARMA-L2 Controller Block" on page 5-18. The **NARMA-L2 Plant Identification** window should then be open.

2 Select **Export** from the **File** menu, as shown below.



This causes the following window to open.



Export Neural Network Plant-Controller Para...

Select

○ All Variables

● Neural Network Controller Weights: netn_contr

● Neural Network Plant Weights: netn_plant

☑ Use Neural Network Object Definition

You can save the networks as network objects, or as weights and biases.

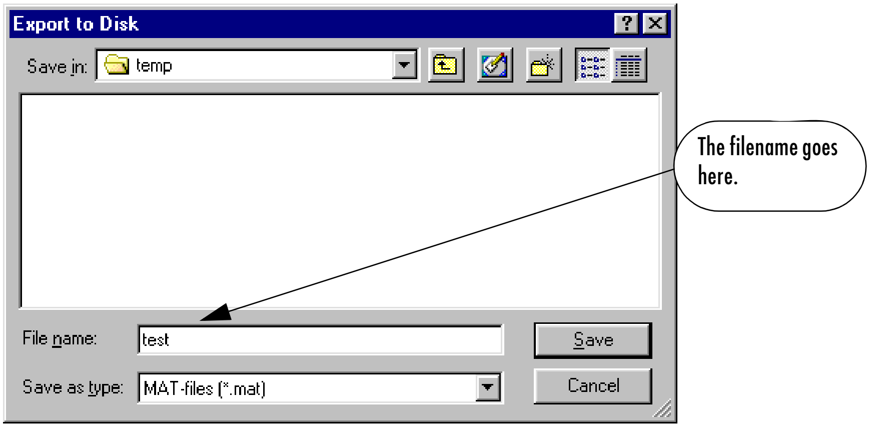
You can send the networks to disk, or to the workspace.

Here you can select which variables or networks will be exported.

Here you can choose names for the network objects.

You can also save the networks as Simulink models.

3 Select **Export to Disk**. The following window opens. Enter the filename `test` in the box, and select **Save**. This saves the controller and plant networks to disk.



Export to Disk

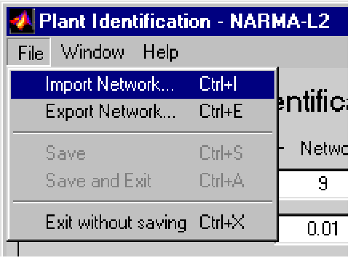
Save in: temp

File name: test

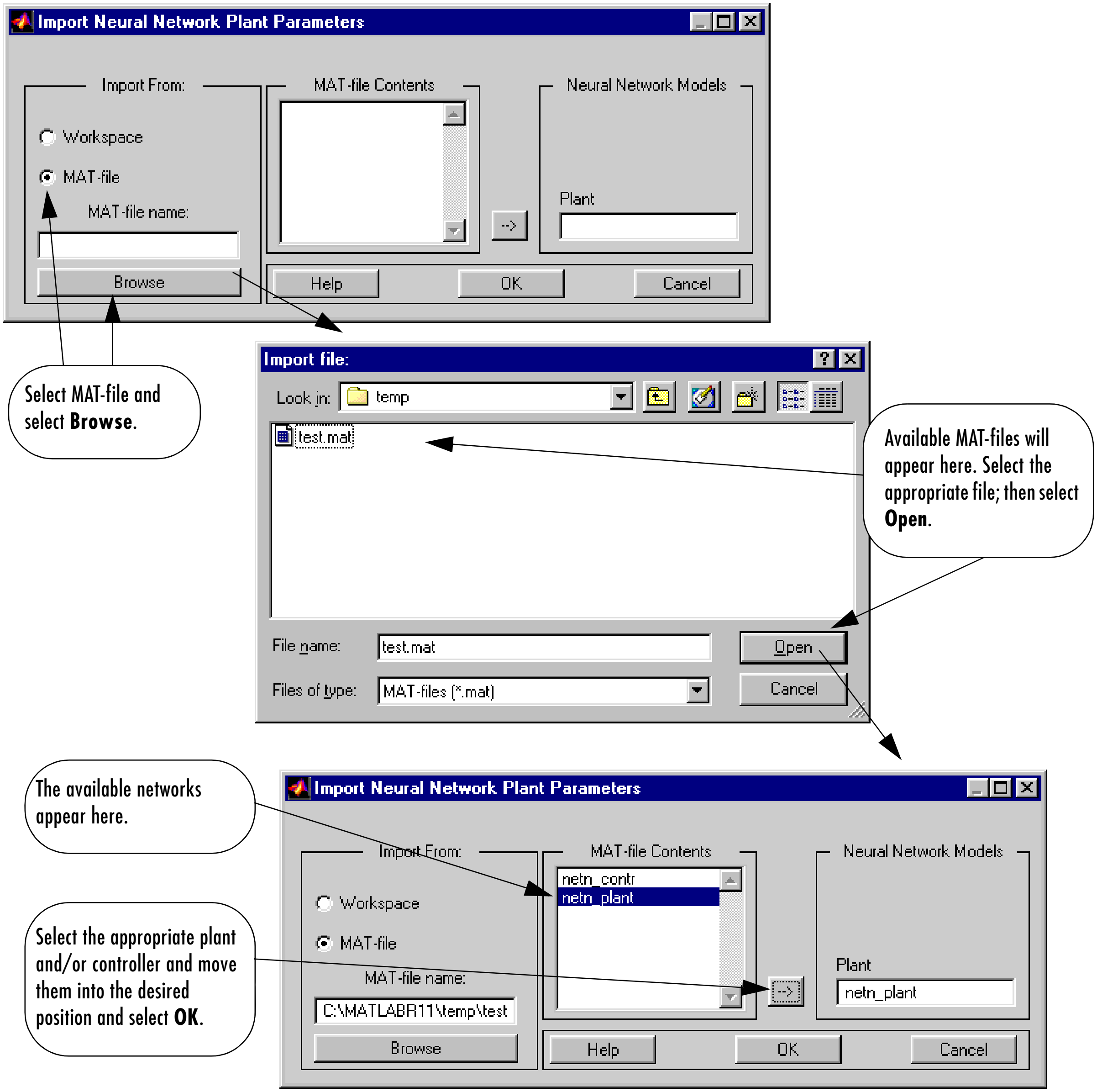
Save as type: MAT-files (*.mat)

The filename goes here.

4 Retrieve that data with the **Import** menu option. Select **Import Network** from the **File** menu, 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 selecting **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 don't need to import both networks.



Select MAT-file and select **Browse**.

Available MAT-files will appear here. Select the appropriate file; then select **Open**.

The available networks appear here.

Select the appropriate plant and/or controller and move them into the desired position and select **OK**.

Import file:

Look in: temp

File name: test.mat

Files of type: MAT-files (*.mat)

MAT-file name: C:\MATLABR11\temp\test

MAT-file Contents: netn_contr, netn_plant

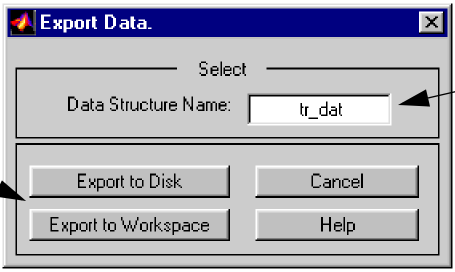
Plant: netn_plant

Importing and Exporting 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 demonstration leads you through the import and export processes. (The NN Predictive Control window is used for this demonstration, but the same procedure applies to all the controllers.)

1 Repeat the first five steps of the NN Predictive Control demonstration "Using the NN 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.

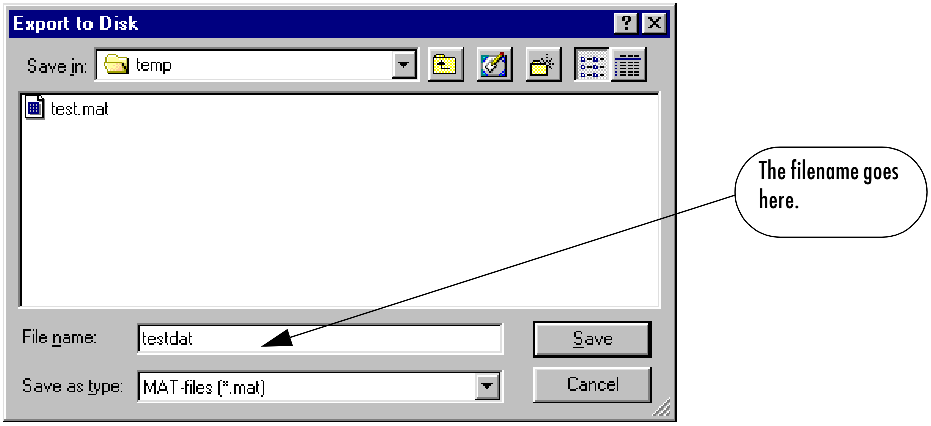
2 Select the **Export** button. This causes the following window to open.



You can export the data to the workspace or to a disk file.

You can select a name for the data structure. The structure contains at least two fields: name.U, and name.Y. These two fields contain the input and output arrays.

3 Select **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.



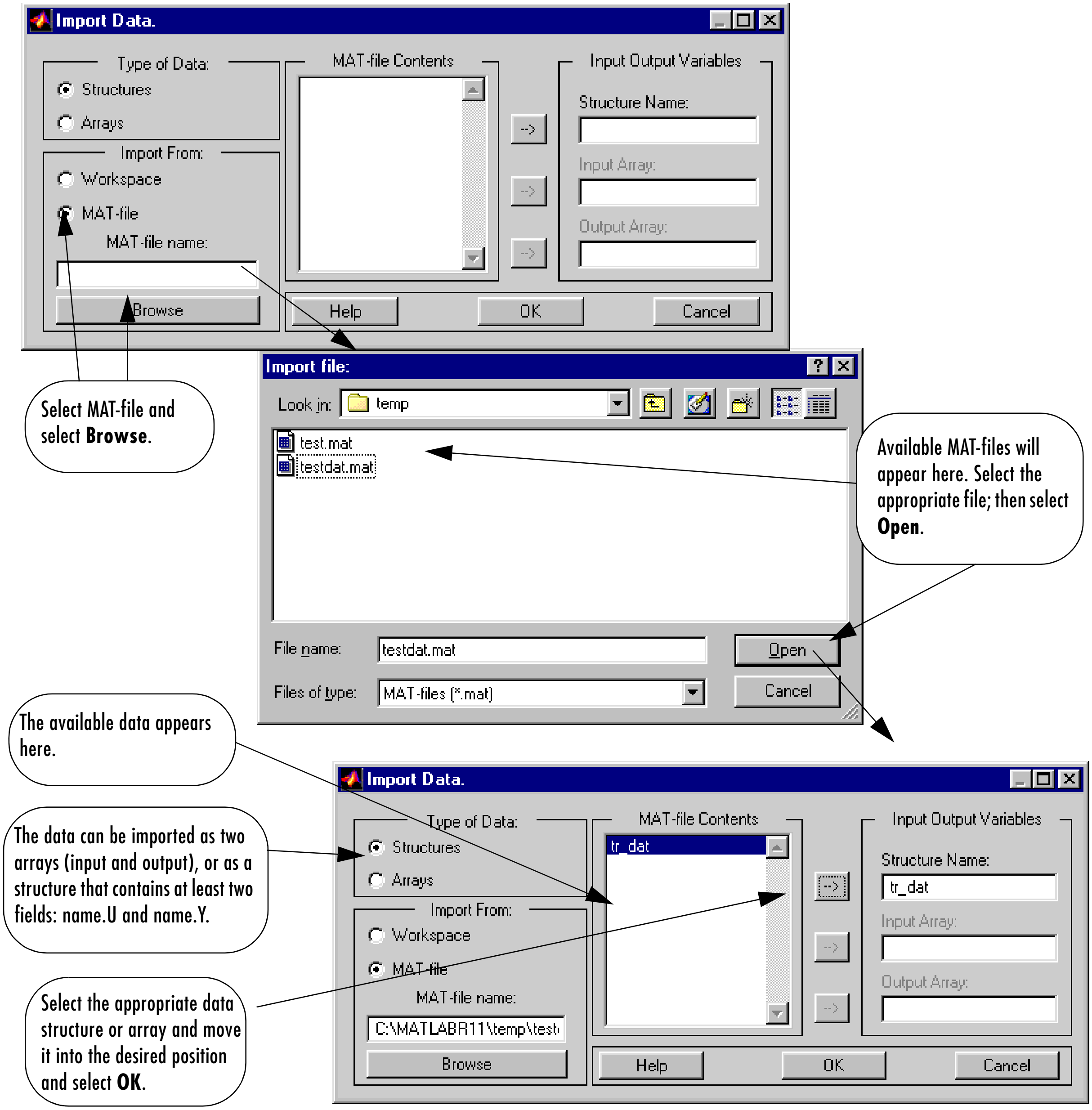
Save in: temp

File name: testdat

Save as type: MAT-files (*.mat)

The filename goes here.

4 Now retrieve the data with the import command. Select the **Import** button in the Plant Identification window. This causes the following window to appear. 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.



Select MAT-file and select **Browse**.

Available MAT-files will appear here. Select the appropriate file; then select **Open**.

The available data appears here.

The data can be imported as two arrays (input and output), or as a structure that contains at least two fields: name.U and name.Y.

Select the appropriate data structure or array and move it into the desired position and select **OK**.

Import Data.

Type of Data: ● Structures ○ Arrays

Import From: ○ Workspace ● MAT-file

Import file:

Look in: temp

test.mat
testdat.mat

File name: testdat.mat

Files of type: MAT-files (*.mat)

Import Data.

Type of Data: ● Structures ○ Arrays

MAT-file Contents: tr_dat

Import From: ○ Workspace ● MAT-file

MAT-file name: C:\MATLABR11\temp\testd

Structure Name: tr_dat

Radial Basis Networks

Introduction

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 chapter 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

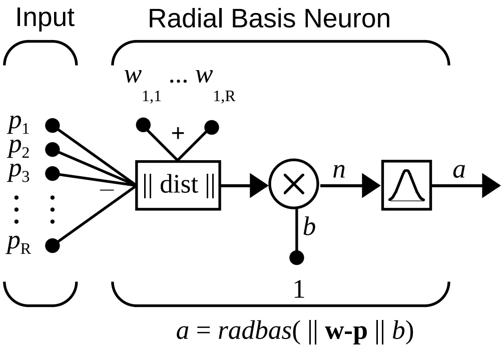
Radial basis networks can be designed with either `newrbe` or `newrb`. GRNNs and PNNs can be designed with `newgrnn` and `newpnn`, respectively.

Type `help radbasis` to see a listing of all functions and demonstrations related to radial basis networks.

Radial Basis Functions

Neuron Model

Here is a radial basis network with R inputs.

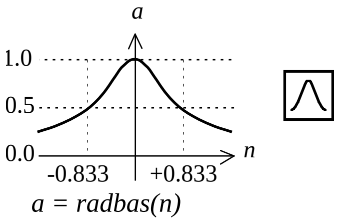


Notice that the expression for the net input of a `radbas` neuron is different from that of other neurons. Here the net input to the `radbas` transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias b. (The `|| dist ||` box in this figure 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

$$radbas(n) = e^{-n^2}$$

Here is a plot of the `radbas` transfer function.



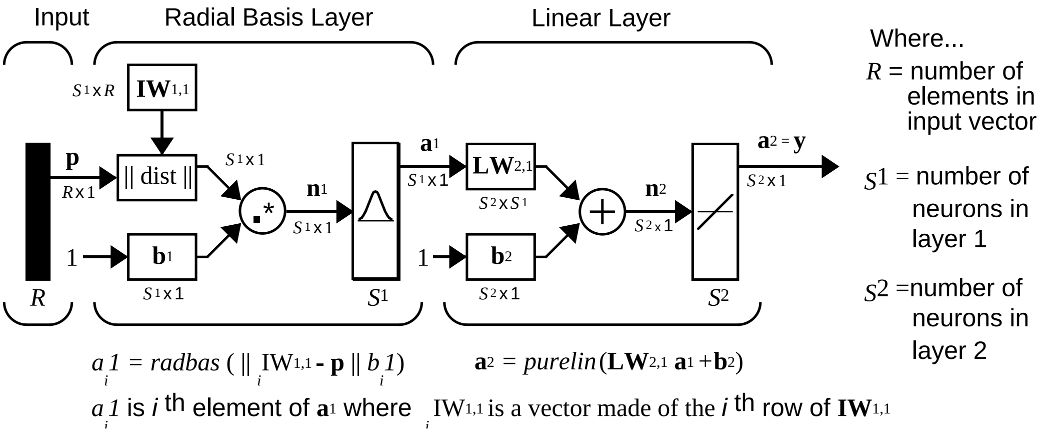
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between **w** and **p** decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input **p** is identical to its weight vector **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 **p** at vector distance of 8.326 (0.8326/b) from its weight vector **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.


Input
Radial Basis Layer
Linear Layer
Where...
R = number of elements in input vector
S1 = number of neurons in layer 1
S2 =number of neurons in layer 2
p
R x 1
R
1
S1 x R
IW1,1
|| dist ||
S1 x 1
.*
b1
S1 x 1
n1
S1 x 1
S1
a1
S1 x 1
1
LW2,1
S2 x S1
b2
S2 x 1
n2
S2 x 1
S2
a2 = y
S2 x 1
a_i1 = radbas (|| _iIW1,1 - p || b_i1)
a2 = purelin(LW2,1 a1 +b2)
a_i1 is i th element of a1 where _iIW1,1 is a vector made of the i th row of IW1,1

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 ${}_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The bias vector $\mathbf{b}^1$ 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 $\mathbf{a}^2$. 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 0's (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 +/- 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 $\mathrm{IW}^{2,1}$ (or in code, IW{2,1}) and biases b^2 (or in code, b{2}) are found by simulating the first-layer outputs a^1 (A{1}), and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones] = 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/[P; 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 demonstration.

Demonstrations

The demonstration `demorb1` shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Demonstrations `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` demonstrated 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. Demonstration `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.

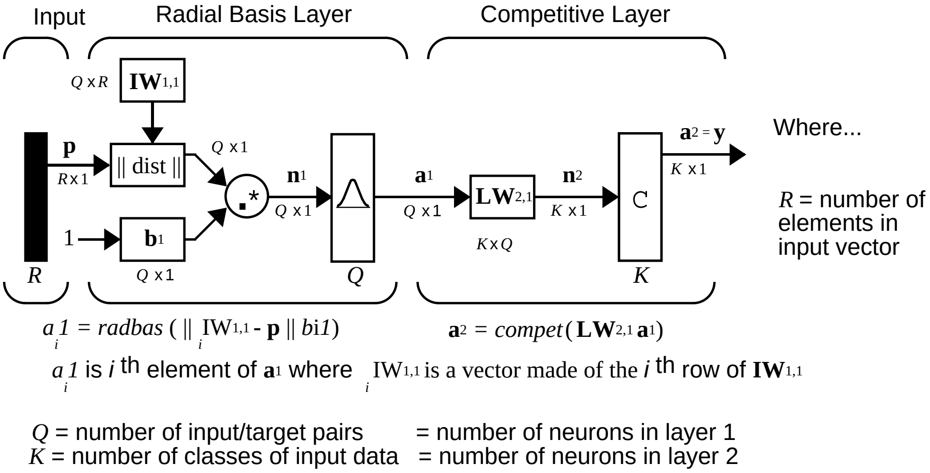
The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

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

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


Input
Radial Basis Layer
Competitive Layer
Q x R
IW1,1
p
R x 1
|| dist ||
Q x 1
.*
n1
Q x 1
1
b1
Q x 1
R
Q
a1
Q x 1
LW2,1
K x Q
n2
K x 1
C
K
a2 = y
K x 1
Where...
R = number of elements in input vector
a_i1 = radbas (|| _iIW1,1 - p || bi1)
a2 = compet(LW2,1 a1)
a_i1 is i th element of a1 where _iIW1,1 is a vector made of the i th row of IW1,1
Q = number of input/target pairs = number of neurons in layer 1
K = number of classes of input data = number of neurons in layer 2

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, $\mathrm{IW}^{1,1}$ (`net.IW{1,1}`), are set to the transpose of the matrix formed from the Q training pairs, **P**`'`. When an input is presented, the `|| 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 **T** of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0's elsewhere. (Use function `ind2vec` to create the proper vectors.) The multiplication $\mathbf{Ta}^1$ sums the elements of $\mathbf{a}^1$ due to each of the `K` input classes. Finally, the second-layer transfer function, `compete`, produces a 1 corresponding to the largest element of $\mathbf{n}^2$, and 0's 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 1's in the right places. You can get it with the function `ind2vec`. It gives a matrix with 0's 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.

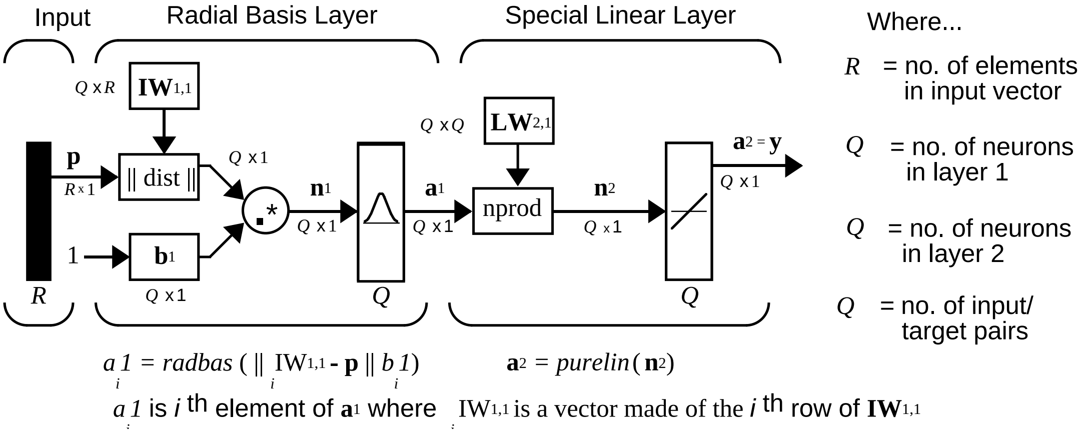
You might want to try demopnn1. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

Generalized Regression Networks

A generalized regression neural network (GRNN) is often used for function approximation. It has a radial basis layer and a special linear layer.

Network Architecture

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.


Input
Radial Basis Layer
Special Linear Layer
Where...
R = no. of elements in input vector
Q = no. of neurons in layer 1
Q = no. of neurons in layer 2
Q = no. of input/ target pairs
Q x R
IW1,1
p
R x 1
|| dist ||
Q x 1
1
b1
Q x 1
.*
n1
Q x 1
Q
a1
Q x 1
Q x Q
LW2,1
nprod
n2
Q x 1
Q
a2 = y
Q x 1
R
a1 = radbas (|| iIW1,1 - p || bi1)
i
a2 = purelin(n2)
ai1 is i th element of a1 where iIW1,1 is a vector made of the i th row of IW1,1

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 $\mathbf{b}^1$ 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 newbe 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 $\mathbf{p}_i$, one of the input vectors among the input vector/target pairs used in designing layer 1 weights. This input **p** produces a layer 1 $\mathbf{a}^1$ output close to 1. This leads to a layer 2 output close to $\mathbf{t}_i$, 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 `demogrn1`. It shows how to approximate a function with a GRNN.

Self-Organizing and Learning Vector Quantization Nets

Introduction

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 `newc` and `newsom`, respectively. You can type `help selforg` to find a listing of all self-organizing functions and demonstrations.

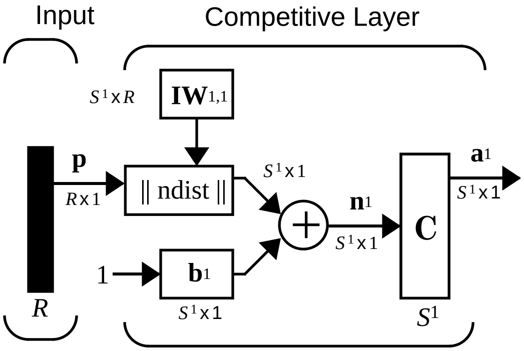
You can create an LVQ network with the function `newlvq`. For a list of all LVQ functions and demonstrations, type `help lvq`.

Competitive Learning

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 `|| 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 negative of the distances between the input vector and vectors ${}_i\mathbf{IW}^{1,1}$ 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 **p** and the weight vectors and adding the biases **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 $\mathbf{n}^1$. 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.

Creating a Competitive Neural Network (newc)

You can create a competitive neural network with the function newc. 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 layer with two input elements ranging from 0 to 1. The first argument gives the ranges of the two input vectors, and the second argument says that there are to be two neurons.

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

The weights are initialized to the centers of the input ranges with the function midpoint. You can check to see these initial values as follows:

```
wts = net.IW{1,1}
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, as you would expect when using midpoint for initialization.

The biases are computed by initcon, which gives

```
biases = net.b{1}
biases =
    5.4366
    5.4366
```

Now you have a network, but you need to train it to do the classification job.

Recall that each neuron competes to respond to an input vector **p**. If the biases are all 0, the neuron whose weight vector is closest to **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 `trainr`. You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

This code produces

```
ans =
trainr
```

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)
```

This yields

```
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. They are

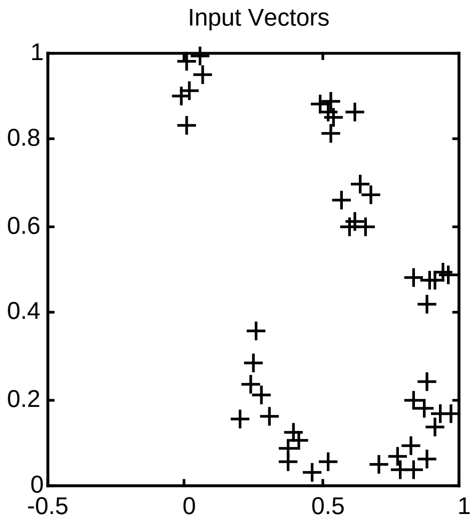
```
wts =
    0.1000    0.1467
    0.8474    0.8525
biases =
    5.4961
    5.3783
```

(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.

Self-Organizing Feature Maps

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 7-10 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:

$${}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \text{ or}$$

$${}_i\mathbf{w}(q) = (1-\alpha)\,{}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

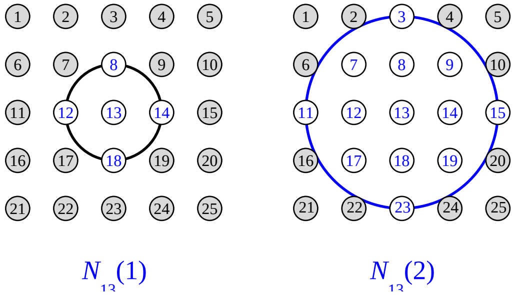
Here the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*.

$$N_i(d) = \{j, d_{ij} \le d\}$$

Thus, when a vector **p** is presented, the weights of the winning neuron *and* its close neighbors move toward **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

$$N_{13}(1) = \{8, 12, 13, 14, 18\} \text{ and}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$$

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.

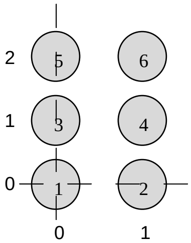
Topologies (gridtop, hextop, randtop)

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.



gridtop(2,3)

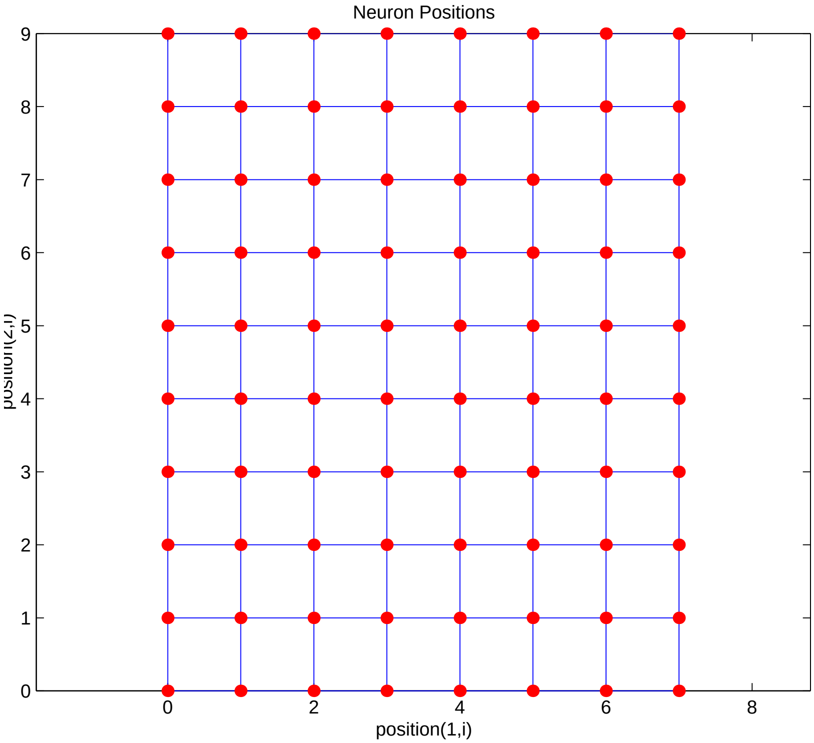
Note that had you asked for a `gridtop` with the arguments reversed, you would have gotten a slightly different arrangement:

```
pos = gridtop(3,2)
pos =
     0     1     2     0     1     2
     0     0     0     1     1     1
```

An 8-by-10 set of neurons in a `gridtop` topology can be created and plotted with the following code:

```
pos = gridtop(8,10);
plotsom(pos)
```

to give the following graph.



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
```

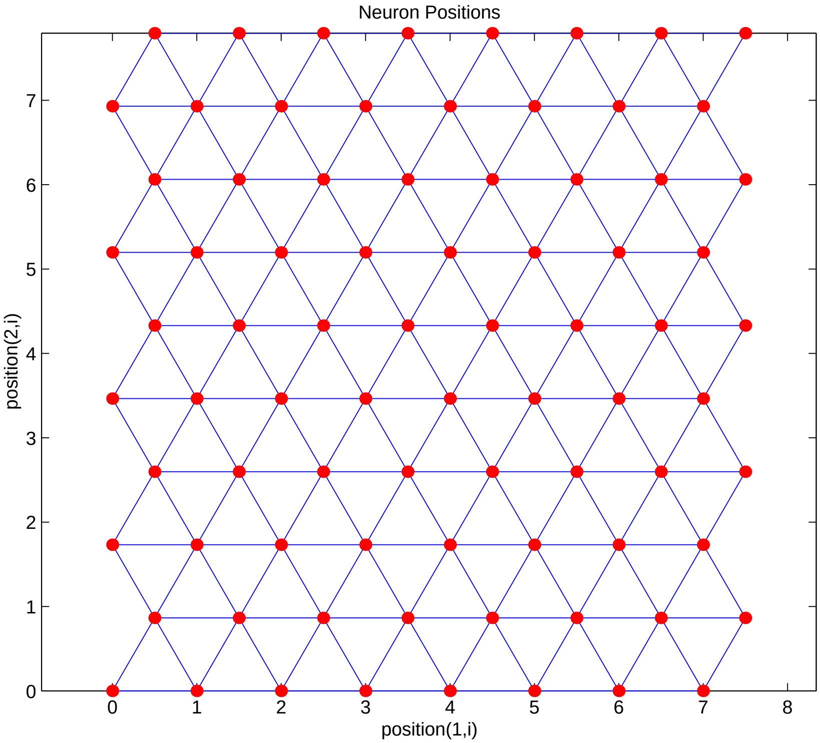
Note that `hextop` is the default pattern for SOFM networks generated with `newsom`.

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)
```

to give the following graph.



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.

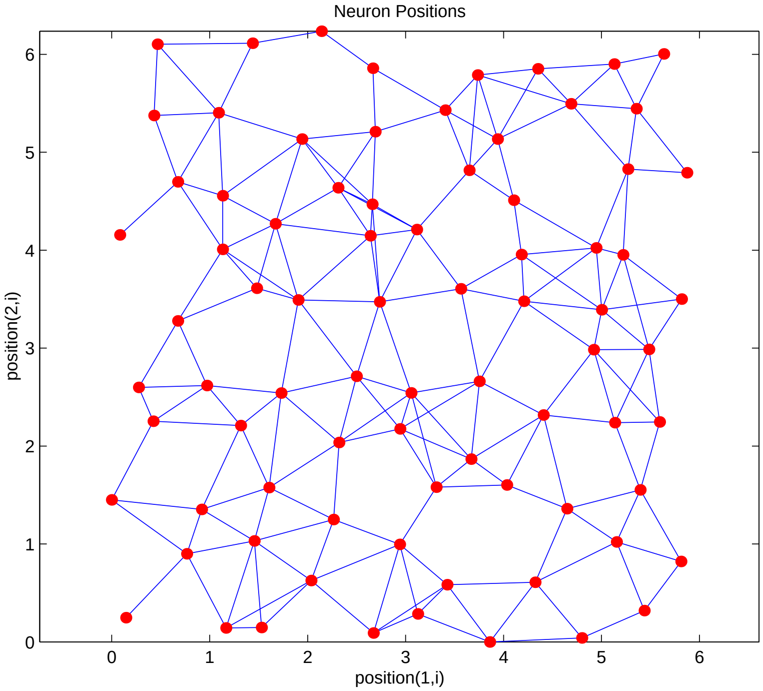
```
pos = randtop(2,3)
pos =
         0    0.7620    0.6268    1.4218    0.0663    0.7862
    0.0925         0    0.4984    0.6007    1.1222    1.4228
```

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)
```

to give the following graph.



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:

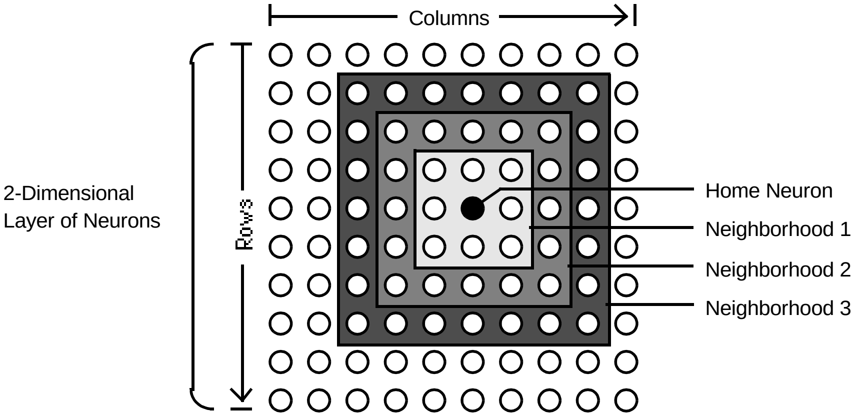
```
pos2 = [0 1 2; 0 1 2]
pos2 =
     0     1     2
     0     1     2
```

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
```

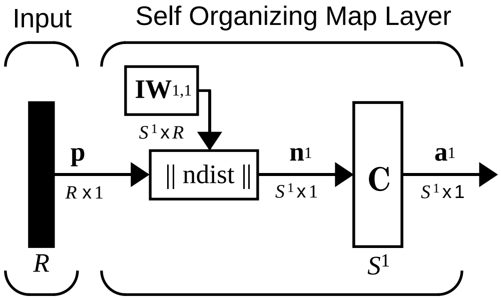
then you get for the distances

```
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.



$$n^1_i = -\,\|\,{}_i\mathrm{IW}^{1,1} - \mathbf{p}\,\|$$

$$\mathbf{a}^1 = \mathbf{compet}(\mathbf{n}^1)$$

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 $\mathbf{a}^1_i$ corresponding to $i*$, the winning neuron. All other output elements in $\mathbf{a}^1$ 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.

Creating a Self-Organizing MAP Neural Network (newsom)

You can create a new SOM network with the function newsom. 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 that fall in the ranges 0 to 2 and 0 to 1, respectively. Further suppose that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is

```
net = newsom([0 2; 0 1],[2 3]);
```

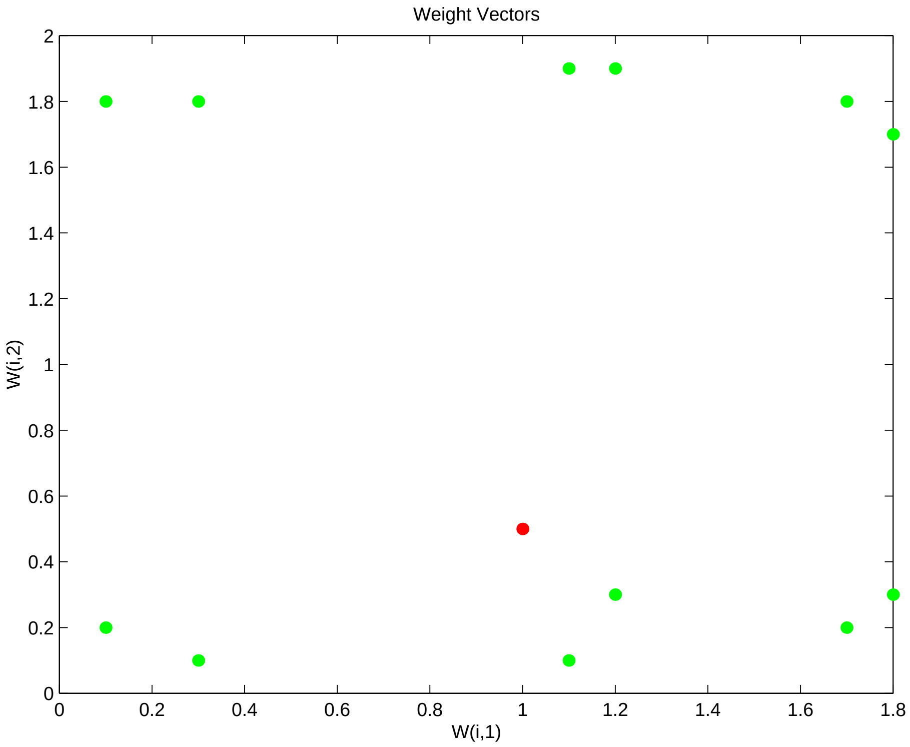
Suppose also that the vectors to train on are

```
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 plot all of this with

```
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

to give



The various training vectors are seen as fuzzy gray spots around the perimeter of this figure. The initialization for `newsom` is `midpoint`. Thus, the initial network neurons are all concentrated at the black spot at (1, 0.5).

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 (`compete`) 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 (`trainbuwb`). 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 distance stays at the tuning neighborhood distance, (which should include only close neighbors, i.e., typically 1.0). The small neighborhood fine-tunes the network, while keeping the ordering learned in the previous phase stable.

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
`LP.init_neighborhood`	`3`	Initial neighborhood size
`LP.steps`	`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 always set to 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.

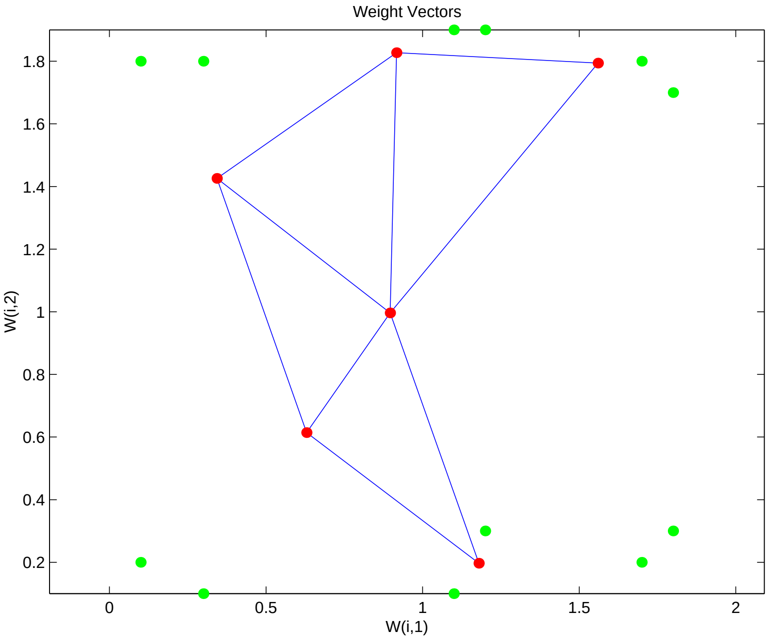
As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature 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);
```

Call `plotsom` to see the data produced by the training procedure, shown in the following plot.



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 might try the demonstrations `demosm1` and `demosm2` to see similar examples.

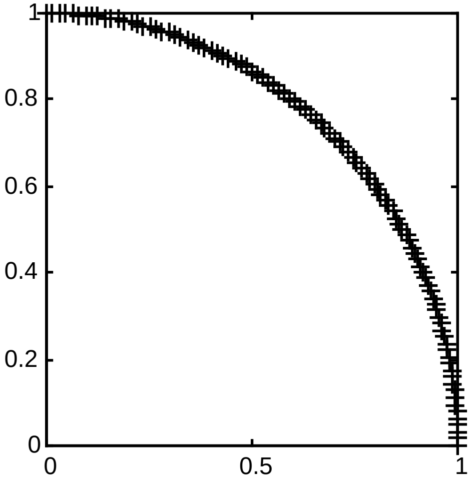
One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90°.

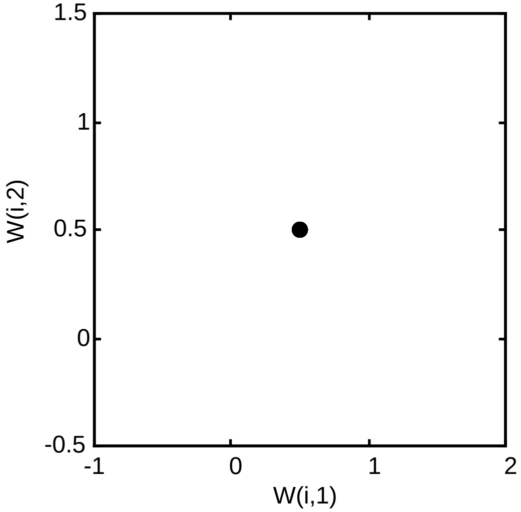
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

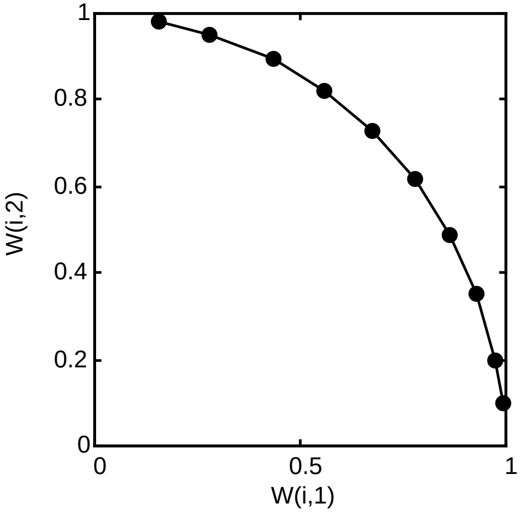


A a self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons are at the center of the figure.



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.

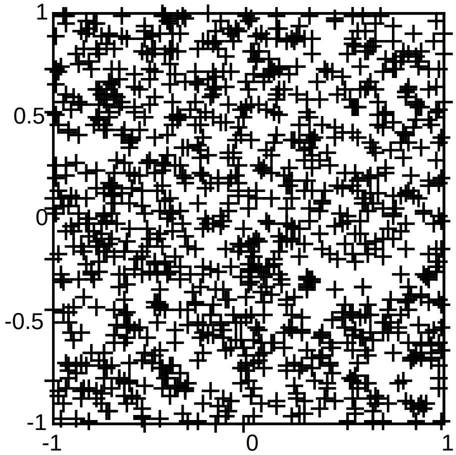
Two-Dimensional Self-Organizing Map

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);
```

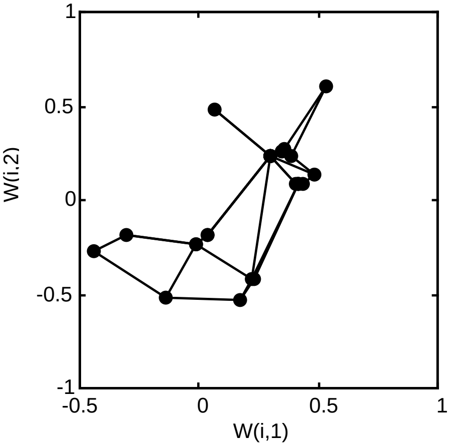
Here is a plot of these 1000 input vectors.



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`.

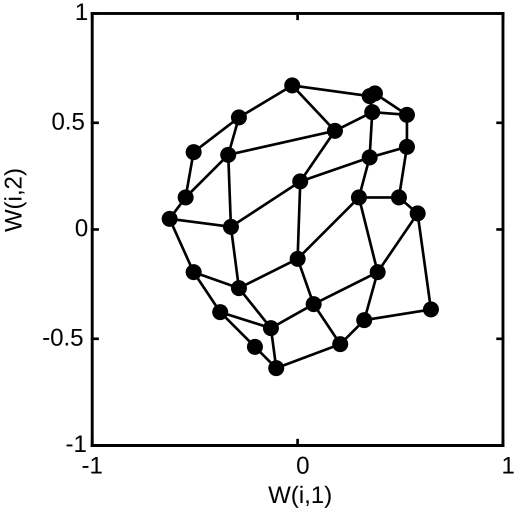
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



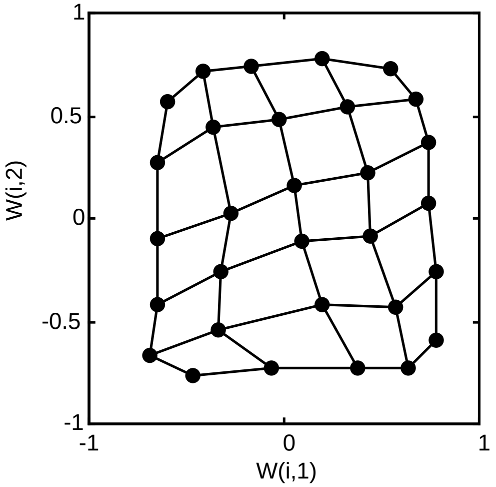
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

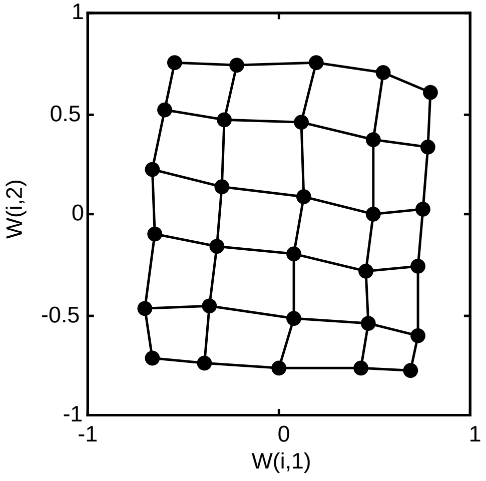


After 120 cycles, the map has begun to organize itself according to the topology of the input space, which constrains input vectors.

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.

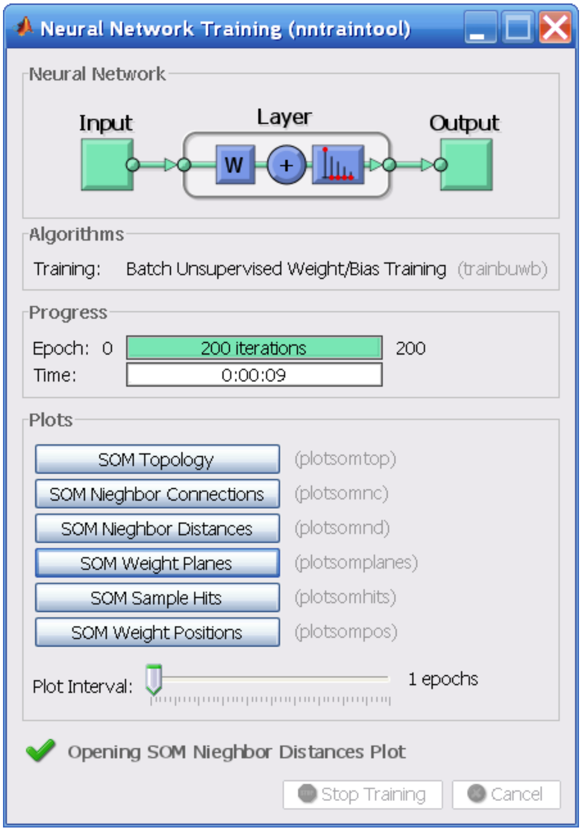
It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

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:

```
load simplecluster_dataset
net = newsom(simpleclusterInputs,[6 6]);
[net2,tr] = train(net,simpleclusterInputs);
```

This command sequence creates and trains a 6-by-6 two-dimensional map of 36 neurons. During training, the following figure appears.



Training: Batch Unsupervised Weight/Bias Training (trainbuwb)

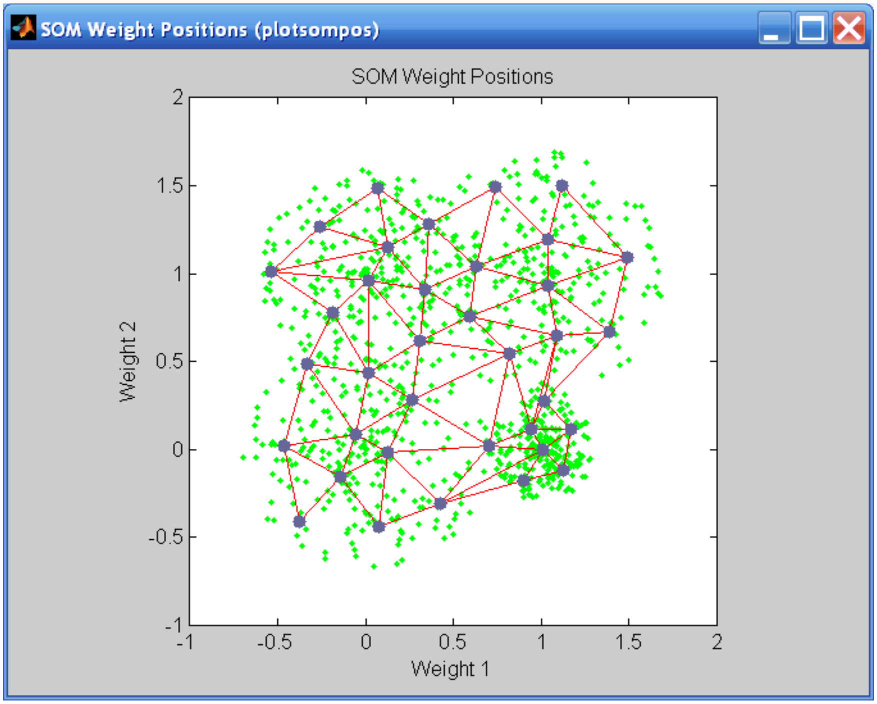
Epoch: 0 200 iterations 200

Time: 0:00:09

Plot Interval: 1 epochs

✓ Opening SOM Nieghbor Distances Plot

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.

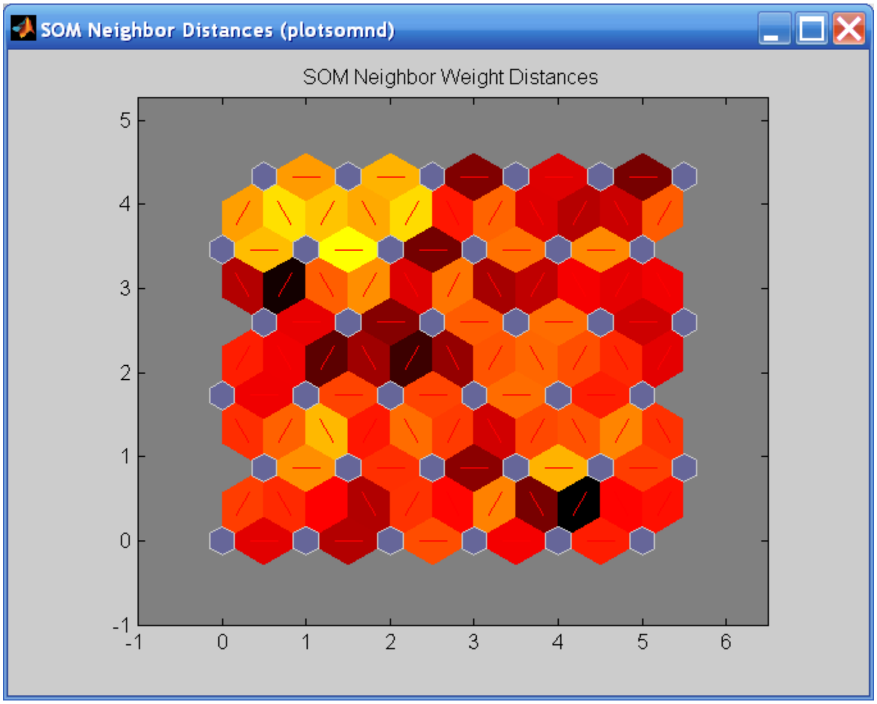


When the input space is high dimensional, you cannot visualize all the weights at the same time. In this case, click **SOM Neighbor Distances**. The following figure appears, which indicates the distances between neighboring neurons.

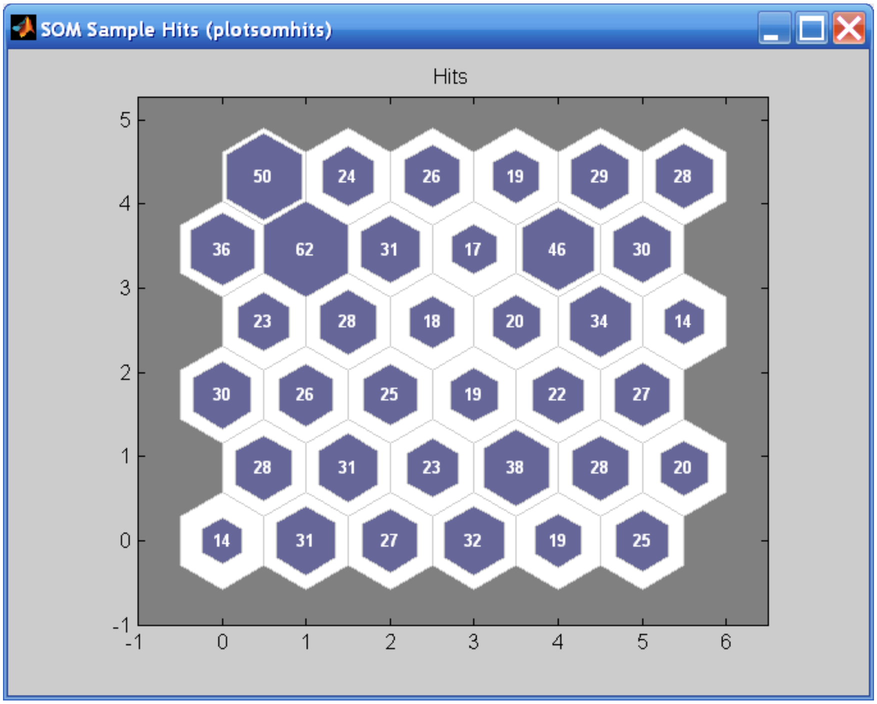
This figure uses the following color coding:

- 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.

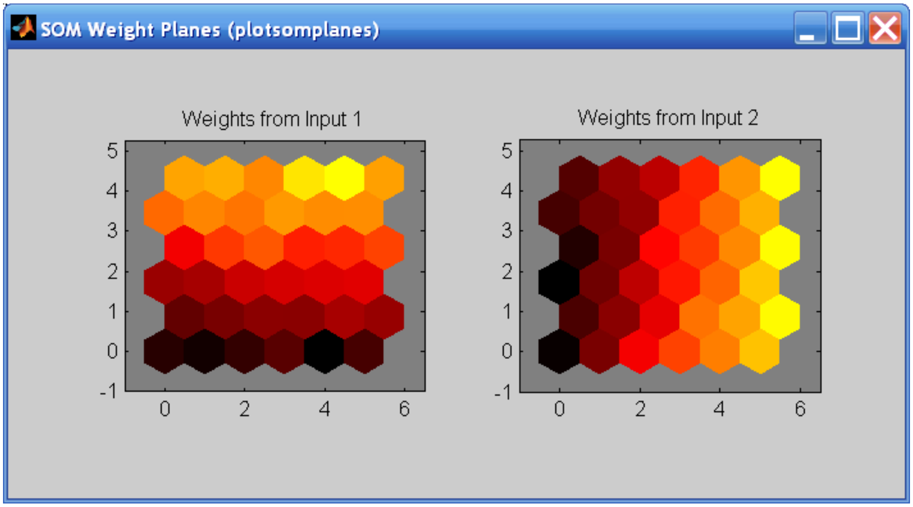
A group of light segments appear in the upper-left region, bounded by some darker segments. This grouping indicates that the network has clustered the data into two groups. These two groups can be seen in the previous weight position figure. The lower-right region of that figure contains a small group 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.


SOM Neighbor Distances (plotsomnd)
SOM Neighbor Weight Distances
5
4
3
2
1
0
-1
-1
0
1
2
3
4
5
6

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. (Darker colors represent larger weights.) 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.

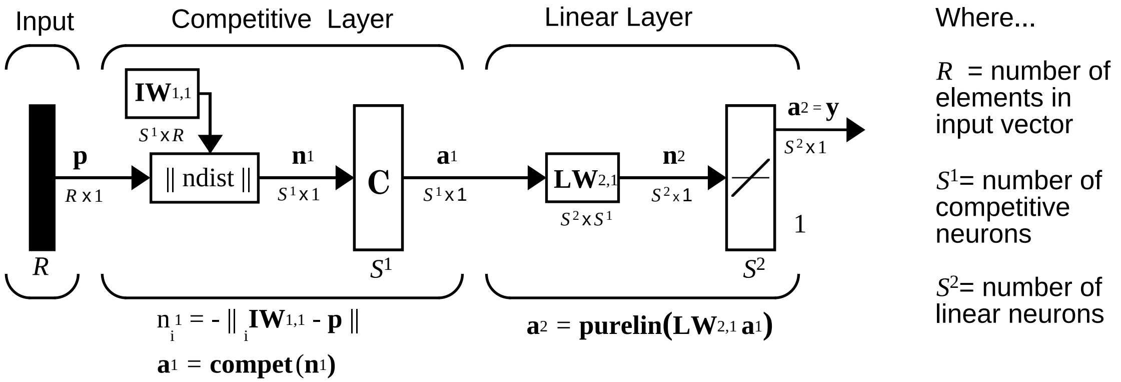


You can also produce all of the previous figures from the command line. Try these plotting commands: `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`. (See their reference pages for details.)

Learning Vector Quantization Networks

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 "Self-Organizing and Learning Vector Quantization Nets" described in this chapter. 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 ith row of $\mathbf{a}^1$ (the rest to the elements of $\mathbf{a}^1$ will be zero) effectively picks the ith column of $\mathbf{LW}^{2,1}$ as the network output. Each such column contains a single 1, corresponding to a specific class. Thus, subclass 1's 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 7-40. First, consider how to create the original network.

Creating an LVQ Network (newlvq)

You can create an LVQ network with the function `newlvq`,

```
net = newlvq(PR,S1,PC,LR,LF)
```

where

- `PR` is an R-by-2 matrix of minimum and maximum values for R input elements.
- S^1 is the number of first-layer hidden neurons.
- `PC` is an S^2-element vector of typical class percentages.
- `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];
```

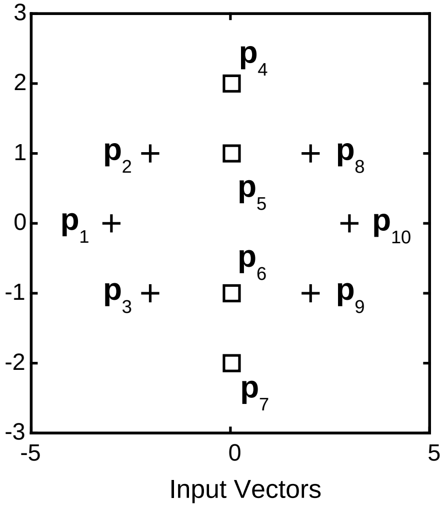
and

```
Tc = [1 1 1 2 2 2 2 1 1 1];
```

It might help to show the details of what you get from these two lines of code.

```
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 `Tc` 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 `newlvq`.

Call `newlvq` with the proper arguments so that it creates a network with four neurons in the first layer and two neurons in the second layer. The first-layer weights are initialized to the centers of the input ranges with the function `midpoint`. The second-layer weights have 60% (6 of the 10 in `Tc` above) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns will have a 1 in the second row (corresponding to class 2).

```
net = newlvq(P,4,[.6 .4]);
```

Confirm the initial values of the first-layer weight matrix.

```
net.IW{1,1}
ans =
     0     0
     0     0
     0     0
     0     0
```

These zero weights are indeed the values at the midpoint of the ranges (-3 to +3) of the inputs, as you would expect when using `midpoint` for initialization.

You can look at the second-layer weights with

```
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 = sim(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.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

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 `ndist`. The hidden neurons of layer 1 compete. Suppose that the ith 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^2_{k*} 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,

$$(a^2_{k*} = t_{k*} = 1)$$

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,

$$(a^2_{k*} = 1 \neq t_{k*} = 0)$$

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`:

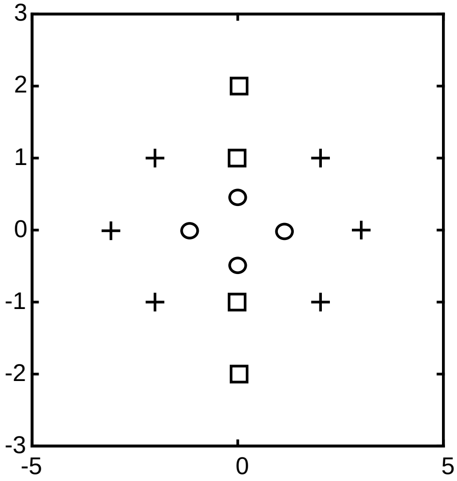
```
net.trainParam.epochs = 150;
```

```
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.



Weights (circles) after training

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 = sim(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 = sim(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 = sim(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 demonstration 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 learnlv1 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

$$\min\left(\frac{d_i}{d_j},\frac{d_j}{d_i}\right) > s \quad \text{where} \quad \mathrm{s} \equiv \frac{1-w}{1+w}$$

(where d_i and d_j are the Euclidean distances of 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 **p** and ${}_{j*}\mathbf{IW}^{1,1}$ belong to the same class, and **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.

Adaptive Filters and Adaptive Training

Introduction

The ADALINE (adaptive linear neuron) networks discussed in this chapter 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 only solve 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 chapter, 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 cancellation, 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 chapter.

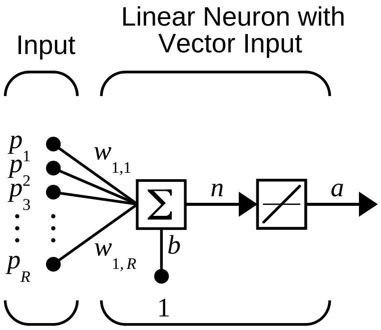
Important Adaptive Functions

This chapter introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

You can type `help linnet` to see a list of linear and adaptive network functions, demonstrations, and applications.

Linear Neuron Model

A linear neuron with R inputs is shown below.

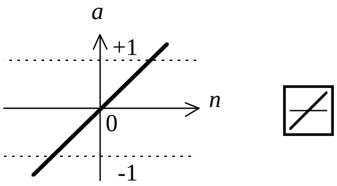


Where...

R = number of elements in input vector

$a = purelin(\mathbf{Wp} + b)$

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`.



$a = purelin(n)$

Linear Transfer Function

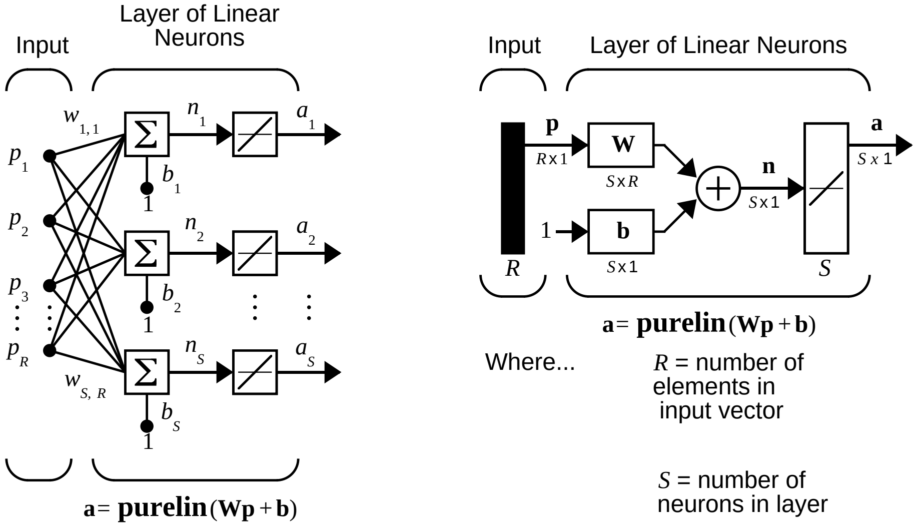
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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.

Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of S neurons connected to R inputs through a matrix of weights $\mathbf{W}$.



Where...

R = number of elements in input vector

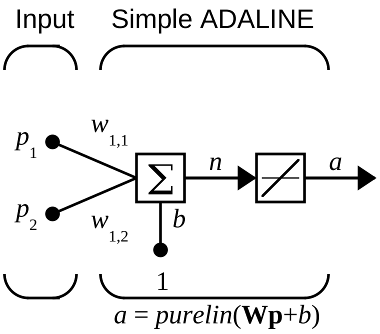
S = number of neurons in layer

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 (newlin)

Consider a single ADALINE with two inputs. The following figure shows the diagram for this network.



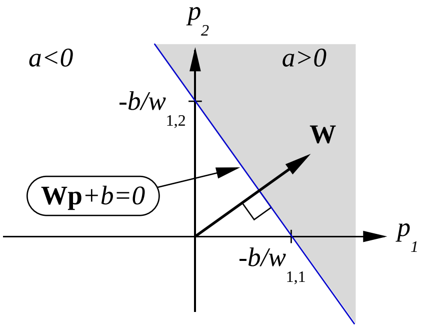
The weight matrix **W** in this case has only one row. The network output is

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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 $\mathbf{Wp} + b = 0$ 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.

We can create a network similar to the one shown using this command:

```
net = newlin([-1 1; -1 1],1);
```

The first matrix of arguments specifies typical two-element input vectors, and the last argument `1` indicates that the network has a single output.

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 ADAPLINE for a particular input vector.

```
p = [5; 6];
a = sim(net,p)
a =
    24
```

To summarize, you can create an ADALINE 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.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

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.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

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 below, is discussed in detail in “Linear Networks” on page 10-18.

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

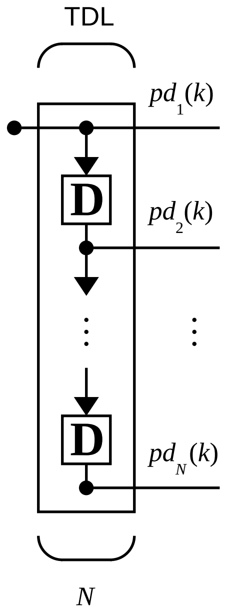
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

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.

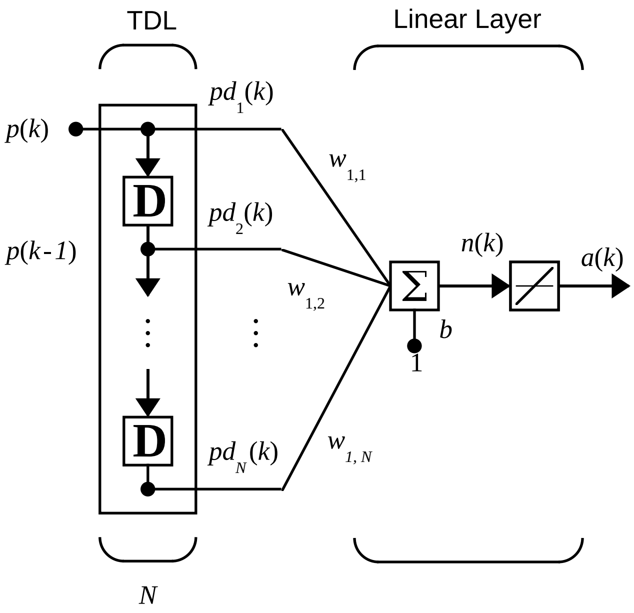
Tapped Delay Line

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.



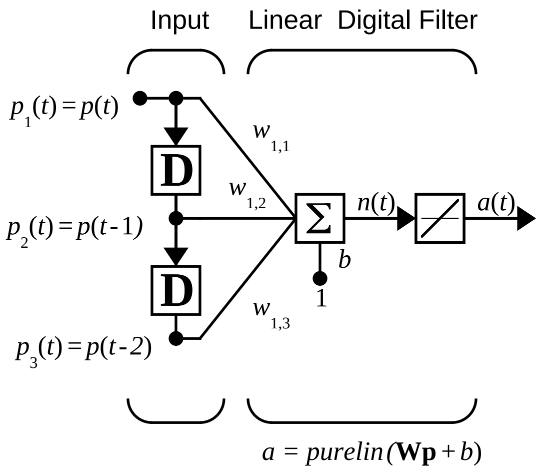
The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i} a(k - i + 1) + b$$

In digital signal processing, this network is referred to as a f*inite 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 `newlin`.



Assume that the input values have a range from 0 to 10. You can now define the single output network.

```
net = newlin([0,10],1);
```

Specify the delays in the tapped delay line with

```
net.inputWeights{1,1}.delays = [0 1 2];
```

This definition indicates that the delay line connects to the network weight matrix through delays of 0, 1, and 2 time units. (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. Specify 10 passes through the input sequence with

```
net.adaptParam.passes = 10;
```

Then launch the training with

```
[net,y,E,pf,af] = adapt(net,p,t,pi);
```

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
```

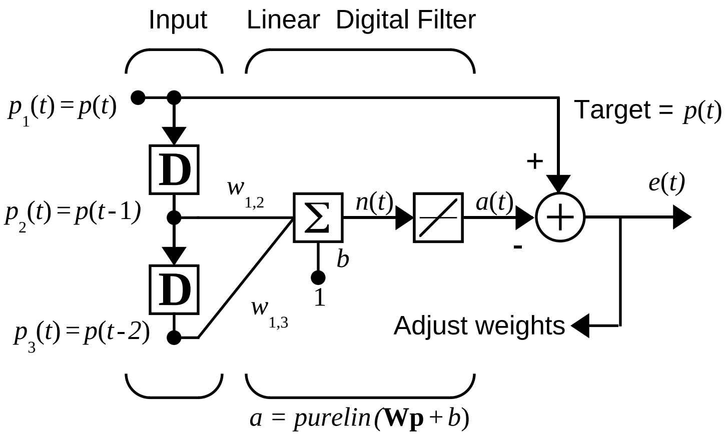
```
y =
    [11.8558]    [20.7735]    [29.6679]    [39.0036]
```

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 cancellation.

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.



Predictive Filter: $a(t)$ is approximation to $p(t)$

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.

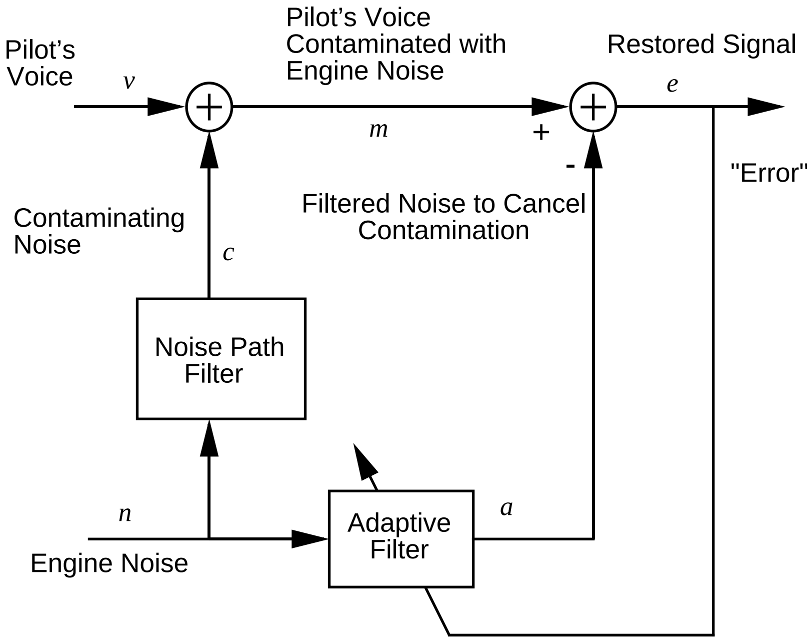
Given the autocorrelation function of the stationary random process *p(t)*, you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be performed. 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 demonstration `nnd10nc` to see an adaptive noise cancellation program example in action. This demonstration allows you to pick a learning rate and *momentum* (see Chapter 3, "Multilayer Networks and Backpropagation Training"), and shows the learning trajectory, and the original and cancellation signals versus time.

Noise Cancellation 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.

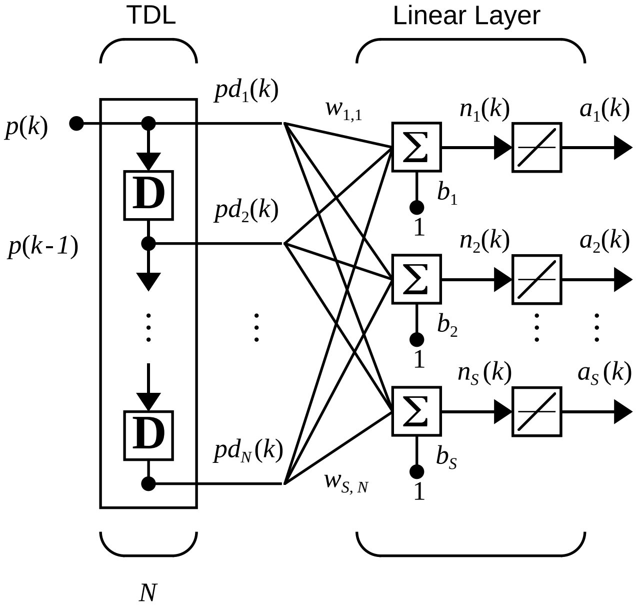
The network does its best to output m adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal m. The network error e is equal to m, the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, e contains only the pilot's 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.

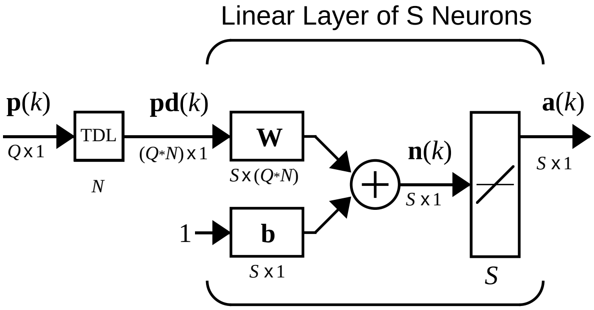
Try `demolin8` for an example of adaptive noise cancellation.

Multiple Neuron Adaptive Filters

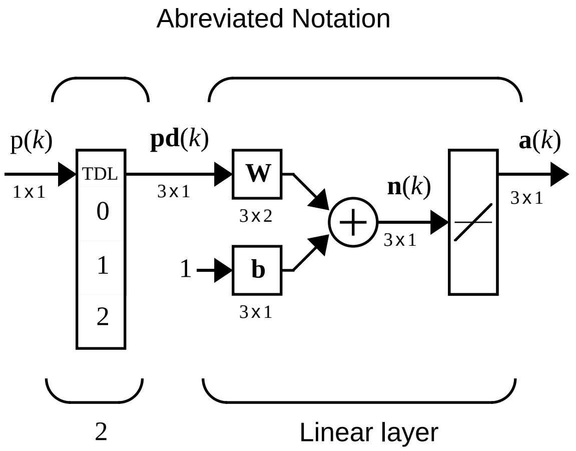
You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with S linear neurons, as shown in the next figure.



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:



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 appears in increasing order as they go from top to bottom.

Advanced Topics

Custom Networks

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 nnnetwork
```

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.

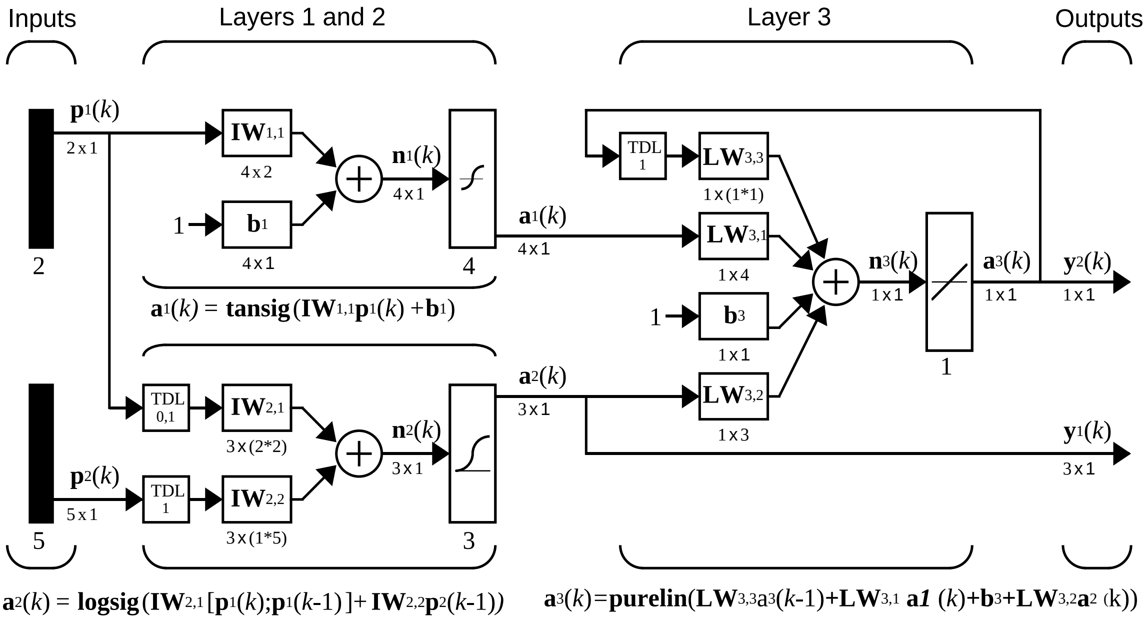
```
net = network
```

The network object consists of many properties that you can set to specify the structure and behavior of your network. See Chapter 11, “Network Object Reference,” for descriptions of all network properties.

The following sections demonstrate 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 are `numInputs` and `numLayers`. These properties allow you to select how many inputs and layers you want the network to have.

```
net =

    Neural Network object:

    architecture:

         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 object:

    architecture:

         numInputs: 2
         numLayers: 3
```

Examine the next four properties:

```
       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 1's and 0's 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 is the number of outputs:

```
numOutputs: 2 (read-only)
```

By defining output connection from layers 2 and 3, you specified that the network has two outputs.

Subobject Properties

The next group of properties is

```
subobject structures:

            inputs: {2x1 cell} of inputs
            layers: {3x1 cell} of layers
           outputs: {1x3 cell} containing 1 output
            biases: {3x1 cell} containing 2 biases
      inputWeights: {3x2 cell} containing 3 input weights
      layerWeights: {3x3 cell} containing 3 layer 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 struct]
    [1x1 struct]
```

To see the properties associated with the first input, type

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

The properties appear as follows:

```
ans = 

     exampleInput: [0 1]
      processFcns: {}
    processParams: {}
  processSettings: {}
   processedRange: [0 1]
    processedSize: 1
            range: [0 1]
             size: 1
         userdata: [1x1 struct]
```

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 pages for each processing function to learn more about the function 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 =

     dimensions: 1
    distanceFcn: 'dist'
      distances: 0
        initFcn: 'initwb'
    netInputFcn: 'netsum'
  netInputParam: [1x1 struct]
      positions: 0
           size: 1
    topologyFcn: 'hextop'
    transferFcn: 'purelin'
  transferParam: [1x1 struct]
       userdata: [1x1 struct]
```

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 only need to set its initialization function, as follows:

```
net.layers{3}.initFcn = 'initnw';
```

Outputs. Look at how the outputs property is arranged with this line of code.

```
net.outputs
ans = 

     []    [1x1 struct]    [1x1 struct]
```

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 = 

    exampleOutput: []
      processFcns: {}
    processParams: {}
  processSettings: {}
   processedRange: []
    processedSize: 1
            range: []
             size: 3
         userdata: [1x1 struct]
```

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 like.

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 struct]
              []
    [1x1 struct]
```

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:

```
ans =
       initFcn: ''
         learn: 1
      learnFcn: ''
    learnParam: ''
          size: 4
      userdata: [1x1 struct]
```

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)
          divideFcn: (none)
        gradientFcn: (none)
            initFcn: (none)
         performFcn: (none)
          plotFcns: {}
           trainFcn: (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, look at the final group of network properties (aside from the userdata property).

```
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 1's 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 =
      epochs: 100
        goal: 0
    max_fail: 5
   mem_reduc: 1
    min_grad: 1.0000e-10
          mu: 1.0000e-03
      mu_dec: 0.1000
      mu_inc: 10
      mu_max: 1.0000e+10
        show: 25
        time:
```

You will not often need to modify these values. See the documentation for the training function for information about what each of these mean. They have been initialized with default values that work well for a large range of problems, so we will not 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.

Additional Toolbox Functions

Most toolbox functions are explained in chapters dealing with networks that use them. However, some functions are not used by toolbox networks, but are included because they might be useful to you in creating custom networks.

For instance, `satlin` and `softmax` are two transfer functions not used by any standard network in the toolbox, but which you can use in your custom networks. See the reference pages for more information.

Speed and Memory Comparison for Training Multilayer Networks

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	
LM	`trainlm`	Levenberg-Marquardt
BFG	`trainbfg`	BFGS Quasi-Newton
RP	`trainrp`	Resilient Backpropagation
SCG	`trainscg`	Scaled Conjugate Gradient
CGB	`traincgb`	Conjugate Gradient with Powell/Beale Restarts
CGF	`traincgf`	Fletcher-Powell Conjugate Gradient
CGP	`traincgp`	Polak-Ribiére Conjugate Gradient
OSS	`trainoss`	One Step Secant
GDX	`traingdx`	Variable Learning Rate Backpropagation

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

Problem Title	Problem Type	Network Structure	Error Goal	Computer
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

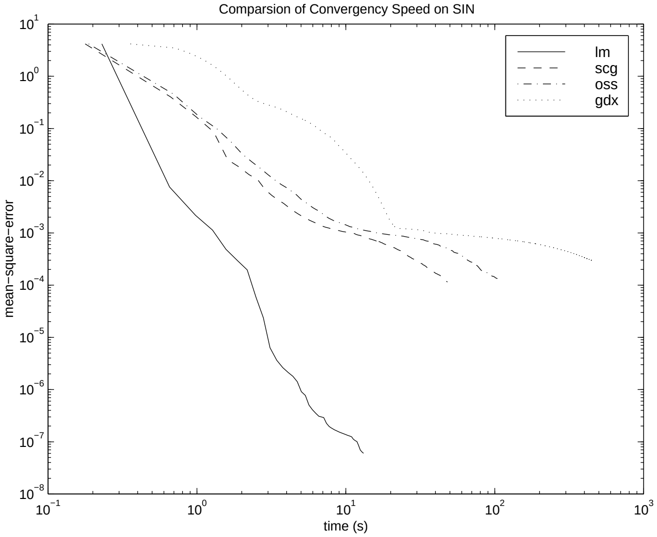
SIN Data Set

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.

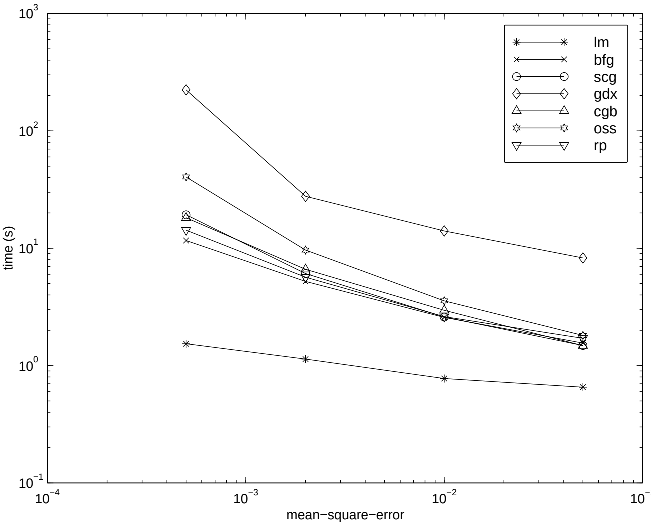
Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	1.14	1.00	0.65	1.83	0.38
BFG	5.22	4.58	3.17	14.38	2.08
RP	5.67	4.97	2.66	17.24	3.72

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
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 demonstrated 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).

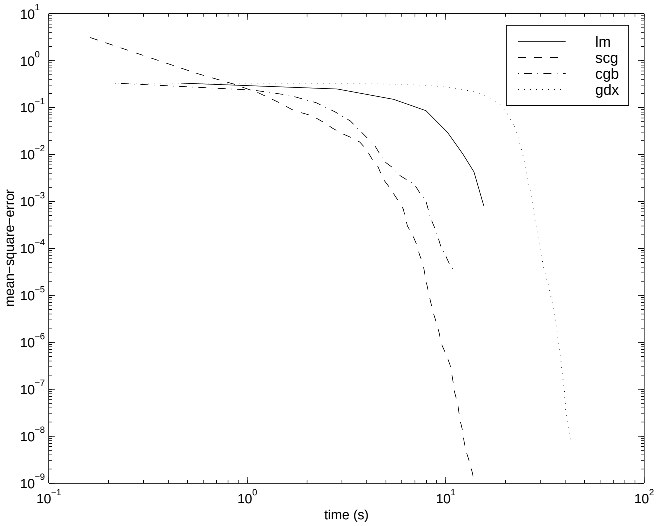


PARITY Data Set

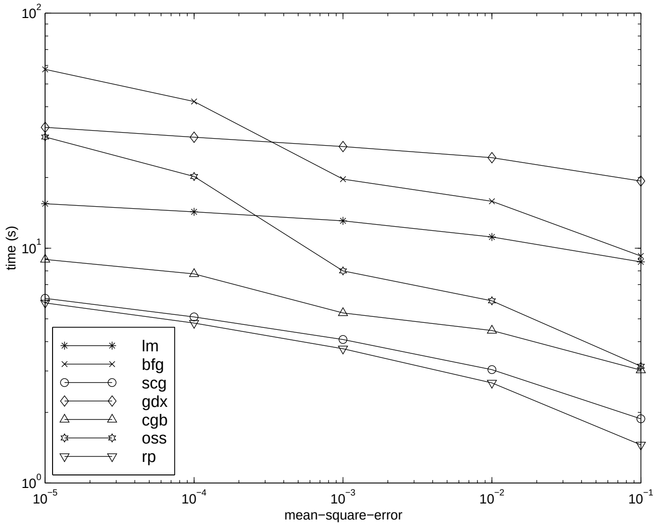
The second benchmark problem is a simple pattern recognition problem—detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a 1; otherwise, it should output a -1. The network used for this problem is a 3-10-10-1 network with tansig neurons 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
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 demonstrated 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).

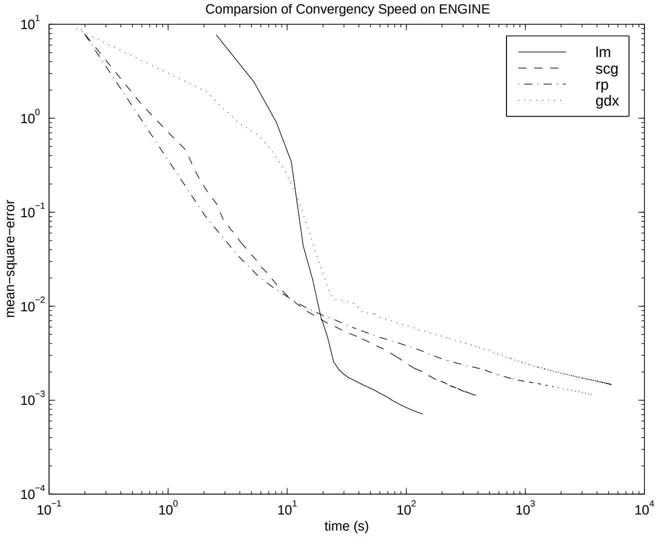


ENGINE Data Set

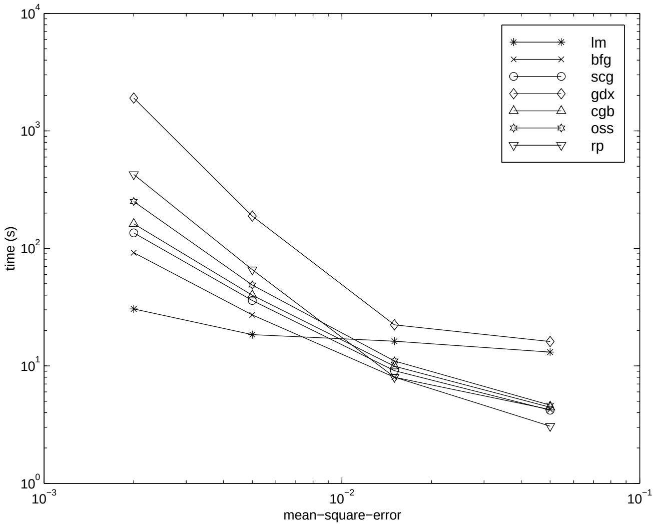
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem 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
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.



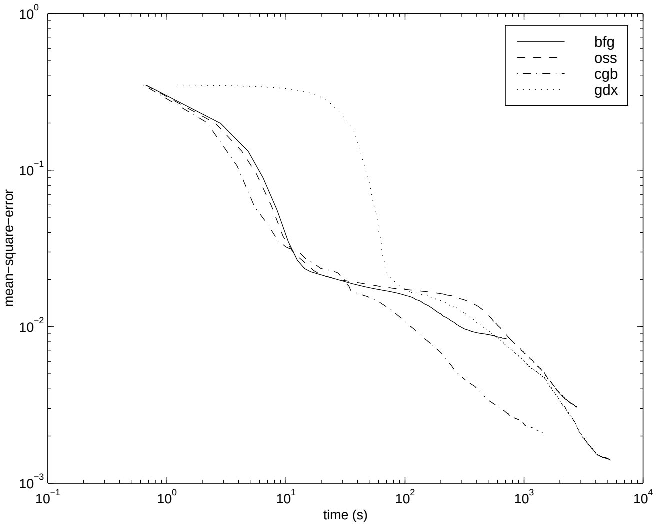
CANCER Data Set

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.

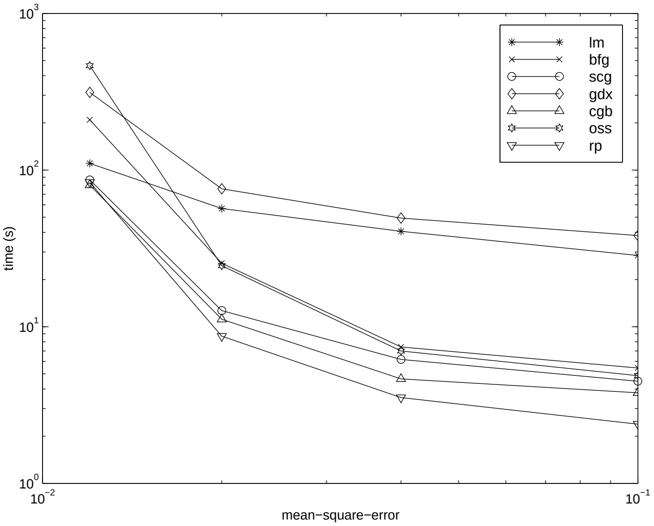
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

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.



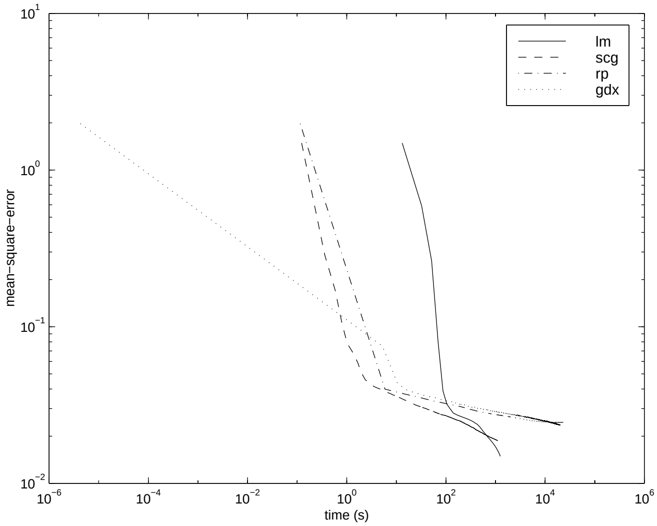
CHOLESTEROL Data Set

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.

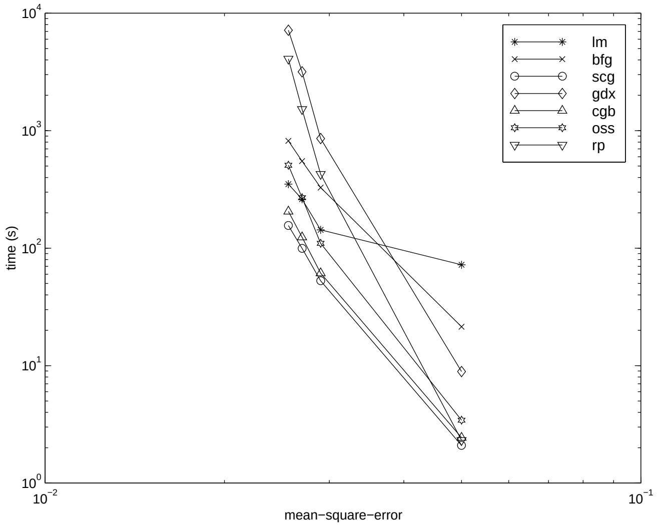
The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases 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	1.24	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

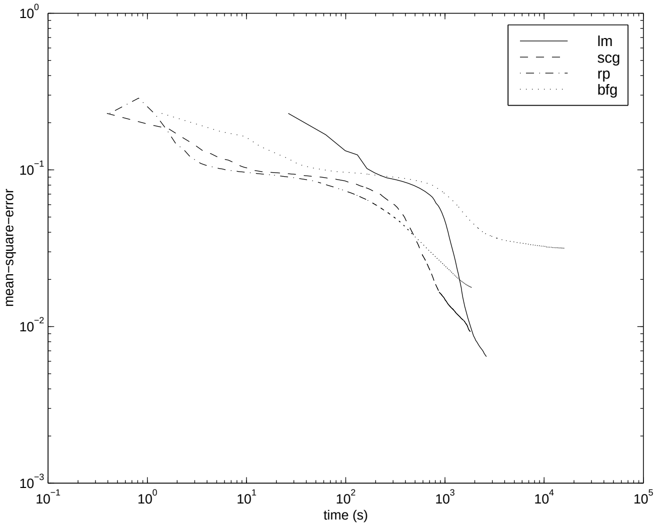
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide whether an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance 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.

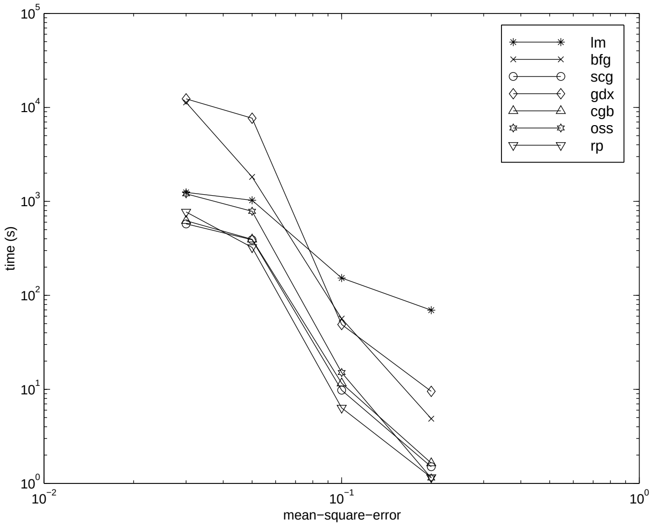
Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	323.90	1.00	187.43	576.90	111.37
SCG	390.53	1.21	267.99	487.17	75.07
CGB	394.67	1.22	312.25	558.21	85.38
CGP	415.90	1.28	320.62	614.62	94.77
OSS	784.00	2.42	706.89	936.52	76.37
CGF	784.50	2.42	629.42	1082.20	144.63
LM	1028.10	3.17	802.01	1269.50	166.31

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
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.

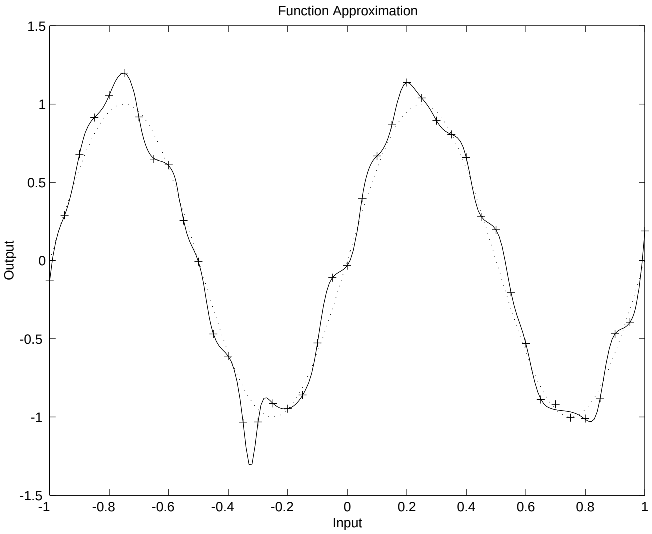
The performance of trainbfg is similar to that of trainlm. It does not require as much storage as trainlm, but the computation required does increase geometrically with the size of the network, because the equivalent of a matrix inverse must be computed at each iteration.

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.

Improving Generalization

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* demonstration `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.

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 `newff`.

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.

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

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

$$msereg = \gamma mse + (1-\gamma)msw$$

where γ 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.)

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(p,t,3,{},'trainbfg');
net.divideFcn = '';
net.performFcn = 'msereg';
net.performParam.ratio = 0.5;
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,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 on page 9-34. (Data division is cancelled by setting net.divideFcn so that the effects of trainbr are isolated from early stopping.)

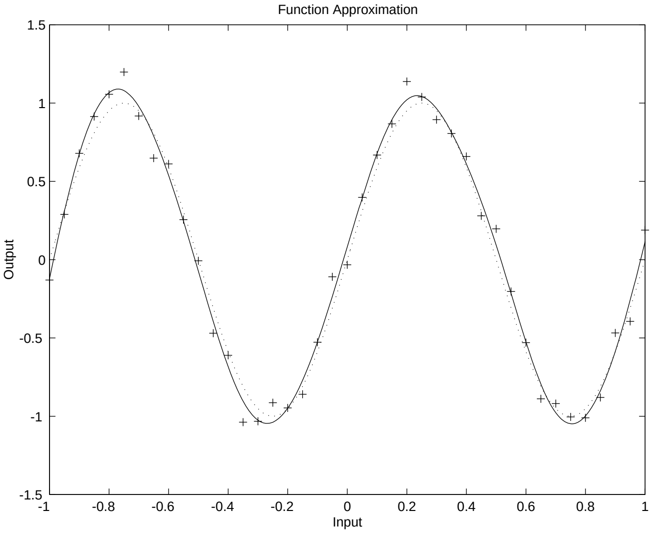
```
p = [-1:.05:1];
t = sin(2*pi*p)+0.1*randn(size(p));
net=newff(p,t,20,{},'trainbr');
net.divideFcn ='';
net.trainParam.show = 10;
net.trainParam.epochs = 50;
randn('seed',192736547);
net = init(net);
[net,tr]=train(net,p,t);
```

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 "Preprocessing and Postprocessing" on page 3-7.

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 `trainrp` 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
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 (postreg)

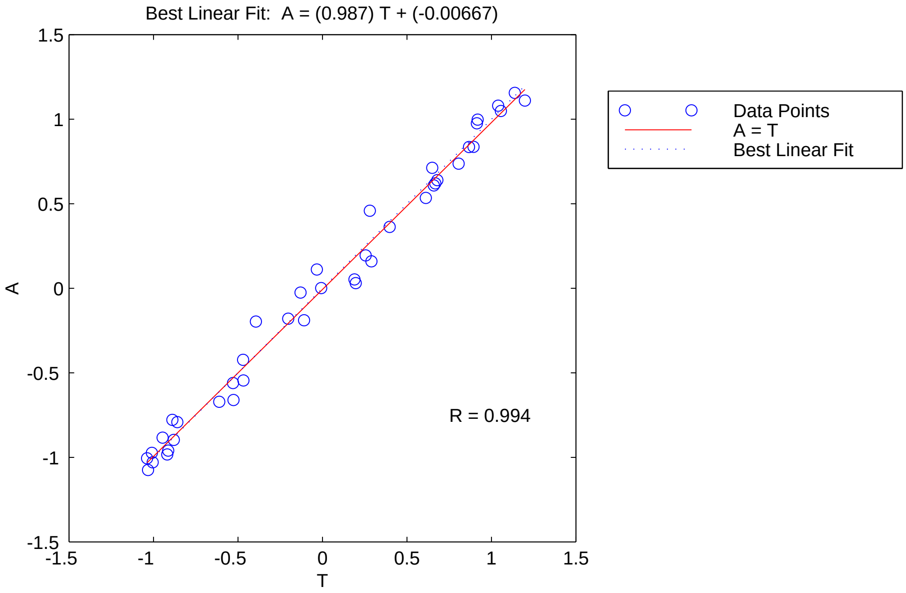
The performance of a trained network can be measured to some extent by the errors on the training, validation, and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine `postreg` is designed to perform this analysis.

The following commands illustrate how to perform a regression analysis on the network trained in "Summary and Discussion of Early Stopping and Regularization" on page 9-40.

```
a = sim(net,p);
[m,b,r] = postreg(a,t)
m =
    0.9874
b =
   -0.0067
r =
    0.9935
```

The network output and the corresponding targets are passed to `postreg`. 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 `postreg` 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 `postreg`. 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.


Best Linear Fit: A = (0.987) T + (-0.00667)
1.5
1
0.5
0
-0.5
-1
-1.5
A
-1.5
-1
-0.5
0
0.5
1
1.5
T
R = 0.994
Data Points
A = T
Best Linear Fit

Custom 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.

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 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 `template_transfer.m`. Rename the new file `mytransfer`. Start editing the file by changing the function name at the top from `template_transfer` 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 directory, and assign the name of your transfer function to the `transferFcn` property of any layer of any network object to put it to use.

Historical Networks

Introduction

This chapter covers networks that are of historical interest, but that are not as actively used today as networks presented in earlier chapters. 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 chapter also covers two recurrent networks: Elman and Hopfield networks.

The perceptron network is 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.

Elman networks are two-layer backpropagation networks, with the addition of a feedback connection from the output of the hidden layer to its input. This feedback path allows Elman networks to learn to recognize and generate temporal patterns, as well as spatial patterns. The best paper on the Elman network is Elman, J.L., “Finding structure in time,” *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

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.

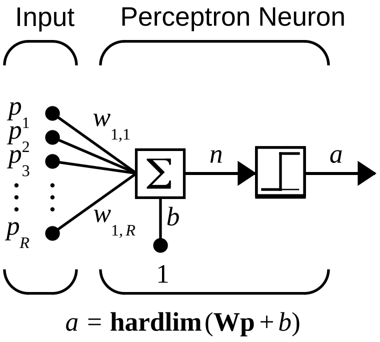
Perceptron Networks

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 chapter 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

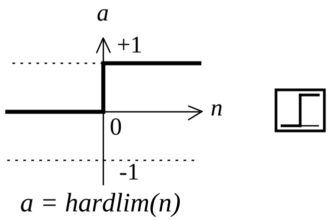
A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



Where

R = number of elements in input vector

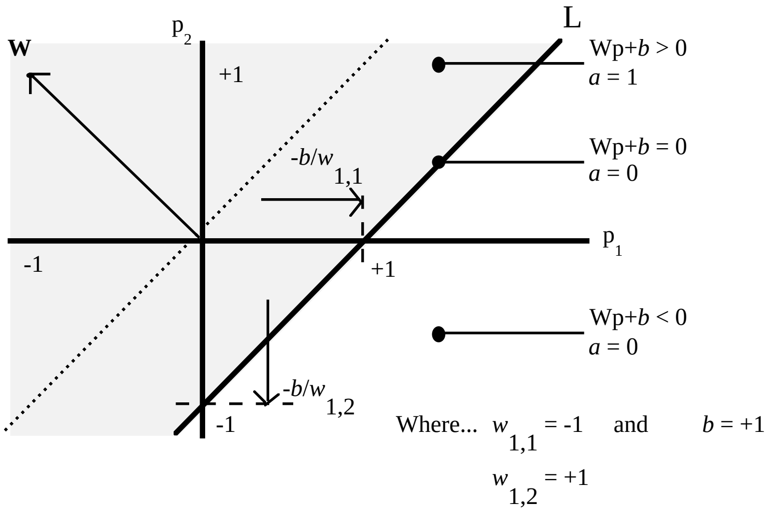
Each external input is weighted with an appropriate weight w_{1j}, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



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 $b = 1$.



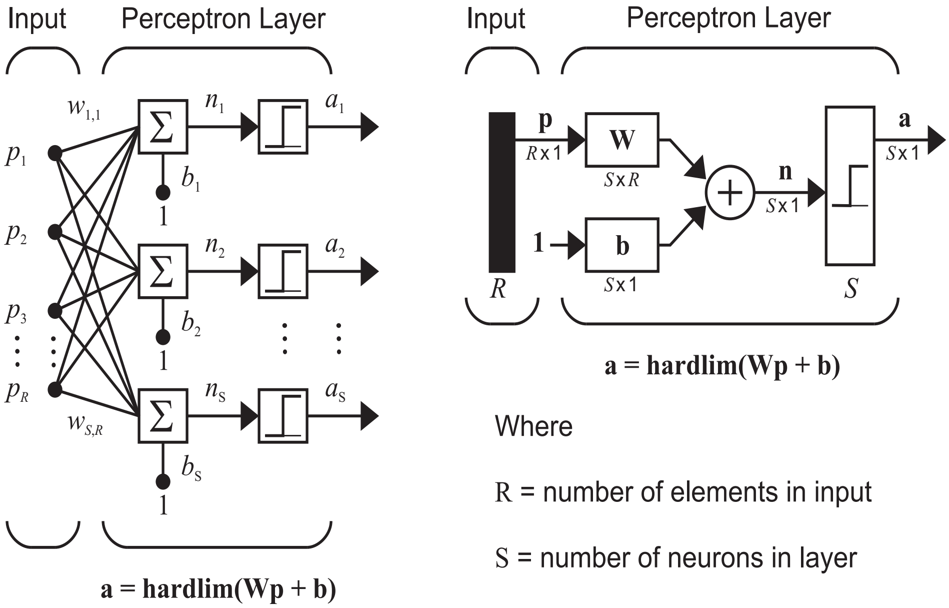
Two classification regions are formed by the *decision boundary* line L at $\mathbf{Wp} + b = 0$. This line is perpendicular to the weight matrix **W** and shifted according to the bias b. Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause 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 demonstration 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 jth input to the ith neuron.



Where

R = number of elements in input

S = number of neurons in layer

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 10-16.

Creating a Perceptron (newp)

A perceptron can be created with the `newp` function:

```
net = newp(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 = newp(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: 1
      learnFcn: 'learnp'
    learnParam: []
          size: [1 1]
      userdata: [1x1 struct]
     weightFcn: 'dotprod'
   weightParam: [1x1 struct]
```

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

$$\mathbf{p}_1\mathbf{t}_1,\mathbf{p}_2\mathbf{t}_1,\ldots, p_Q\mathbf{t}_Q$$

where **p** is an input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response **a** and the target vector **t**. The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector **p** and the associated error **e**. The target vector **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 **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 **w** and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **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 **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 **p** is added to the weight vector **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 **p** is subtracted from the weight vector **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 **e** = **t** – **a** and the change to be made to the weight vector Δ**w**:

CASE 1. If **e** = 0, then make a change Δ**w** equal to 0.

CASE 2. If **e** = 1, then make a change Δ**w** equal to $\mathbf{p}^T$.

CASE 3. If **e** = –1, then make a change Δ**w** equal to $-\mathbf{p}^T$.

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t-a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

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:

$$\Delta b = (t-a)(1) = e$$

For the case of a layer of neurons you have

$$\Delta\mathbf{W} = (\mathbf{t}-\mathbf{a})(\mathbf{p})^{\mathrm{T}} = \mathbf{e}(\mathbf{p})^{\mathrm{T}}$$

and

$$\Delta\mathbf{b} = (\mathbf{t}-\mathbf{a}) = e$$

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. Here are input vectors with the values -2 and 2, and outputs with values 0 and 1.

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

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 = sim(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 demo `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

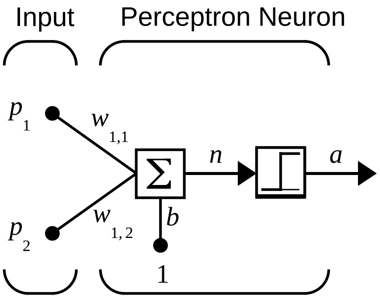
Training (train)

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 **W** and **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 10-16.

To illustrate the training procedure, work through a simple problem. Consider a one-neuron perceptron with a single vector input having two elements:



$$= \mathbf{hardlim}(\mathbf{Wp} + b)$$

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.

$$\mathbf{p}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, t_1 = 0 \Bigg\} \Bigg\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, t_2 = 1 \Bigg\} \Bigg\{ \mathbf{p}_3 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, t_3 = 0 \Bigg\} \Bigg\{ \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_4 = 1$$

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 **W**(0) and b(0).

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0$$

Start by calculating the perceptron's output a for the first input vector $\mathbf{p}_1$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0 \right) = hardlim(0) = 1$$

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.

$$e = t_1 - a = 0 - 1 = -1$$
$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)\begin{bmatrix} 2 & 2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix}$$
$$\Delta b = e = (-1) = -1$$

You can calculate the new weights and bias using the perceptron update rules.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T = \begin{bmatrix} 0 & 0 \end{bmatrix} + \begin{bmatrix} -2 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix} = \mathbf{W}(1)$$

$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

Now present the next input vector, $\mathbf{p}_2$. The output is calculated below.

$$\begin{aligned} a &= hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1)) \\ &= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-2\\-2\end{bmatrix} - 1\right) = hardlim(1) = 1 \end{aligned}$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = \begin{bmatrix}-2 & -2\end{bmatrix}$ and $p(2) = p(1) = -1$.

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 $\mathbf{W}(4) = \begin{bmatrix}-3 & -1\end{bmatrix}$ 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

$$\mathbf{W}(6) = \begin{bmatrix}-2 & -3\end{bmatrix} \text{ and } b(6) = 1$$

This concludes the hand calculation. Now, how can you do this using the `train` function?

The following code defines a perceptron like that shown in the previous figure, with initial weights and bias values of 0.

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

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 = newp([-2 2;-2 2],[0 1]);
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 = sim(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 = sim(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 `newp` is `trainc`. (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 demonstration 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`, which is discussed in detail in the next chapter. 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:

$$\Delta \mathbf{w} = (t-a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

As shown above, the larger an input vector **p**, the larger its effect on the weight vector **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:

$$\Delta \mathbf{w} = (t-a)\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|} = e\frac{\mathbf{p}^{\mathrm{T}}}{\|\mathbf{p}\|}$$

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 Networks

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.

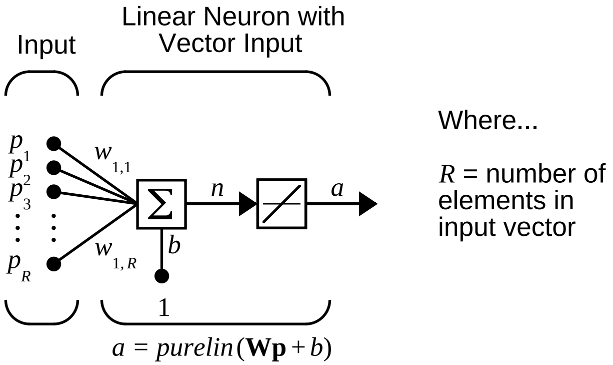
This section introduces `newlin`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

You can type `help linnet` to see a list of linear network functions, demonstrations, and applications.

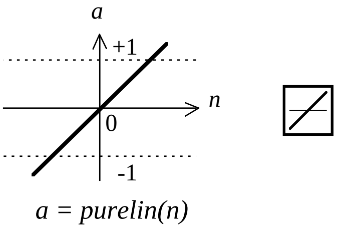
The use of linear filters in adaptive systems is discussed in Chapter 8, "Adaptive Filters and Adaptive Training."

Neuron Model

A linear neuron with R inputs is shown below.


Input
Linear Neuron with Vector Input
p_1
p_2
p_3
p_R
$w_{1,1}$
$w_{1,R}$
Σ
n
a
b
1
$a = purelin(\mathbf{Wp} + b)$
Where...
R = number of elements in input vector

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

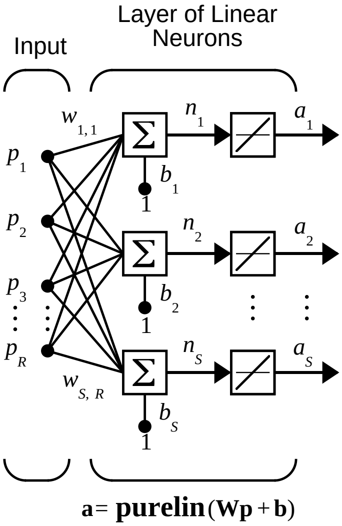
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

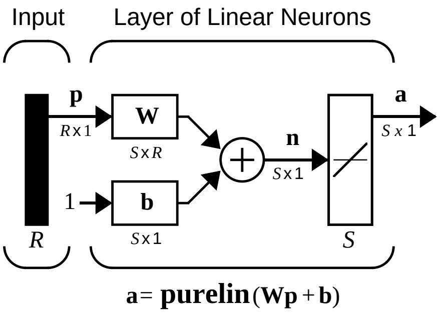
$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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 $\mathbf{W}$.





Where...

R = number of elements in input vector

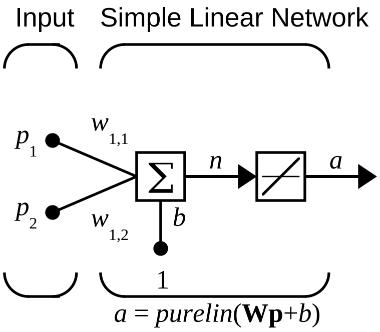
S = number of neurons in layer

Note that the figure on the right defines an *S*-length output vector **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.

Creating a Linear Neuron (newlin)

Consider a single linear neuron with two inputs. The following figure shows the diagram for this network.



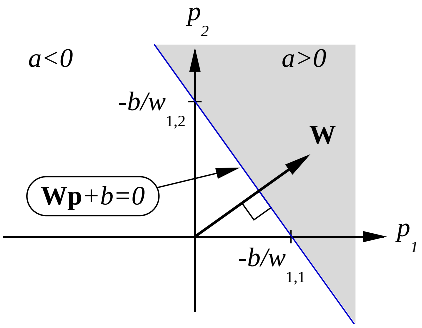
The weight matrix **W** in this case has only one row. The network output is

$$a = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

or

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

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 $\mathbf{Wp} + b = 0$ 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 the following command, which specifies typical input vectors of [-1; -1] and [1; 1] and typical outputs of [-1 1]. (These values are arbitrary. For a real problem, use real values.)

```
net = newlin([-1 1; -1 1],[-1 1]);
```

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 = sim(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:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

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.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

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 = sim(net,P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

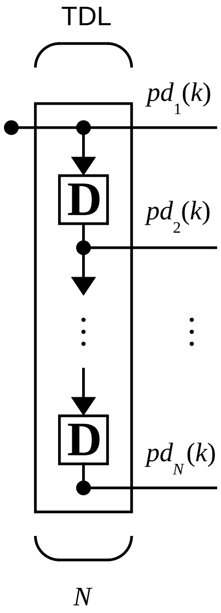
You might try `demolin1`. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

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

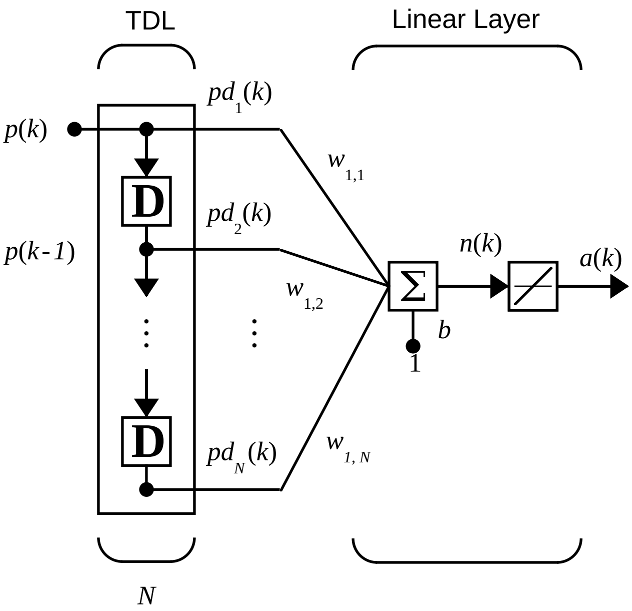
Tapped Delay Line

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

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i}p(k-i+1) + b$$

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 `Pi`.

```
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 = sim(net,P,Pi)
```

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 $j = 1, 2, \ldots, R$ 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.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial[t(k)-a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k)-(\mathbf{Wp}(k)+b)]$$

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 $p_\mathrm{i}(k)$ is the ith element of the input vector at the kth 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 **e** and the bias **b** are vectors, and α is a *learning rate*. If α 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

```
dw = lr*e*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`.

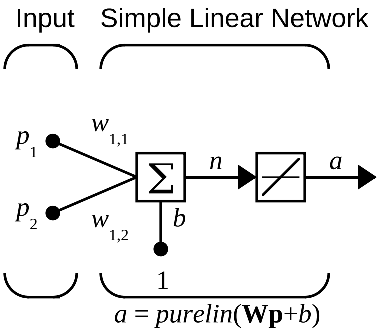
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

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`, discussed in Chapter 8, “Adaptive Filters and Adaptive Training,” 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 = newlin(P,T);
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 = sim(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 demonstration program, demolin2, shows the training of a linear neuron and plots the weight trajectory and error during training.

You might also try running the demonstration program nnd10lc. 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 demonstration `demolin6`, the network cannot solve the problem with zero error. You might want to try `demolin6`.

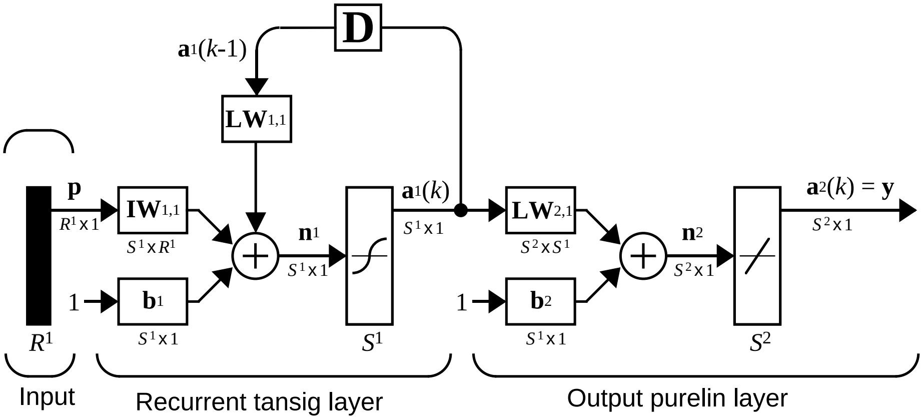
Too Large a Learning Rate

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. Demonstration `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.

Elman Networks

Architecture

The Elman network commonly is a two-layer network with feedback from the first-layer output to the first-layer input. This recurrent connection allows the Elman network to both detect and generate time-varying patterns. A two-layer Elman network is shown below.



$$\mathbf{a}^1(k) = \mathbf{tansig}(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{LW}^{1,1}\mathbf{a}^1(k\text{-}1) + \mathbf{b}^1) \qquad \mathbf{a}^2(k) = \mathbf{purelin}(\mathbf{LW}^{2,1}\mathbf{a}^1(k) + \mathbf{b}^2)$$

The Elman network has `tansig` neurons in its hidden (recurrent) layer, and `purelin` neurons in its output layer. This combination is special in that two-layer networks with these transfer functions can approximate any function (with a finite number of discontinuities) with arbitrary accuracy. The only requirement is that the hidden layer must have enough neurons. More hidden neurons are needed as the function being fitted increases in complexity.

Note that the Elman network differs from conventional two-layer networks in that the first layer has a recurrent connection. The delay in this connection stores values from the previous time step, which can be used in the current time step.

Thus, even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different because of different feedback states.

Because the network can store information for future reference, it is able to learn temporal patterns as well as spatial patterns. The Elman network can be trained to respond to, and to generate, both kinds of patterns.

Creating an Elman Network (newelm)

You can use the function newelm to create an Elman network with two or more layers. The hidden layers commonly have tansig transfer functions and this is the default for newelm. The architecture diagram shows that purelin is commonly the output-layer transfer function.

The default backpropagation training function is trainbfg. You might use trainlm, but it tends to proceed so rapidly that it does not necessarily do well in the Elman network. The backpropagation weight/bias learning function default is learngdm, and the default performance function is mse.

When the network is created, the weights and biases of each layer are initialized with the Nguyen-Widrow layer-initialization method, which is implemented in the function initnw.

For example, consider a sequence of single-element input vectors in the range from 0 to 1 and with outputs in the same range. Suppose that you want to have five hidden-layer tansig neurons and a single logsig output layer. The following command creates this network:

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

Simulation

Suppose that you want to find the response of this network to an input sequence of eight digits that are either 0 or 1:

```
P = round(rand(1,8))
P =
     0     1     0     1     1     0     0     0
```

Recall that a sequence to be presented to a network is to be in cell array form. Convert P to this form:

```
Pseq = con2seq(P)
Pseq =
    [0]    [1]    [0]    [1]    [1]    [0]    [0]    [0]
```

Now you can find the output of the network with the function sim:

```
Y = sim(net,Pseq)
Y =
Columns 1 through 5
   [1.9875e-04]   [0.1146]   [5.0677e-05]   [0.0017]   [0.9544]
Columns 6 through 8
    [0.0014]    [5.7241e-05]    [3.6413e-05]
```

Convert this back to concurrent form with

```
z = seq2con(Y);
```

and display the output in concurrent form with

```
z{1,1}
ans =
  Columns 1 through 7
   0.0002   0.1146   0.0001   0.0017   0.9544   0.0014   0.0001
Column 8
    0.0000
```

Thus, once the network is created and the input specified, you need only call sim.

Training an Elman Network

Elman networks can be trained with either of two functions, train or adapt.

When you use the function train to train an Elman network the following occurs:

At each epoch,

1. The entire input sequence is presented to the network, and its outputs are calculated and compared with the target sequence to generate an error sequence.

2. For each time step, the error is backpropagated to find *gradients* of errors for each weight and bias. This *gradient* is actually an approximation, because the contributions of weights and biases to errors via the delayed recurrent connection are ignored.

3. This gradient is then used to update the weights with the chosen backprop training function. The function traingdx is recommended.

When you use the function adapt to train an Elman network, the following occurs:

At each time step,

1 Input vectors are presented to the network, and it generates an error.

2 The error is backpropagated to find gradients of errors for each weight and bias. This gradient is actually an approximation, because the contributions of weights and biases to the error, via the delayed recurrent connection, are ignored.

3 This approximate gradient is then used to update the weights with the chosen learning function. The function learngdm is recommended.

Elman networks are not as reliable as some other kinds of networks, because both training and adaptation happen using an approximation of the error gradient.

For an Elman to have the best chance at learning a problem, it needs more hidden neurons in its hidden layer than are actually required for a solution by another method. While a solution might be available with fewer neurons, the Elman network is less able to find the most appropriate weights for hidden neurons because the error gradient is approximated. Therefore, having a fair number of neurons to begin with makes it more likely that the hidden neurons will start out dividing up the input space in useful ways.

The function train trains an Elman network to generate a sequence of target vectors when it is presented with a given sequence of input vectors. The input vectors and target vectors are passed to train as matrices P and T. Train takes these vectors and the initial weights and biases of the network, trains the network using backpropagation with momentum and an adaptive learning rate, and returns new weights and biases.

Continue with the example, and suppose that you want to train a network with an input P and targets T as defined below,

```
P = round(rand(1,8))
P =
     1     0     1     1     1     0     1     1
```

and

```
T = [0 (P(1:end-1)+P(2:end) == 2)]
T =
     0     0     0     1     1     0     0     1
```

Here T is defined to be 0, except when two 1's occur in P, in which case T is 1.

As noted previously, the network has five hidden neurons in the first layer.

```
net = newelm(P,T,5,{'tansig','logsig'});
```

Use trainbfg as the training function and train for 100 epochs. After training, simulate the network with the input P and calculate the difference between the target output and the simulated network output.

```
Pseq = con2seq(P);
Tseq = con2seq(T);
net = train(net,Pseq,Tseq);
Y = sim(net,Pseq);
z = seq2con(Y);
z{1,1};
diff1 = T - z{1,1};
```

Note that the difference between the target and the simulated output of the trained network is very small. Thus, the network is trained to produce the desired output sequence on presentation of the input vector **P**.

Hopfield Network

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.

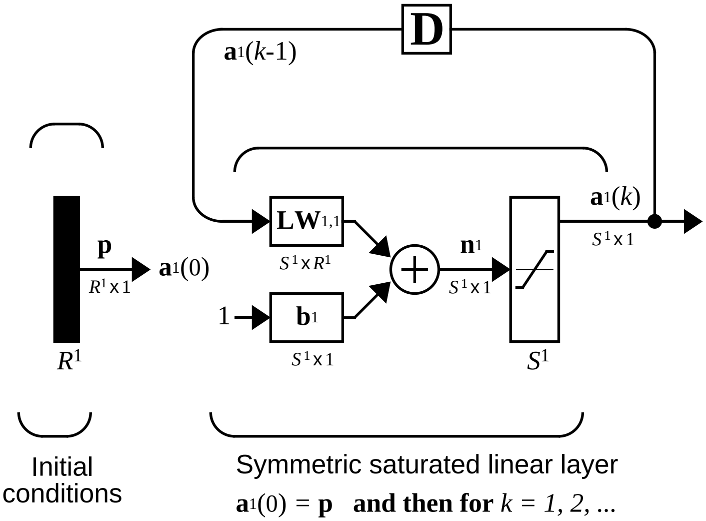
The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space. The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and 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, read Chapter 18 of the *Hopfield Network* [HDB96].

Architecture

The architecture of the Hopfield network follows.

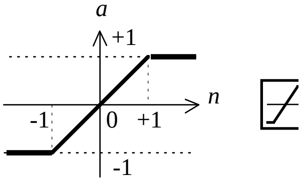


$\mathbf{a}^1(0) = \mathbf{p}$ **and then for** $k = 1, 2, \ldots$

$\mathbf{a}^1(k) = \mathbf{satlins}(\mathbf{LW}^{1,1}\mathbf{a}^1(k-1)) + \mathbf{b}^1)$

As noted, the *input* **p** to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function `satlins`.



a = satlins(n)

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 **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 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 second argument to `sim` below is Q = 2 when you are using matrix notation.)

```
Ai = T;
[Y,Pf,Af] = sim(net,2,[],Ai);
Y
```

This gives you

```
Y =
    -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. Note, incidentally, that the original point was specified in cell array form. This allows you to run the network for more than one step.

```
[Y,Pf,Af] = sim(net,{1 5},{},Ai);
Y{1}
```

This produces

```
ans =
    -1
    -1
     1
```

Thus, an original condition close to a design point did converge to that point.

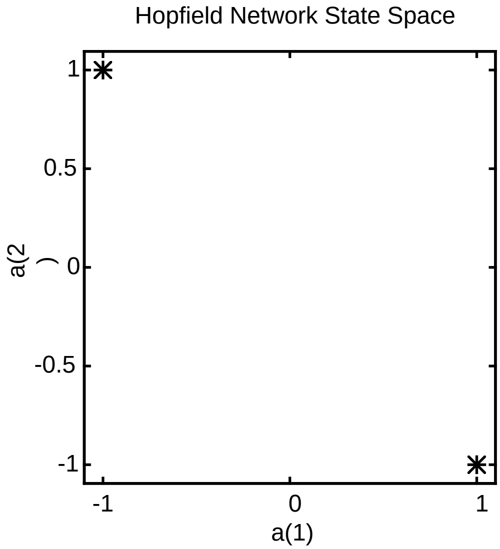
This is, of course, the hope for all such inputs. Unfortunately, even the best known Hopfield designs occasionally include spurious undesired stable points that attract the solution.

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 **T**.

```
T = [1 -1; -1 1]'
T =
     1    -1
    -1     1
```

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

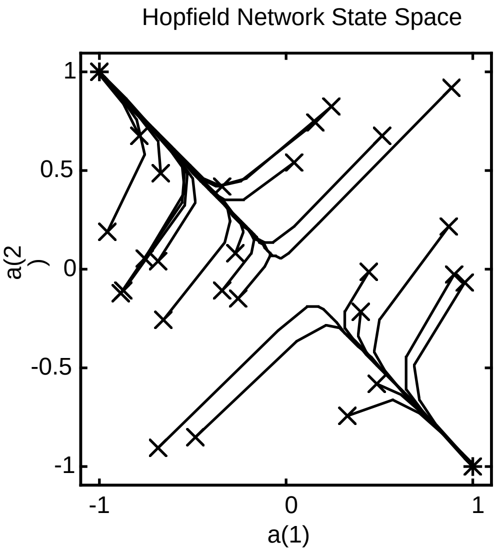
```
b =
     0
     0
```

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] = sim(net,2,[],Ai);
Y

Y =
      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 demonstration `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 demonstrations `demohop2` and `demohop4` to investigate these issues.

11

Network Object Reference

Network Properties

These properties define the basic features of a network. “Subobject Properties” on page 11-13 describes properties that define network details.

General

Here are two 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™ software users:

```
net.userdata.note
```

Efficiency

Here are two efficiency properties of neural networks.

net.efficiency.cacheDelayedInput

This property can be set to true (the default) or false. If true then the delayed inputs of each input weight are calculated once during training and reused, instead of recalculated each time they are needed. This results in faster training, but at the expense of memory efficiency. For greater memory efficiency set this property to false.

net.efficiency.flattenTime

This property can be set to true (the default) or false. If true then time series data used to train static networks will be reformatted as static data before training. This results in faster training at the expense of memory efficiency. For greater memory efficiency, either only use static data for static networks, or set this property to false.

net.efficiency.memoryReduction

This property can be set to 1 (the default) or any integer greater than 1. If set to an integer N, then simulation and error gradient and Jacobian calculations will be split in time into N subcalculations by groups of samples. This will result in greater time overhead but result in reduced memory requirements for storing intermediate values. For greater memory efficiency set this to higher values.

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 ith 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 x 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 ith layer from the jth 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 x 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 x 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 "Subobject Properties" on page 11-13.

net.inputs

This property holds structures of properties for each of the network's inputs. It is always an N_i x 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}
```

Input Properties. See “Inputs” on page 11-13 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 x 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 11-15 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 x 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).

Output Properties. See “Outputs” on page 11-20 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 x 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 11-22 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 x 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 11-23 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 x 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 11-25 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 us 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)
```

For a list of functions, type

```
help nninit
```

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 launched 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. The training function 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 x 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 x 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 x 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
```

Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

Inputs

These properties define the details of each *i*th network input.

net.inputs{1}.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 the `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 the `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 the `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 x 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 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 nndistance
```

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 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.

For a list of functions, type

```
help nninit
```

net.layers{i}.netInputFcn

This property defines which of the 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)

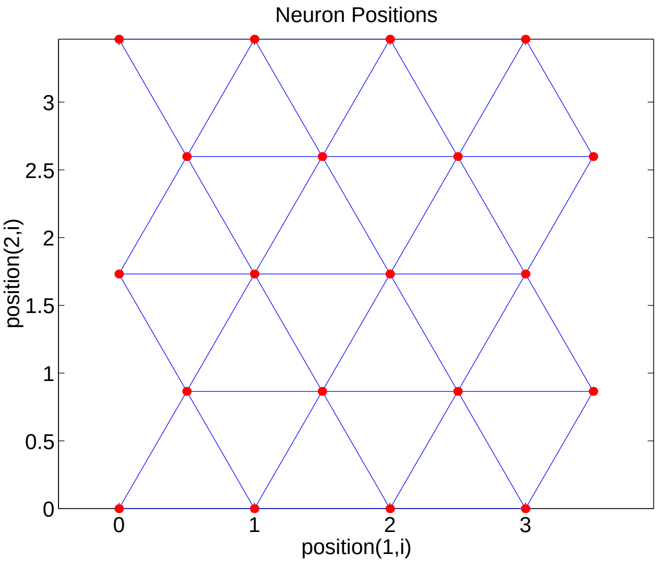
This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

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 x 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 are used to calculate the *i*th layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

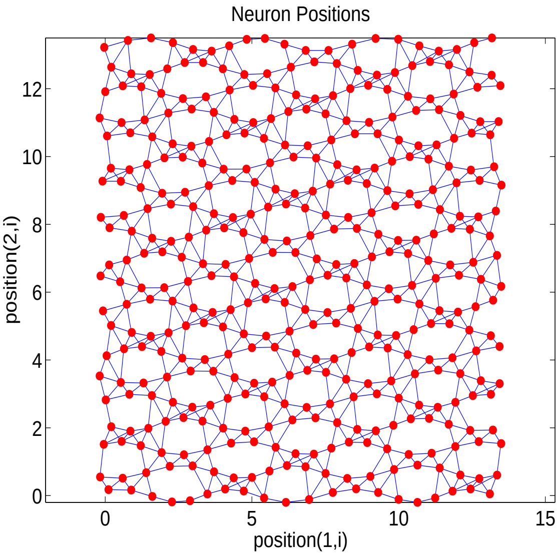
For a list of functions, type

```
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 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 `preparets` 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 the `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 the `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 the `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 the `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 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`.

For a list of functions, type

```
help nninit
```

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 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 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.

For a list of functions, type

```
help nninit
```

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 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 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 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.

For a list of functions, type

```
help nninit
```

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 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 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.

Function Reference

“Transfer Functions” on page 12-21	Transform output of network layer
“Weight and Bias Initialization Functions” on page 12-22	Internal functions for network computations
“Weight and Bias Initialization Functions” on page 12-22	Initialize weights and biases
“Weight Functions” on page 12-23	Convolution, dot product, scalar product, and distances weight functions

DataFunctions

`catelements`	Concatenate neural network data elements
`catsamples`	Concatenate neural network data samples
`catsignals`	Concatenate neural network data signals
`cattimesteps`	Concatenate neural network data timesteps
`cellmat`	Create a cell array of matrices
`combvec`	Create all combinations of vectors
`con2seq`	Convert concurrent vectors to sequential vectors
`concur`	Create concurrent bias vectors
`confusion`	Classification confusion matrix
`errsurf`	Error surface of single-input neuron
`extendts`	Extend time seriess data to a given number of timesteps
`fromnndata`	Convert data from standard neural network cell array form
`gadd`	Generalized addition
`gdivide`	Generalized division
`getelements`	Get neural network data elements
`getsamples`	Get neural network data samples
`getsignals`	Get neural network data signals
`gettimesteps`	Get neural network data timesteps
`gmultiply`	Generalized multiply
`gnegate`	Generalized negate
`gsqrt`	Generalized square root
`gsubtract`	Generalized subtract
`ind2vec`	Convert indices to vectors
`maxlinlr`	Maximum learning rate for a linear layer
`meanabs`	Mean of absolute elements of a matrix or matrices

`meansqr`	Mean of squared elements of a matrix or matrices
`minmax`	Ranges of matrix rows
`nncell2mat`	Combines neural network cell data into a matrix
`nncorr`	Cross-correlation between neural time series
`nndata`	Create neural network data
`nndata2sim`	Convert neural network data to Simulink time series
`nnsize`	Number of neural data elements, samples, timesteps and signals
`normc`	Normalize columns of a matrix or matrices
`normr`	Normalize rows of a matrix or matrices
`numelements`	Number of elements in neural network data
`numfinite`	Number of finite elements in neural network data
`numnan`	Number of NaN elements in neural network data
`numsamples`	Number of samples in neural network data
`numsignals`	Number of signals in neural network data
`numtimesteps`	Number of timesteps in neural network data
`plotep`	Plot a weight-bias position on an error surface
`plotes`	Plot the error surface of a single input neuron
`plotpc`	Plot a classification line on a perceptron vector plot
`plotpv`	Plot perceptron input/target vectors
`plotv`	Plot vectors as lines from the origin
`plotvec`	Plot vectors with different colors
`pnormc`	Psuedo-normalize columns of a matrix
`preparets`	Prepare time series data for network simulation or training
`prunedata`	Prune data for a pruned network
`quant`	Discretize neural network data as multiples of a quantity
`regression`	Linear regression

`roc`	Reciever Operating Characteristic
`seq2con`	Convert sequential vectors to concurrent vectors
`setelements`	Set neural network data elements
`setsamples`	Set neural network data samples
`setsignals`	Set neural network data signals
`settimesteps`	Set neural network data timesteps
`sim2nndata`	Convert Simulink time series to neural network data
`sumabs`	Sum of absolute elements of a matrix or matrices
`sumsqr`	Sum of squared elements of matrix or matrices
`tapdelay`	Shift neural network time series data for a tap delay
`tonndata`	Convert data to standard neural network cell array form
`vec2ind`	Convert vectors to indices

Distance Functions

`boxdist`	Distance between two position vectors
`dist`	Euclidean distance weight function
`linkdist`	Link distance function
`mandist`	Manhattan distance weight function

Graphical Interface Functions

`nctool`	Neural network classification tool
`nftool`	Open Neural Network Fitting Tool
`nntool`	Open Network/Data Manager
`nntraintool`	Neural network training tool
`nprtool`	Neural network pattern recognition tool
`ntstool`	Neural network time series tool
`view`	View neural network

Layer Initialization Functions

`initnw`	Nguyen-Widrow layer initialization function
`initwb`	By-weight-and-bias layer initialization function

Learning Functions

`learncon`	Conscience bias learning function
`learngd`	Gradient descent weight/bias learning function
`learngdm`	Gradient descent with momentum weight/bias learning function
`learnh`	Hebb weight learning function
`learnhd`	Hebb with decay weight learning rule
`learnis`	Instar weight learning function
`learnk`	Kohonen weight learning function
`learnlv1`	LVQ1 weight learning function
`learnlv2`	LVQ2 weight learning function
`learnos`	Outstar weight learning function
`learnp`	Perceptron weight and bias learning function
`learnpn`	Normalized perceptron weight and bias learning function
`learnsom`	Self-organizing map weight learning function
`learnsomb`	Batch self-organizing map weight learning function
`learnwh`	Widrow-Hoff weight and bias learning rule

Line Search Functions

`srchbac`	1-D minimization using backtracking search
`srchbre`	1-D interval location using Brent's method
`srchcha`	1-D minimization using Charalambous' method
`srchgol`	1-D minimization using golden section search
`srchhyb`	1-D minimization using hybrid bisection/cubic search

Net Input Functions

`netprod`	Product net input function
`netsum`	Sum net input function

Network Initialization Function

`init`	Initialize neural network
`initlay`	Layer-by-layer network initialization function

Network Use Functions

`adapt`	Allow neural network to change weights and biases on inputs
`adddelay`	Add a delay to a neural network's response
`closeloop`	Convert neural network open loop feedback to closed loop
`configure`	Configure neural network inputs and outputs
`disp`	Neural network properties
`display`	Name and properties of neural network variables
`formwb`	Form bias and weights into a single vector
`getwb`	Get all network weight and bias values as a single vector
`init`	Initialize neural network
`isconfigured`	Is neural network configured?
`noloop`	Remove neural network open and closed feedback loops
`openloop`	Convert neural network closed loop feedback to open loop
`perform`	Neural network performance
`prune`	Delete neural inputs, layers and outputs with sizes of zero
`removedelay`	Remove a delay from a neural network's response
`separatewb`	Separate biases and weights from a weight/bias vector
`setwb`	Set all network weight and bias values with a single vector
`sim`	Simulate neural network
`train`	Train neural network
`unconfigure`	Unconfigured neural network inputs and outputs
`view`	View a neural network

New Networks Functions

`network`	Create custom neural network
`cascadeforwardnet`	Cascade-forward neural network
`competlayer`	Competitive neural layer
`distdelaynet`	Distributed delay neural network
`elmannet`	Elman neural network
`feedforwardnet`	Feed-forward neural network
`fitnet`	Function fitting neural network
`layrecnet`	Layer recurrent neural network
`linearlayer`	Linear neural layer
`lvqnet`	Learning vector quantization (LVQ) neural network
`narnet`	Nonlinear auto-associative time series network
`narxnet`	Nonlinear auto-associative time series network with external input
`newgrnn`	Generalized regression neural network
`newlind`	Designed linear layer
`newpnn`	Probabilistic neural network
`newrb`	Radial basis network
`newrbe`	Exact radial basis network
`patternnet`	Pattern recognition network
`perceptron`	Perceptron
`selforgmap`	Self-organizing map
`timedelaynet`	Time-delay neural network

Performance Functions

`mae`	Mean absolute error performance function
`mse`	Mean squared error performance function
`sse`	Sum squared error performance function

Plotting Functions

`plotconfusion`	Plot classification confusion matrix
`plotep`	Plot weight and bias position on error surface
`ploterrcorr`	Plot autocorrelation of error time series
`ploterrhist`	Plot error histogram
`plotes`	Plot error surface of single-input neuron
`plotfit`	Plot function fit
`plotinerrcorr`	Plot input to error time series cross-correlation
`plotpc`	Plot classification line on perceptron vector plot
`plotperform`	Plot network performance
`plotpv`	Plot perceptron input target vectors
`plotregression`	Plot linear regression
`plotroc`	Plot receiver operating characteristic
`plotsomhits`	Plot self-organizing map sample hits
`plotsomnc`	Plot self-organizing map neighbor connections
`plotsomnd`	Plot self-organizing map neighbor distances
`plotsomplanes`	Plot self-organizing map weight planes
`plotsompos`	Plot self-organizing map weight positions
`plotsomtop`	Plot self-organizing map topology
`plottrainstate`	Plot training state values
`plotv`	Plot vectors as lines from origin
`plotvec`	Plot vectors with different colors
`plotwb`	Plot Hinton diagram of weight and bias values

Processing Functions

`fixunknowns`	Process data by marking rows with unknown values
`lvqoutputs`	Define settings for LVQ outputs
`mapminmax`	Process matrices by mapping row minimum and maximum values to `[-1 1]`
`mapstd`	Process matrices by mapping each row's means to 0 and deviations to 1
`processpca`	Process columns of matrix with principal component analysis
`removeconstantrows`	Process matrices by removing rows with constant values
`removerows`	Process matrices by removing rows with specified indices

Simulink® Support Function

`getsiminit`	Get neural network Simulink block initial conditions
`gensim`	Generate Simulink block for neural network simulation
`nndata2sim`	Convert neural network data to Simulink time series
`setsiminit`	Set neural network Simulink block initial conditions
`sim2nndata`	Convert Simulink time series to neural network data

Topology Functions

`gridtop`	Grid layer topology function
`hextop`	Hexagonal layer topology function
`randtop`	Random layer topology function
`tritop`	Triangle layer topology function

Training Functions

`train`	Train neural network
`trainb`	Batch training with weight and bias learning rules
`trainbfg`	BFGS quasi-Newton backpropagation
`trainbfgc`	BFGS quasi-Newton backpropagation for use with NN model reference adaptive controller
`trainbr`	Bayesian regularization
`trainbu`	Batch unsupervised weight/bias training
`trainc`	Cyclical order incremental update
`traincgb`	Powell-Beale conjugate gradient backpropagation
`traincgf`	Fletcher-Powell conjugate gradient backpropagation
`traincgp`	Polak-Ribiére conjugate gradient backpropagation
`traingd`	Gradient descent backpropagation
`traingda`	Gradient descent with adaptive learning rule backpropagation
`traingdm`	Gradient descent with momentum backpropagation
`traingdx`	Gradient descent with momentum and adaptive learning rule backpropagation
`trainlm`	Levenberg-Marquardt backpropagation
`trainoss`	One step secant backpropagation
`trainr`	Random order incremental training with learning functions
`trainrp`	Resilient backpropagation (Rprop)
`trainru`	Random order unsupervised weight/bias training
`trains`	Sequential order incremental training with learning functions
`trainscg`	Scaled conjugate gradient backpropagation

Transfer Functions

`compet`	C	Competitive transfer function
`hardlim`		Hard-limit transfer function
`hardlims`		Symmetric hard-limit transfer function
`logsig`		Log-sigmoid transfer function
`netinv`		Inverse transfer function
`poslin`		Positive linear transfer function
`purelin`		Linear transfer function
`radbas`		Radial basis transfer function
`radbasn`		Normalized radial basis transfer function
`satlin`		Saturating linear transfer function
`satlins`		Symmetric saturating linear transfer function
`softmax`	S	Softmax transfer function
`tansig`		Hyperbolic tangent sigmoid transfer function
`tribas`		Triangular basis transfer function

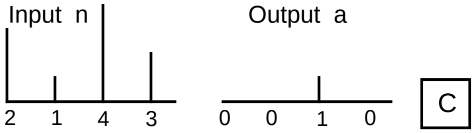
Weight and Bias Initialization Functions

`initcon`	Conscience bias initialization function
`initlvq`	LVQ weight initialization function
`initsompc`	Initialize SOM weights with principal components
`initzero`	Zero weight and bias initialization function
`midpoint`	Midpoint weight initialization function
`randnc`	Normalized column weight initialization function
`randnr`	Normalized row weight initialization function
`rands`	Symmetric random weight/bias initialization function
`randsmall`	Small random weight/bias initialization function

Weight Functions

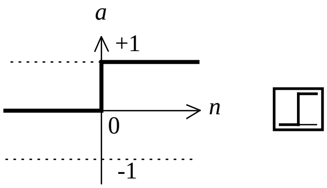
`convwf`	Convolution weight function
`dist`	Euclidean distance weight function
`dotprod`	Dot product weight function
`mandist`	Manhattan distance weight function
`negdist`	Negative distance weight function
`normprod`	Normalized dot product weight function
`scalprod`	Scalar product weight function

Transfer Function Graphs



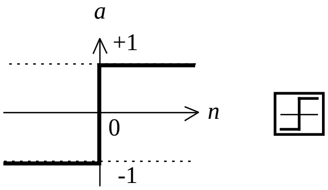
a = compet(n)

Compet Transfer Function



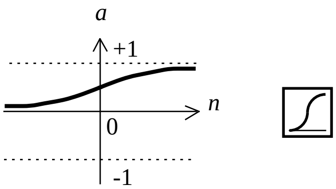
a = hardlim(n)

Hard-Limit Transfer Function



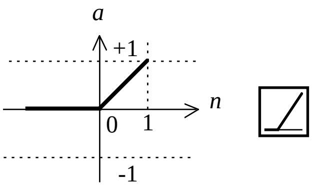
a = hardlims(n)

Symmetric Hard-Limit Transfer Function



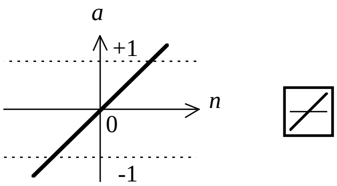
a = logsig(n)

Log-Sigmoid Transfer Function



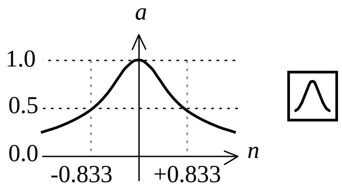
a = poslin(n)

Positive Linear Transfer Function



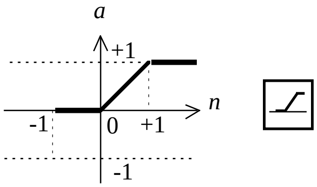
a = purelin(n)

Linear Transfer Function



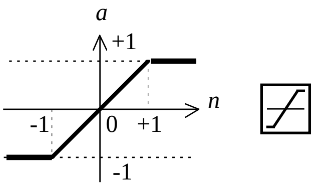
a = radbas(n)

Radial Basis Function



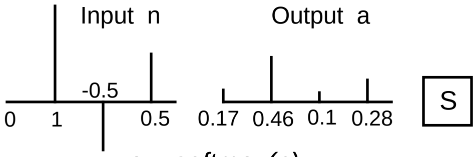
a = satlin(n)

Satlin Transfer Function



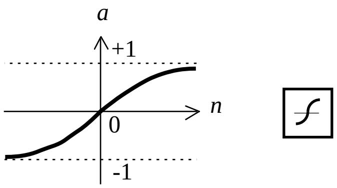
a = satlins(n)

Satlins Transfer Function



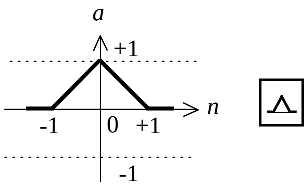
a = softmax(n)

Softmax Transfer Function



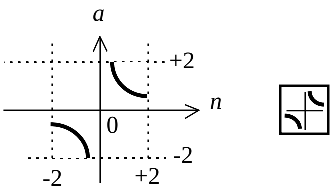
a = tansig(n)

Tan-Sigmoid Transfer Function



a = tribas(n)

Triangular Basis Function



a = netinv(n)

Netinv Transfer Function

13

Functions — Alphabetical List

Purpose

Adapt neural network to data as it is simulated

Syntax

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

To Get Help

Type `help network/adapt`.

Description

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

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

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

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

`net`	Updated network
`Y`	Network outputs
`E`	Network errors
`Pf`	Final input delay conditions
`Af`	Final layer delay conditions
`tr`	Training record (`epoch` and `perf`)

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

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

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

P	Ni x TS cell array	Each element P{i,ts} is an Ri x Q matrix.
T	Nt x TS cell array	Each element T{i,ts} is a Vi x Q matrix.
Pi	Ni x ID cell array	Each element Pi{i,k} is an Ri x Q matrix.
Ai	Nl x LD cell array	Each element Ai{i,k} is an Si x Q matrix.
Y	No x TS cell array	Each element Y{i,ts} is a Ui x Q matrix.
E	No x TS cell array	Each element E{i,ts} is a Ui x Q matrix.
Pf	Ni x ID cell array	Each element Pf{i,k} is an Ri x Q matrix.
Af	Nl x LD cell array	Each element Af{i,k} is an Si x 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) x Q matrix
T	(sum of Vi) x Q matrix
Pi	(sum of Ri) x (ID*Q) matrix
Ai	(sum of Si) x (LD*Q) matrix
Y	(sum of Ui) x Q matrix
E	(sum of Ui) x Q matrix
Pf	(sum of Ri) x (ID*Q) matrix
Af	(sum of Si) x (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 0.5. The linear layer is then simulated.

```
net = linearlayer([0 1],0.5);
```

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];
net.adaptParam.passes = 100;
[net,y,e] = adapt(net,p3,t3);
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.

Algorithm

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

Purpose

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.

`adapt(net,Pd,T,Ai)`

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

Returns

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

Examples

Linear layers use this adaption function. Here a linear layer with input delays of 0 and 1, and a learning rate of 0.5, is created and adapted to produce some target data `t` when given some input data `x`. The response is then plotted, showing the network's error going down over time.

```
x = {-1  0 1 0 1 1 -1  0 -1 1 0 1};
t = {-1 -1 1 1 1 2  0 -1 -1 0 1 1};
net = linearlayer([0 1],0.5);
net.adaptFcn
[net,y,e,xf] = adapt(net,x,t);
plotresponse(t,y)
```

See Also

`adapt`

adddelay

Purpose

Add delay to neural network's response

Syntax

```
net = adddelay(net)
net = adddelay(net,delay)
```

Description

`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 which 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

Here a time delay network is created, trained and simulated in its original form on an input time series `X` and target series `T`. It is then simulated with a delay removed and then added back. These first and third outputs will be identical, while the second will be shifted by one timestep.

```
[X,T] = simpleseries_dataset;
net = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi);
y1 = net(Xs)
net2 = removedelay(net);
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);
y2 = net2(Xs,Xi)
net3 = adddelay(net2)
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi)
```

See Also

`closeloop`, `openloop`, `removedelay`

boxdist

Purpose

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.

`boxdist(pos)` takes one argument,

`pos`   `N` x `S` matrix of neuron positions

and returns the `S` x `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

You can create a standard network that uses `boxdist` as a distance function by calling `newsom`.

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`.

Algorithm

The box distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is

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

See Also

`sim`, `dist`, `mandist`, `linkdist`

bttderiv

Purpose

Backpropagation through time derivative function

Syntax

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

Description

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

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

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

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

Purpose

Cascade-forward neural network

Syntax

`cascadeforwardnet(hiddenSizes,trainFcn)`

Description

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

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

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

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

and returns a new cascade-forward neural network.

Examples

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

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

See Also

`feedforwardnet`

catelements

Purpose

Concatenate neural network data elements

Syntax

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

Description

`catelements` 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

Purpose

Concatenate neural network data samples

Syntax

```
catsamples(x1,x2,...,xn)
catsamples(x1,x2,...,'pad')
catsamples(x1,x2,...,'pad',v)
```

Description

`catsamples` 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

Purpose

Concatenate neural network data signals

Syntax

```
catsignals(x1,x2,...,xn)
```

Description

`catsignals` 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`

Purpose

Concatenate neural network data timesteps

Syntax

```
cattimesteps(x1,x2,...,xn)
```

Description

cattimesteps 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

Purpose

Create cell array of matrices

Syntax

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

Purpose

Convert neural network open-loop feedback to closed loop

Syntax

```
net = closeloop(net)
```

Description

`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).

Examples

Here a NARX network is designed in open-loop form and then converted 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);
```

See Also

`noloop`, `openloop`

combvec

Purpose

Create all combinations of vectors

Syntax

```
combvec(a1,a2...)
```

Description

`combvec(A1,A2...)` takes any number of inputs,

`A1`	Matrix of `N1` (column) vectors
`A2`	Matrix of `N2` (column) vectors

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

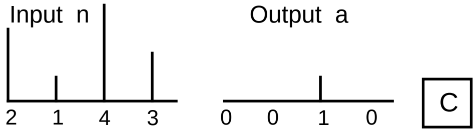
Examples

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

Purpose

Competitive transfer function

Graph and Symbol



a = compet(n)

Compet Transfer Function

Syntax

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

Description

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

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

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

and returns the `S` x `Q` matrix `A` with a `1` in each column where the same column of `N` has its maximum value, and `0` elsewhere.

`compet('dn',N,A,FP)` returns the 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`.

`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

Purpose

Competitive layer

Syntax

`competlayer(numClasses,kohonenLR,conscienceLR)`

Description

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

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

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

and returns a competitive layer with `numClasses` neurons.

Examples

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

```
inputs = iris_dataset;
net = competlayer(6);
net = train(net,inputs);
view(net)
outputs = net(inputs);
classes = vec2ind(outputs);
```

See Also

`selforgmap`, `patternnet`, `lvqnet`

con2seq

Purpose

Convert concurrent vectors to sequential vectors

Syntax

```
s = con2seq(b)
```

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.

`con2seq(b)` takes one input,

`b`	`R` x `TS` matrix

and returns one output,

`S`	`1` x `TS` cell array of `R` x `1` vectors

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

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

and returns

`S`	`N` x `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`

Purpose

Create concurrent bias vectors

Syntax

`concur(B,Q)`

Description

`concur(B,Q)`

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

Returns an `S` x `B` matrix of copies of `B` (or `Nl` x `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` x `1` vectors. However, if the network is being simulated by `sim` (or `adapt` or `train`) in response to `Q` concurrent vectors, then `Z1` and `Z2` will be `S` x `Q` matrices. Before `B` can be combined with `Z1` and `Z2`, you must make `Q` copies of it.

```
n = netsum(z1,z2,concur(b,q))
```

See Also

`netsum`, `netprod`, `sim`, `seq2con`, `con2seq`

configure

Purpose

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

`configure(net,x,t)` takes input data `x` and target data `t`, and configures the network's inputs and outputs to match.

`configure(net,x)` configures only inputs.

`configure(net,'inputs',x,i)` configures the inputs specified with the index vector `i`. If `i` is not specified all inputs are configured.

`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`, `unconfigure`, `init`, `train`

Purpose

Classification confusion matrix

Syntax

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

Description

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

`targets`	`S` x `Q` matrix, where each column vector contains a single `1` value, with all other elements `0`. The index of the `1` indicates which of `S` categories that vector represents.
`outputs`	`S` x `Q` matrix, where each column contains values in the range `[0.1]`. The index of the largest element in the column indicates which of `S` categories that vector represents.

and returns these values:

`c`	Confusion value = fraction of samples misclassified
`cm`	`S` x `S` confusion matrix, where `cm(i,j)` is the number of samples whose target is the `i`th class that was classified as `j`
`ind`	`S` x `S` cell array, where `ind{i,j}` contains the indices of samples with the `i`th target class, but `j`th output class
`per`	`S` x `3` matrix, where each `i`th row represents the percentage of false negatives, false positives, and true positives for the `i`th category

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

`targets`	`1` x `Q` vector of 1/0 values representing membership
`outputs`	`S` x `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` x `2` confusion matrix

`ind`	2 x 2 cell array, where `ind{i,j}` contains the indices of samples whose target is 1 versus 0, and whose output was greater than or equal to `0.5` versus less than `0.5`
`per`	2 x 3 matrix where each ith row represents the percentage of false negatives, false positives, and true positives for the class and out-of-class

Examples

```
load simpleclass_dataset
net = newpr(simpleclassInputs,simpleclassTargets,20);
net = train(net,simpleclassInputs,simpleclassTargets);
simpleclassOutputs = sim(net,simpleclassInputs);
[c,cm,ind,per] = ...
confusion(simpleclassTargets,simpleclassOutputs)
```

See Also

`plotconfusion`, `roc`

Purpose

Convolution weight function

Syntax

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

Description

`convwf` is the convolution weight function. Weight functions apply weights to an input to get weighted inputs.

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

`'deriv'`	Name of derivative function
`'fullderiv'`	Reduced derivative = 2, full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'wfullderiv'`	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

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

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

`convwf('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(4,1);
P = rand(8,1);
```

```
Z = convwf(W,P)
```

Network Use

To change a network so an input weight uses `convwf`, set `net.inputWeight{i,j}.weightFcn` to `'convwf'`. For a layer weight, set `net.layerWeight{i,j}.weightFcn` to `'convwf'`.

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

defaultderiv

Purpose

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.

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

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

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).

`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 = fpderiv('dperf_dwb',net,x,t)
jwb = fpderiv('de_dwb',net,x,t)
```

See Also bttderiv, defaultderiv, num2deriv, num5deriv, staticderiv

Purpose

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

Purpose 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`

Purpose

Euclidean distance weight function

Syntax

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

Description

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

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

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

and returns the `S` x `Q` matrix of vector distances.

`dist(code)` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'fullderiv'`	Full derivative = 1, linear derivative = 0
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

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

`dist('dp',W,P,Z,FP)` returns the derivative of `Z` with respect to `P`.

`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.

dist(pos) takes one argument,

pos	N x S matrix of neuron positions

and returns the S x 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 = 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.inputWeight{i,j}.weightFcn to 'dist'.
For a layer weight, set net.layerWeight{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.

Algorithm

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

Purpose

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.

`timedelaynet(delays,hiddenSizes,trainFcn)` takes these arguments,

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

and returns a distributed delay neural network.

Examples

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

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

See Also

`preparets`, `removedelay`, `timedelaynet`, `narnet`, `narxnet`

divideblock

Purpose

Divide targets into three sets using blocks of indices

Syntax

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

Description

`divideblock` is used to separate targets into three sets: training, validation, and testing. It takes the following inputs:

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

and returns

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

Examples

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

Network Use

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

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

See Also

`divideind`, `divideint`, `dividerand`, `dividetrain`

divideind

Purpose

Divide targets into three sets using specified indices

Syntax

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

Description

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

It takes the following inputs,

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

and returns

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

Examples

```
[trainInd,valInd,testInd] =
divideind(3000,1:12000,2001:2500,2501:3000);
```

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

Purpose

Divide targets into three sets using interleaved indices

Syntax

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

Description

`divideint` is used to separate targets into three sets: training, validation, and testing. It takes the following inputs,

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

and returns

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

Examples

```
[trainInd,valInd,testInd] = divideint(3000,0.6,0.2,0.2);
```

Network Use

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

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

See Also

`divideblock`, `divideind`, `dividerand`, `dividetrain`

Purpose

Divide targets into three sets using random indices

Syntax

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

Description

`dividerand` is used to separate targets into three sets: training, validation, and testing. It takes the following inputs,

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

and returns

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

Examples

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

Network Use

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

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

See Also

`divideblock`, `divideind`, `divideint`, `dividetrain`

dividetrain

Purpose

Assign all targets to training set

Syntax

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

Description

`dividetrain` is used to assign all targets to the training set and no targets to either the validation or test sets. It takes the following inputs,

`Q`	Number of targets to divide up.

and returns

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

Examples

```
[trainInd,valInd,testInd] = dividetrain(3000);
```

Network Use

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

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

See Also

`divideblock`, `divideind`, `divideint`, `dividerand`

Purpose

Dot product weight function

Syntax

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

Description

`dotprod` is the dot product weight function. Weight functions apply weights to an input to get weighted inputs.

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

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

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

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

`'deriv'`	Name of derivative function
`'pfullderiv'`	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'wfullderiv'`	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

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

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

`dotprod('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 = dotprod(W,P)
```

Network Use

You can create a standard network that uses dotprod by calling newp or newlin.

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

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

See help newp and help newlin for simulation examples.

See Also

sim, dist, negdist, normprod

Purpose

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

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

and returns an Elman neural network.

Examples

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

```
[X,T] = simpleseries_dataset;
net = elmannet(1:2,10)
[shift,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

Purpose

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` x `Q` matrix of input vectors
`T`	`1` x `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`

Purpose

Extend time series data to given number of timesteps

Syntax

```
extendts(x,ts,v)
extendts(x,ts)
```

Description

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

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

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

Examples

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

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

See Also

`nndata`, `catsamples`, `preparets`

feedforwardnet

Purpose Feedforward neural network

Syntax `feedforwardnet(hiddenSizes,trainFcn)`

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

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

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

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

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

and returns a feedforward neural network.

Examples Here a feedforward neural network is used 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)
```

See Also `fitnet`, `patternnet`, `cascadeforwardnet`

Purpose

Function fitting neural network

Syntax

`fitnet(hiddenSizes,trainFcn)`

Description

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

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

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

and returns a fitting neural network.

Examples

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

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

See Also

`feedforwardnet`, `nftool`

fixunknowns

Purpose

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)
dx_dy = fixunknowns('dx',x,y,ps)
dx_dy = fixunknowns('dx',x,[],ps)
name = fixunknowns('name');
fp = fixunknowns('pdefaults');
names = fixunknowns('pnames');
fixunknowns('pcheck',fp);
```

Description

`fixunknowns` processes matrixes 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.

`fixunknowns(X)` takes these inputs,

`X`	Single `N` x `Q` matrix or a `1` x `TS` row cell array of `N` x `Q` matrices

and returns

`Y`	Each `M` x `Q` matrix with `M - N` rows added (optional)
`PS`	Process settings that allow consistent processing of values

`fixunknowns(X,FP)` takes an empty struct `FP` of parameters.

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

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

`fixunknowns('dx',X,Y,PS)` returns the `M` x `N` x `Q` derivative of `Y` with respect to `X`.

`fixunknowns('dx',X,[],PS)` returns the derivative, less efficiently.

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

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

`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)
```

See Also

`mapminmax`, `mapstd`, `processpca`

Definition

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

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

The network will not be able to process the `NaN` values properly. Use the function `fixunknowns` to transform each row with `NaN` values (in this case only the first row) into two rows that encode that same information numerically.

```
[p2,ps] = fixunknowns(p1);
```

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

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

The first new row is the original first row, but with the mean value for that row (in this case `2`) replacing all `NaN` values. The elements of the second new row are now either `1`, indicating the original element was a known value, or `0` indicating that it was unknown. The original second row is now the new third row. In this way both known and unknown values are encoded numerically in a way that lets the network be trained and simulated.

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

```
p2new = fixunknowns('apply',p1new,ps);
```

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

Purpose

Form bias and weights into single vector

Syntax

```
wb = 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

Purpose

Forward propagation derivative function

Syntax

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

Description

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

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

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

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

Purpose

Convert data from standard neural network cell array form

Syntax

```
y = fromnndata(x,columnSample,cellTime)
```

Description

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

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

and returns the original data reformatted accordingly.

Examples

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

```
x = rands(5,6)
columnSamples = true; % samples are by columns.
cellTime = false; % time-steps represented by a matrix, not cell.
[y,wasMatrix] = tonndata(x,columnSamples,cellTime)
x2 = fromnndata(y,wasMatrix,columnSamples,cellTime)
```

Here data is defined in standard neural network data cell form. Converting this data does not change it. The data consists of three time series samples of 2-element signals over 3 timesteps.

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

See Also

`tonndata`

gadd

Purpose

Generalized addition

Syntax

```
gadd(a,b)
```

Description

This function generalizes matrix addition to the addition of cell arrays of matrices combined in an element-wise fashion.

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

Examples

Here matrix and cell array values are added.

```
gadd([1 2 3; 4 5 6],[10;20])
gadd({1 2; 3 4},{1 3; 5 2})
gadd({1 2 3 4},{10;20;30})
```

See Also

gsubtract, gmultiply, gdivide, gnegate, gsqrt

Purpose

Generalized division

Syntax

```
gdivide(a,b)
```

Description

This function generalizes matrix element-wise division to the division of cell arrays of matrices combined in an element-wise fashion.

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

Examples

Here matrix and cell array values are added.

```
gdivide([1 2 3; 4 5 6],[10;20])
gdivide({1 2; 3 4},{1 3; 5 2})
gdivide({1 2 3 4},{10;20;30})
```

See Also

`gadd`, `gsubtract`, `gmultiply`, `gnegate`, `gsqrt`

gensim

Purpose Generate Simulink block for neural network simulation

Syntax `gensim(net,st)`

To Get Help Type `help network/gensim`.

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

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

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

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

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

Examples

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

Purpose

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

Purpose

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

Purpose

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

Purpose

Get Simulink neural network block initial input and layer delays states

Syntax

```
[xi,ai] = getsimitinit(sysName,netName,net)
```

Description

`getsiminit(sysName,netName,net)` takes these arguments,

`sysName` The name of the Simulink® system containing the neural network block

`netName` The name of the Simulink neural network block

`net` The original neural network

and returns,

`xi` Initial input delay states

`ai` Initial layer delay states

Examples

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

```
[x,t] = simplenarx_dataset;
     net = narxnet(1:2,1:2,20);
     view(net)
     [xs,xi,ai,ts] = preparets(net,x,{},t);
     net = train(net,xs,ts,xi,ai);
     y = net(xs,xi,ai);
```

Now the network is converted to closed-loop, and the data is reformatted to simulate the network's closed-loop response.

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

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

```
[sysName,netName] = gensim(net,'InputMode','Workspace',...
   'OutputMode','WorkSpace','SolverMode','Discrete');
```

```
setsiminit(sysName,netName,net,xi,ai,1);
x1 = nndata2sim(x,1,1);
```

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

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

See Also

`gensim`, `setsiminit`, `nndata2sim`, `sim2nndata`

gettimesteps

Purpose

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`

Purpose

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,net.b,net.iw,net.lw)
```

See Also

`setwb`, `formwb`, `separatewb`

gmultiply

Purpose

Generalized multiplication

Syntax

```
gmultiply(a,b)
```

Description

This function generalizes matrix multiplication to the multiplication of cell arrays of matrices combined in an element-wise fashion.

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

Examples

Here matrix and cell array values are added.

```
gmulitiply([1 2 3; 4 5 6],[10;20])
gmultiply({1 2; 3 4},{1 3; 5 2})
gmultiply({1 2 3 4},{10;20;30})
```

See Also

`gadd`, `gsubtract`, `gdivide`, `gnegate`, `gsqrt`

Purpose

Generalized negation

Syntax

`gnegate(x)`

Description

This function generalizes matrix negation to the negation of cell arrays of matrices combined in an element-wise fashion.

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

Examples

Here is an example of negating a cell array:

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

See Also

`gadd`, `gsubtract`, `gdivide`, `gmultiply`, `gsqrt`

gridtop

Purpose

Grid layer topology function

Syntax

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

Description

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

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

`dimi`	Length of layer in dimension `i`

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

Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in an 8-by-5 grid.

```
pos = gridtop(8,5);
plotsompos(pos)
```

See Also

`hextop`, `randtop`, `tritop`

Purpose

Generalized square root

Syntax

```
gnegate(x)
```

Description

This function generalizes matrix element-wise square root to the square root of cell arrays of matrices combined in an element-wise fashion.

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

Examples

Here is an example of taking the element-wise square root of a cell array:

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

See Also

`gadd`, `gsubtract`, `gdivide`, `gmultiply`, `gnegate`

gsubtract

Purpose Generalized subtraction

Syntax `gsubtract(a,b)`

Description This function generalizes matrix subtraction to the subtraction of cell arrays of matrices combined in an element-wise fashion.

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

Examples Here matrix and cell array values are added.

```
gsubtract([1 2 3; 4 5 6],[10;20])
gsubtract({1 2; 3 4},{1 3; 5 2})
gsubtract({1 2 3 4},{10;20;30})
```

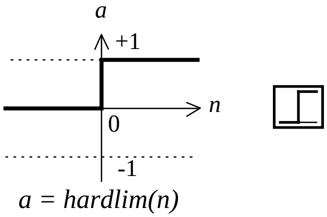
See Also `gadd`, `gmultiply`, `gdivide`, `gnegate`, `gsqrt`

hardlim

Purpose

Hard-limit transfer function

Graph and Symbol



Hard-Limit Transfer Function

Syntax

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

Description

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

`hardlim(N,FP)` takes `N` and optional function parameters,

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

and returns `A`, the `S` x `Q` Boolean matrix with 1s where `N` ≥ 0.

`hardlim('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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';
```

Algorithm

`hardlim(n)` = 1 if `n` ≥ 0

0 otherwise

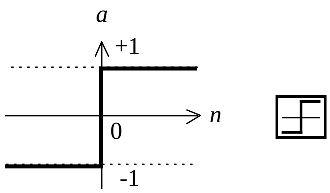
See Also

`sim`, `hardlims`

Purpose

Symmetric hard-limit transfer function

Graph and Symbol



a = hardlims(n)

Symmetric Hard-Limit Transfer Function

Syntax

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

Description

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

`hardlims(N,FP)` takes `N` and optional function parameters,

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

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

`hardlims('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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';
```

Algorithm

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

See Also

`sim`, `hardlim`

Purpose

Hexagonal layer topology function

Syntax

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

Description

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

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

dimi	Length of layer in dimension i

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

Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in an 8-by-5 hexagonal pattern.

```
pos = hextop(8,5);
plotsompos(pos)
```

See Also

gridtop, randtop, tritop

ind2vec

Purpose

Convert indices to vectors

Syntax

```
vec = ind2vec(ind)
```

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`.

Examples

Here four indices are defined and converted to vector representation.

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

See Also

`vec2ind`

Purpose

Initialize neural network

Syntax

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

To Get Help

Type `help network/init`.

Description

`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 with a two-element input (with ranges of 0 to 1 and -2 to 2) and one neuron. Once it is created you can display the neuron's weights and bias.

```
net = newp([0 1;-2 2],1);
net.iw{1,1}
net.b{1}
```

Training the perceptron alters its weight and bias values.

```
P = [0 1 0 1; 0 0 1 1];
T = [0 0 0 1];
net = train(net,P,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 (see `help newp`).

Algorithm

`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`

Purpose

Conscience bias initialization function

Syntax

```
b = 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` x 2 matrix of `R = [Pmin Pmax]` (default = `[1 1]`)

and returns an `S` x `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 `newc`.

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`. See `help help newc` for initialization examples.

Algorithm

`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

`initwb`, `initlay`, `init`, `learncon`

Purpose

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`.

`initlay(net)` takes

`net`	Neural network

and returns the network with each layer updated. `initlay(code)` returns useful information for each `code` string:

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

`initlay` does not have any initialization parameters.

Network Use

You can create a standard network that uses `initlay` by calling `newp`, `newlin`, `newff`, `newcf`, and many other `new*` 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`. See `help newp` and `help newlin` for initialization examples.

Algorithm

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

See Also

`initwb`, `initnw`, `init`

initlvq

Purpose

LVQ weight initialization function

Syntax

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

Purpose

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
```

initnw(net,i) takes two arguments,

net	Neural network
i	Index of a layer

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

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

Network Use

You can create a standard network that uses initnw by calling newff or newcf.

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`. See `help newff` and `help newcf` for training examples.

Algorithm

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

`initwb`, `initlay`, `init`

Purpose

Initialize SOM weights with principal components

Syntax

```
weights = initsom(inputs,dimensions,positions)
weights = initsom(inputs,dimensions,topologyFcn)
```

Description

initsompc initializes the weights of an N-dimensional self-organizing map so that the initial weights are distributed across the space spanned by the most significant N principal components of the inputs. Distributing the weight significantly speeds up SOM learning, as the map starts out with a reasonable ordering of the input space.

initsompc takes these arguments:

inputs	R x Q matrix of Q R-element input vectors
dimensions	D x 1 vector of positive integer SOM dimensions
positions	D x S matrix of S D-dimension neuron positions

and returns the following:

weights	S x R matrix of weights

Alternatively, initsompc can be called with topologyfcn (the name of a layer topology function) instead of positions. topologyfcn is called with dimensions to obtain positions.

Examples

```
inputs = rand(2,100)+[2;3]*ones(1,100);
dimensions = [3 4];
positions = gridtop(dimensions);
weights = initsompc(inputs,dimensions,positions);
```

See Also

gridtop, hextop, randtop

initwb

Purpose

By weight and bias layer initialization function

Syntax

```
net = initwb(net,i)
```

Description

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

`initwb(net,i)` takes two arguments,

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

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

Network Use

You can create a standard network that uses `initwb` by calling `newp` or `newlin`.

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`.

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

Algorithm

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

See Also

`initnw`, `initlay`, `init`

Purpose

Zero weight and bias initialization function

Syntax

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

Description

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

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

and returns an `S` x `R` weight matrix of zeros.

`initzero(S,[1 1])` returns an `S` x 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'`.
   Set each `net.layerWeights{i,j}.initFcn` to `'initzero'`.
   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

Purpose

Indicate if network inputs and outputs are configured

Syntax

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

Description

`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`

Purpose

Layer recurrent neural network

Syntax

```
layrecnet(layerDelays,hiddenSizes,trainFcn)
```

Description

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

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

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

and returns a layer recurrent neural network.

Examples

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

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

See Also

`preparets`, `removedelay`, `distdelaynet`, `timedelaynet`, `narnet`, `narxnet`

learncon

Purpose

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.

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

`B`	`S` x `1` bias vector
`P`	`1` x `Q` ones vector
`Z`	`S` x `Q` weighted input vectors
`N`	`S` x `Q` net input vectors
`A`	`S` x `Q` output vectors
`T`	`S` x `Q` layer target vectors
`E`	`S` x `Q` layer error vectors
`gW`	`S` x `R` gradient with respect to performance
`gA`	`S` x `Q` output gradient with respect to performance
`D`	`S` x `S` neuron distances
`LP`	Learning parameters, none, `LP = []`
`LS`	Learning state, initially should be = `[]`

and returns

`dB`	`S` x `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

`learncon(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`

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'`.
   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`).

Algorithm

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

Purpose

Gradient descent weight and bias learning function

Syntax

```
[dW,LS] = learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learngd(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngd(code)
```

Description

`learngd` is the gradient descent weight and bias learning function.

`learngd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `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

learngd(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 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.

Algorithm

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

Purpose

Gradient descent with momentum weight and bias learning function

Syntax

```
[dW,LS] = learngdm(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learngdm(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learngdm(code)
```

Description

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

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

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

and returns

`dW`	`S` x `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

`learngdm(code)` returns useful information for each `code` string:

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

Examples

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

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

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

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

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

Network Use

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

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

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

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

To allow the network to adapt,

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

See help newff or help newcf for examples.

Algorithm

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

Purpose

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.

`learnh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

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

`LP.lr - 0.01` Learning rate

`learnh(code)` returns useful information for each `code` string:

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

Examples

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

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

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

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

Network Use

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

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

Algorithm

`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'
```

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

See Also `learnhd`, `adapt`, `train`

Purpose

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.

`learnhd(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

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

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

`learnhd(code)` returns useful information for each `code` string:

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

Examples

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

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

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

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

Network Use

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

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

Algorithm

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

Purpose

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.

learnis(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs,

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

and returns

dW	S x R weight (or bias) change matrix
LS	New learning state

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

LP.lr - 0.01 Learning rate

learnis(code) returns useful information for each code string:

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

Examples

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

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

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

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

Network Use

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

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

Algorithm

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)
```

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

See Also `learnk`, `learnos`, `adapt`, `train`

Purpose

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.

`learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `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

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'.
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).

Algorithm

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

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

Reference

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

See Also

`learnis`, `learnos`, `adapt`, `train`

learnlv1

Purpose

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.

`learnlv1(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

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

`LP.lr - 0.01` Learning rate

`learnlv1(code)` returns useful information for each `code` string:

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

Examples

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

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

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

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

Network Use

You can create a standard network that uses `learnlv1` with `newlvq`. 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'`.
   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`).

Algorithm

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

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

See Also

learnlv2, adapt, train

Purpose

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.

`learnlv2(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `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)

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 newlvq.

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'.
   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).

Algorithm

learnlv2 implements Learning Vector Quantization 2.1, which works as follows:

For each presentation, if the winning neuron i should not have won, and the runnerup j should have, and the distance di between the winning neuron and the input p is roughly equal to the distance dj from the runnerup neuron to the input p according to the given window,

```
min(di/dj, dj/di) > (1-window)/(1+window)
```

then move the winning neuron i weights away from the input vector, and move the runnerup neuron j weights toward the input according to

```
dw(i,:) = - lp.lr*(p'-w(i,:))
dw(j,:) = + lp.lr*(p'-w(j,:))
```

See Also

learnlv1, adapt, train

learnos

Purpose

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.

`learnos(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

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

`LP.lr - 0.01` Learning rate

learnos(code) returns useful information for each code string:

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

Examples

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

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

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

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

Network Use

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

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

Algorithm

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'
```

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

See Also `learnis`, `learnk`, `adapt`, `train`

Purpose

Perceptron weight and bias learning function

Syntax

```
[dW,LS] = learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learnp(b,ones(1,Q),Z,N,A,T,E,gW,gA,D,LP,LS)
info = learnp(code)
```

Description

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

`learnp(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

`learnp(code)` returns useful information for each `code` string:

`'pnames'`	Names of learning parameters

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

Examples

Here you define a random input `P` and error `E` for a layer with a two-element input and three neurons.

```
p = rand(2,1);
e = rand(3,1);
```

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

```
dW = learnp([],p,[],[],[],[],e,[],[],[],[],[])
```

Network Use

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

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

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

Algorithm

`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'
```

Reference

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

See Also

`adapt`, `learnpn`, `train`

Purpose

Normalized perceptron weight and bias learning function

Syntax

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

Description

`learnpn` is a weight and bias learning function. It can result in faster learning than `learnp` when input vectors have widely varying magnitudes.

`learnpn(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

`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'. Set each net.layerWeights{i,j}.learnFcn to 'learnpn'. 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.

Algorithm

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'
```

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.

See Also

`adapt`, `learnp`, `train`

Purpose

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.

`learnsom(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `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

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'.
   Set each net.layerWeights{i,j}.learnFcn to 'learnsom'.

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).

Algorithm

learnsom calculates the weight change dW for a given neuron from the neuron's input P, activation A2, and learning rate LR:

```
dw = lr*a2*(p'-w)
```

where the activation A2 is found from the layer output A, neuron distances D, and the current neighborhood size ND:

```
a2(i,q) = 1,  if a(i,q) = 1
        = 0.5, if a(j,q) = 1 and D(i,j) <= nd
        = 0, otherwise
```

The learning rate LR and neighborhood size NS are altered through two phases: an ordering phase and a tuning phase.

The ordering phases lasts as many steps as LP.order_steps. During this phase LR is adjusted from LP.order_lr down to LP.tune_lr, and ND is adjusted from the maximum neuron distance down to 1. It is during this phase that neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

During the tuning phase LR decreases slowly from LP.tune_lr, and ND is always set to LP.tune_nd. During this phase the weights are expected to spread out relatively evenly over the input space while retaining their topological order, determined during the ordering phase.

See Also

adapt, train

learnsomb

Purpose

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.

learnsomb(W,P,Z,N,A,T,E,gW,gA,D,LP,LS) takes several inputs:

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

and returns the following:

dW	S x 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

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 x 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 newsom. 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`).

Algorithm

`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`, `train`

Purpose

Widrow-Hoff weight/bias learning function

Syntax

```
[dW,LS] = learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)
[db,LS] = learnwh(b,ones(1,Q),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.

`learnwh(W,P,Z,N,A,T,E,gW,gA,D,LP,LS)` takes several inputs,

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

and returns

`dW`	`S` x `R` weight (or bias) change matrix
`LS`	New learning state

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

`LP.lr`	`0.01`	Learning rate

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 only needs 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 newlin.

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'.
   Set each net.layerWeights{i,j}.learnFcn to 'learnwh'.
   Set net.biases{i}.learnFcn to 'learnwh'.
   Each weight and bias learning parameter property is automatically set to learnwh'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).

See help newlin for adaption and training examples.

Algorithm

`learnwh` calculates the weight change `dW` for a given neuron from the neuron's input `P` and error `E`, and the weight (or bias) learning rate `LR`, according to the Widrow-Hoff learning rule:

```
dw = lr*e*pn'
```

References

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

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

See Also

`adapt`, `train`

linearlayer

Purpose

Linear layer

Syntax

```
linearlayer(inputDelays,widrowHoffLR)
```

Description

Linear layers are single layers of linear neurons. They may be static, with input delays of 0, or dynamic, with input delays greater than 0. They can be trained on simple linear time series problems, but often are used adaptively to continue learning while deployed so they can adjust to changes in the relationship between inputs and outputs while being used.

If a network is needed to solve a nonlinear time series relationship, then better networks to try include `timedelaynet`, `narxnet`, and `narnet`.

`linearlayer(inputDelays,widrowHoffLR)` takes these arguments,

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

and returns a linear layer.

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

Examples

Here a linear layer is trained on a simple time series problem.

```
x = {0 -1 1 1 0 -1 1 0 0 1};
t = {0 -1 0 2 1 -1 0 1 0 1}
net = linearlayer(1:2,0.01)
[Xs,Xi,Ai,Ts] = preparets(net,X,T)
net = train(net,Xs,Ts,Xi,Ai);
view(net)
Y = net(Xs,Xi);
perf = perform(net,Ts,Y)
```

See Also

`preparets`, `removedelay`, `timedelaynet`, `narnet`, `narxnet`

Purpose

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.

`linkdist(pos)` takes one argument,

`pos`	`N` x `S` matrix of neuron positions

and returns the `S` x `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 `newsom`.

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`. See `help newsom` for training and adaption examples.

Algorithm

The link distance `D` between two position vectors `Pi` and `Pj` from a set of `S` vectors is

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

See Also

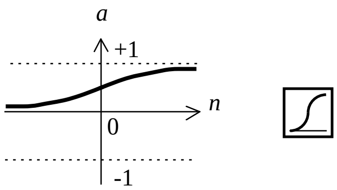
`sim`, `dist`, `mandist`

logsig

Purpose

Log-sigmoid transfer function

Graph and Symbol



a = logsig(n)

Log-Sigmoid Transfer Function

Syntax

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

Description

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

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

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

and returns `A`, the `S` x `Q` matrix of `N`'s elements squashed into `[0, 1]`.

`logsig('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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';
```

Algorithm

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

See Also

`sim`, `tansig`

lvqnet

Purpose

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 maps merges groups of first layer clusters into the classes defined by the target data.

The total number of first layer clusters is determined by the number of hidden neurons. The larger the hidden layer the more clusters the first layer can learn, and the more complex mapping of input to target classes can be made. The relative number of first layer clusters assigned to each target class are determined according to the distribution of target classes at the time of network initialization. This occurs when the network is automatically configured the first time `train` is called, or manually configured with the function `configure`, or manually initialized with the function `init` is called.

`lvqnet(hiddenSize,lvqLR,lvqLF)` takes these arguments,

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

and returns an LVQ neural network.

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

Examples

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

```
[x,t] = iris_dataset;
net = lvqnet(10)
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,y,t)
classes = vec2ind(y)
```

See Also

`preparets`, `removedelay`, `timedelaynet`, `narnet`, `narxnet`

Purpose

LVQ outputs processing function

Syntax

```
[X,PS] = lvqoutputs(X)
X = lvqoutputs('apply',X,PS)
X = lvqoutputs('reverse',X,PS)
dy_dx = lvqoutputs('dy_dx',X,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.

`lvqoutputs('apply',X,PS)` returns `X`.

`lvqoutputs('reverse',X,PS)` returns `X`.

`lvqoutputs('dy_dx',X,X,PS)` returns the identity derivative.

`lvqoutputs('dx_dy',X,X,PS)` returns the identity derivative.

See Also

`lvqnet`, `initlvq`

mae

Purpose

Mean absolute error performance function

Syntax

```
perf = mae(E,Y,X,FP)
dPerf_dy = mae('dy',E,Y,X,perf,FP)
dPerf_dx = mae('dx',E,Y,X,perf,FP)
info = mae(code)
```

Description

`mae` is a network performance function. It measures network performance as the mean of absolute errors.

`mae(E,Y,X,FP)` takes `E` and optional function parameters,

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

and returns the mean absolute error.

`mae('dy',E,Y,X,[perf,FP)` returns the derivative of `perf` with respect to `Y`.

`mae('dx',E,Y,X,perf,FP)` returns the derivative of `perf` with respect to `X`.

`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

Here a perceptron is created with a one-element input ranging from -10 to 10 and one neuron.

```
net = newp([-10 10],1);
```

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 = sim(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 `newp`.

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`'s being used to calculate performance.

See `help newp` for examples.

See Also

`mse`

mandist

Purpose

Manhattan distance weight function

Syntax

```
Z = mandist(W,P)
df = mandist('deriv')
D = mandist(pos);
```

Description

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

mandist(W,P) takes these inputs,

W	S x R weight matrix
P	R x Q matrix of Q input (column) vectors

and returns the S x Q matrix of vector distances.

mandist('deriv') returns '' because mandist does not have a derivative function.

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

mandist(pos) takes one argument,

pos	S row matrix of neuron positions

and returns the S x 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

You can create a standard network that uses mandist as a distance function by calling newsom.

To change a network so an input weight uses mandist, set net.inputWeight{i,j}.weightFcn to 'mandist'.
For a layer weight, set net.layerWeight{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.

Algorithm

The Manhattan distance D between two vectors X and Y is

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

See Also

sim, dist, linkdist

mapminmax

Purpose

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)
dy_dx = mapminmax('dy_dx',X,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].

mapminmax(X,YMIN,YMAX) takes X and optional parameters

X	N x Q matrix or a 1 x TS row cell array of N x Q matrices
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	Each M x Q matrix (where M == N) (optional)
PS	Process settings that allow consistent processing of values

mapminmax(X,FP) takes parameters as a struct: FP.ymin, FP.ymax.

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

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

mapminmax('dy_dx',X,Y,PS) returns the M x N x Q derivative of Y with respect to X.

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)
```

Algorithm

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

Definition

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.

Purpose

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

Syntax

```
[Y,PS] = mapstd(ymean,ystd)
[Y,PS] = mapstd(X,FP)
Y = mapstd('apply',X,PS)
X = mapstd('reverse',Y,PS)
dy_dx = mapstd('dy_dx',X,Y,PS)
dx_dy = mapstd('dx_dy',X,[],PS)
```

Description

mapstd processes matrices by transforming the mean and standard deviation of each row to ymean and ystd.

mapstd(X,ymean,ystd) takes X and optional parameters,

X	N x Q matrix or a 1 x TS row cell array of N x Q matrices
ymean	Mean value for each row of Y (default is 0)
ystd	Standard deviation for each row of Y (default is 1)

and returns

Y	Each M x Q matrix (where M == N) (optional)
PS	Process settings that allow consistent processing of values

mapstd(X,FP) takes parameters as a struct: FP.ymean, FP.ystd.

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

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

mapstd('dy_dx',X,Y,PS) returns the M x N x Q derivative of Y with respect to X.

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)
```

Algorithm

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

Definition

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

```
[pn,ps] = mapstd(p);
[tn,ts] = mapstd(t);
```

The original network inputs and targets are given in the matrices p and t. The normalized inputs and targets pn and tn that are returned will have zero means and unity standard deviation. The settings structures ps and ts contain the means and standard deviations of the original inputs and original targets. After the network has been trained, you should use these settings to transform any future inputs that are applied to the network. They effectively become a part of the network, just like the network weights and biases.

If mapstd is used to scale the targets, then the output of the network is trained to produce outputs with zero mean and unity standard deviation. To convert these outputs back into the same units that were used for the original targets, use ts. The following code simulates the network that was trained in the previous code, and then converts the network output back into the original units.

```
an = sim(net,pn);
a = mapstd('reverse',an,ts);
```

The network output an corresponds to the normalized targets tn. The unnormalized network output a is in the same units as the original targets t.

If mapstd is used to preprocess the training set data, then whenever the trained network is used with new inputs, you should preprocess them with the means and standard deviations that were computed for the training set using ps. The following commands apply a new set of inputs to the network already trained:

```
pnewn = mapstd('apply',pnew,ps);
anewn = sim(net,pnewn);
anew = mapstd('reverse',anewn,ts);
```

For most networks, including feedforwardnet, these steps are done automatically, so that you only need to use the sim command.

maxlinlr

Purpose

Maximum learning rate for linear layer

Syntax

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

Description

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

`maxlinlr(P)` takes one argument,

`P`	`R` x `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`.

`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`

meanabs

Purpose Mean of absolute elements of matrix or matrices

Syntax

```
[m,n] = meanabs(x)
```

Description `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

Purpose Mean of squared elements of matrix or matrices

Syntax `[m,n] = meansqr(x)`

Description `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`

Purpose

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.

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

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

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

Examples

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

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

Network Use

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

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

1. Set `net.initFcn` to `'initlay'`.
   (`net.initParam` automatically becomes `initlay`’s default parameters.)
2. Set `net.layers{i}.initFcn` to `'initwb'`.
3. Set each `net.inputWeights{i,j}.initFcn` to `'midpoint'`.
   Set each `net.layerWeights{i,j}.initFcn` to `'midpoint'`.

To initialize the network, call `init`.

See Also

`initwb`, `initlay`, `init`

minmax

Purpose

Ranges of matrix rows

Syntax

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

Description

`minmax(P)` takes one argument,

`P`	`R` x `Q` matrix

and returns the `R` x `2` matrix `PR` of minimum and maximum values for each row of `P`.

Alternatively, `P` can be an `M` x `N` cell array of matrices. Each matrix `P{i,j}` should have `Ri` rows and `Q` columns. In this case, `minmax` returns an `M` x `1` cell array where the `m`th matrix is an `Ri` x `2` matrix of the minimum and maximum values of elements for the matrix on the `i`th row of `P`.

Examples

```
P = [0 1 2; -1 -2 -0.5]
pr = minmax(P)
```

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

Purpose

Mean squared normalized error performance function

Syntax

```
perf = mse(E,Y,X,FP)
dPerf_dy = mse('dy',E,Y,X,perf,FP)
dPerf_dx = mse('dx',E,Y,X,perf,FP)
info = mse(code)
```

Description

mse is a network performance function. It measures the network's performance according to the mean of squared errors.

mse(E,Y,X,FP) takes E and optional function parameters,

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

and returns the mean squared error.

mse('dy',E,Y,X,perf,FP) returns the derivative of perf with respect to Y.

mse('dx',E,Y,X,perf,FP) returns the derivative of perf with respect to X.

mse('name') returns the name of this function.

mse('pnames') returns the names of the training parameters.

mse('pdefaults') returns the default function parameters.

Examples

Here a two-layer feed-forward network is created with a one-element input ranging from -10 to 10, four hidden tansig neurons, and one purelin output neuron.

```
net = newff([-10 10],[4 1],{'tansig','purelin'});
```

The network is given a batch of inputs P. The error is calculated by subtracting the output A from target T. Then the mean squared error is calculated.

```
p = [-10 -5 0 5 10];
t = [0 0 1 1 1];
y = sim(net,p)
```

```
e = t-y
perf = mse(e)
```

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

Network Use

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

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

In either case, calling train or adapt results in mse's being used to calculate performance.

See help newff or help newcf for examples.

See Also

mae

Purpose

Nonlinear autoregressive neural network

Syntax

`narnet(inputDelays,hiddenSizes,trainFcn)`

Description

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

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

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

and returns a 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)
```

See Also

`preparets`, `removedelay`, `timedelaynet`, `narnet`, `narxnet`

narxnet

Purpose

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,hiddenSizes,trainFcn)` takes these arguments,

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

and returns a NARX neural network.

Examples

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

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

Here the NARX network is simulated in closed loop form.

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

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

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

```
y = netp(Xs,Xi,Ai)
```

See Also

closeloop, narnet, openloop, preparets, removedelay, timedelaynet

nctool

Purpose Neural network classification or clustering tool

Syntax

```
nctool
```

Description `nctool` opens the neural network clustering GUI.

Algorithm `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.

Purpose

Negative distance weight function

Syntax

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

Description

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

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

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

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

`negdist(code)` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'fullderiv'`	Full derivative = 1, linear derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

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

`negdist('dp',W,P,Z,FP)` returns the derivative of `Z` with respect to `P`.

`negdist('size',S,R,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 newc or newsom.

To change a network so an input weight uses negdist, set net.inputWeight{i,j}.weightFcn to 'negdist'.
For a layer weight, set net.layerWeight{i,j}.weightFcn to 'negdist'.

In either case, call sim to simulate the network with negdist. See help newc or help newsom for simulation examples.

Algorithm

negdist returns the negative Euclidean distance:

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

See Also

sim, dotprod, dist

Purpose

Inverse transfer function

Syntax

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

Description

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

netinv(N,FP) takes inputs

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

and returns 1/N.

netinv('dn',N,A,FP) returns the 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.

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 x S x Q or S x 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

Purpose

Product net input function

Syntax

```
N = netprod({Z1,Z2,...,Zn},FP)
dN_dZj = netprod('dz'j,Z,N,FP)
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.

netprod(Z1,Z2,...,Zn) takes

Zi	S x Q matrices in a row cell array
FP	Row cell array of function parameters (optional, ignored)

and returns an elementwise product of Z1 to Zn.

netprod(code) returns information about this function. The following codes are defined:

'deriv'	Name of derivative function
'fullderiv'	Full N x S x Q derivative = 1, elementwise S x Q derivative = 0
'name'	Full name
'fpnames'	Returns names of function parameters
'fpdefaults'	Returns default function parameters

Examples

Here netprod combines two sets of weighted input vectors (user-defined).

```
z1 = [1 2 4;3 4 1];
z2 = [-1 2 2; -5 -6 1];
z = {z1,z2};
n = netprod({Z})
```

Here netprod combines the same weighted inputs with a bias vector. Because Z1 and Z2 each contain three concurrent vectors, three concurrent copies of B must be created with concur so that all sizes match.

```
b = [0; -1];
z = {z1, z2, concur(b,3)};
n = netprod(z)
```

Network Use

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

To change a network so that a layer uses `netprod`, set `net.layers{i}.netInputFcn` to `'netprod'`.

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

See Also

`sim`, `netsum`, `concur`

netsum

Purpose

Sum net input function

Syntax

```
N = netsum({Z1,Z2,...,Zn},FP)
dN_dZj = netsum('dz',j,Z,N,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.

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

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

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

`netsum('dz',j,{Z1,...,Zn},N,FP)` returns the derivative of `N` with respect to `Zj`. If `FP` is not supplied, the default values are used. If `N` is not supplied or is `[]`, it is calculated for you.

`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` x `Q` or `N` x `S` x `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 `newp` or `newlin` to create a standard network that uses `netsum`.

See Also

`netprod`, `netinv`

network

Purpose

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 `newp`, `newlin`, `newff`, etc.

`network` takes these optional arguments (shown with default values):

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

and returns

`net`	New network with the given property values

Properties

Architecture Properties

`net.numInputs`	0 or a positive integer	Number of inputs.
`net.numLayers`	0 or a positive integer	Number of layers.
`net.biasConnect`	`numLayer`-by-1 Boolean vector	If `net.biasConnect(i)` is 1, then layer `i` has a bias, and `net.biases{i}` is a structure describing that bias.
`net.inputConnect`	`numLayer`-by-`numInputs` Boolean vector	If `net.inputConnect(i,j)` is 1, then layer `i` has a weight coming from input `j`, and `net.inputWeights{i,j}` is a structure describing that weight.
`net.layerConnect`	`numLayer`-by-`numLayers` Boolean vector	If `net.layerConnect(i,j)` is 1, then layer `i` has a weight coming from layer `j`, and `net.layerWeights{i,j}` is a structure describing that weight.
`net.numInputs`	0 or a positive integer	Number of inputs.
`net.numLayers`	0 or a positive integer	Number of layers.
`net.biasConnect`	`numLayer`-by-1 Boolean vector	If `net.biasConnect(i)` is 1, then layer `i` has a bias, and `net.biases{i}` is a structure describing that bias.
`net.inputConnect`	`numLayer`-by-`numInputs` Boolean vector	If `net.inputConnect(i,j)` is 1, then layer `i` has a weight coming from input `j`, and `net.inputWeights{i,j}` is a structure describing that weight.
`net.layerConnect`	`numLayer`-by-`numLayers` Boolean vector	If `net.layerConnect(i,j)` is 1, then layer `i` has a weight coming from layer `j`, and `net.layerWeights{i,j}` is a structure describing that weight.
`net.outputConnect`	1-by-`numLayers` Boolean vector	If `net.outputConnect(i)` is 1, then the network has an output from layer `i`, and `net.outputs{i}` is a structure describing that output.

`net.numOutputs`	0 or a positive integer (read only)	Number of network outputs according to `net.outputConnect`.
`net.numInputDelays`	0 or a positive integer (read only)	Maximum input delay according to all `net.inputWeight{i,j}.delays`.
`net.numLayerDelays`	0 or a positive number (read only)	Maximum layer delay according to all `net.layerWeight{i,j}.delays`.

Subobject Structure Properties

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

Function Properties

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

Parameter Properties

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

Weight and Bias Value Properties

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

Other Properties

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

Examples

Here is the code 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
```

Here is the code to create the same network with one line of code.

```
net = network(1,2)
```

Here is the code to create a one-input, two-layer, feed-forward 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 see the properties of subobjects as follows:

```
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 get the weight matrices and bias vector as follows:

```
net.iw.{1,1}, net.iw{2,1}, net.b{1}
```

You can alter the properties of any of these subobjects. Here you change the transfer functions of both layers:

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

Here you change the number of elements in input 1 to 2 by setting each element's range:

```
net.inputs{1}.range = [0 1; -1 1];
```

Next you can simulate the network for a two-element input vector:

```
p = [0.5; -0.1];
y = sim(net,p)
```

See Also

`sim`

Purpose

Design generalized regression neural network

Syntax

```
net = newgrnn(P,T,spread)
```

Description

Generalized regression neural networks (grnns) are a kind of radial basis network that is often used for function approximation. grnns can be designed very quickly.

newgrnn(P,T,spread) takes three inputs,

P	R x Q matrix of Q input vectors
T	S x 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)
```

Reference

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

See Also

`sim`, `newrb`, `newrbe`, `newpnn`

Purpose

Design linear layer

Syntax

```
net = newlind(P,T,Pi)
```

Description

`newlind(P,T,Pi)` takes these input arguments,

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

where each element `Pi{i,k}` is an `Ri` x `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` x `TS` cell array	Each element `P{i,ts}` is an `Ri` x `Q` input matrix
`T`	`Nt` x `TS` cell array	Each element `P{i,ts}` is a `Vi` x `Q` matrix
`Pi`	`Ni` x `ID` cell array	Each element `Pi{i,k}` is an `Ri` x `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,:)
```

Algorithm

`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`

Purpose

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` x `Q` matrix of `Q` input vectors
`T`	`S` x `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)
```

Algorithm

`newpnn` creates a two-layer network. The first layer has `radbas` neurons, and calculates its weighted inputs with `dist` and its net input with `netprod`. The second layer has `compet` neurons, and calculates its weighted input with `dotprod` and its net inputs with `netsum`. Only the first layer has biases.

`newpnn` sets the first-layer weights to `P'`, and the first-layer biases are all set to `0.8326/spread`, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- `spread`. The second-layer weights `W2` are set to `T`.

Reference

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`

Purpose

Design radial basis network

Syntax

```
[net,tr] = 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.

`newrb(P,T,goal,spread,MN,DF)` takes two of these arguments,

`P`	`R` x `Q` matrix of `Q` input vectors
`T`	`S` x `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)
```

Algorithm

`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

Purpose

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.

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)
```

Algorithm

newrbe creates a two-layer network. The first layer has radbas neurons, and calculates its weighted inputs with dist and its net input with netprod. The second layer has purelin neurons, and calculates its weighted input with dotprod and its net inputs with netsum. Both layers have biases.

newrbe sets the first-layer weights to P', and the first-layer biases are all set to 0.8326/spread, resulting in radial basis functions that cross 0.5 at weighted inputs of +/- spread.

The second-layer weights IW{2,1} and biases b{2} are found by simulating the first-layer outputs A{1} and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones] = T
```

See Also

`sim`, `newrb`, `newgrnn`, `newpnn`

Purpose Neural network fitting tool

Syntax

```
nftool
```

Description `nftool` opens the neural network fitting tool GUI.

Algorithm `nftool` leads you through solving a data fitting problem, solving it with a two-layer feed-forward network trained with Levenberg-Marquardt.

See Also `nntool`

nncell2mat

Purpose

Combine neural network cell data into matrix

Syntax

```
[y,i,j] = nncell2mat(x)
```

Description

`[y,i,j] nncell2mat(x)` takes a cell array of matrices and returns,

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

The row and column sizes returned by `nncell2mat` can be used to convert the returned matrix back into a cell of matrices with `mat2cell`.

Examples

Here neural network data is converted to a matrix and back.

```
c = {rands(2,3) rands(2,3); rands(5,3) rands(5,3)};
[m,i,j] = nncell2mat(c)
c3 = mat2cell(m,i,j)
```

See Also

`nndata`, `nnsize`

Purpose

Crross correlation between neural network time series

Syntax

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

Description

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

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

and returns an `N`-by-`M` cell array where each `{i,j}` element is a `2*maxlag+1` length row vector formed from the correllations of `a` elements (i.e., matrix row) `i` and `b` elements (i.e., matrix column) `j`.

If `a` and `b` are specified with row vectors, the result is returned in matrix form.

The options for the normalization flag are:

- `'biased'` — scales the raw cross-correlation by 1/N.
- `'unbiased'` — scales the raw correlation by `1/(N-abs(k))`, where `k` is the index into the result.
- `'coeff'` — normalizes the sequence so that the correlations at zero lag are 1.0.
- `'none'` — no scaling. This is the default.

Examples

Here the autocorrelation of a random 1-element, 1-sample, 20-timestep signal is calculated with a maximum lag of 10.

```
a = nndata(1,1,20)
aa = nncorr(a,a,10)
```

Here the cross-correlation of the first signal with another random 2-element signal are found, with a maximum lag of 8.

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

See Also `confusion`, `regression`

Purpose

Create neural network data

Syntax

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

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

Returns an `M`-by-`TS` cell array where each row `i` has `N(i)`-by-`Q` sized matrices of value `v`. If `v` is not specified, random values are returned.

You can access subsets of neural network data with `getelements`, `getsamples`, `gettimesteps`, and `getsignals`.

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

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

Examples

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

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

To create random data with the same dimensions:

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

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

```
x = nndata(4,12)
```

See Also

`nnsize`, `tonndata`, `fromnndata`, `nndata2sim`, `sim2nndata`

nndata2sim

Purpose Convert neural network data to Simulink time series

Syntax `nndata2sim(x,i,q)`

Description `nndata2sim(x,i,q)`

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

and returns time series `q` of signal `i` as a Simulink time series structure.

Examples Here random neural network data is created with two signals having 4 and 3 elements respectively, over 10 timesteps. Three such series are created.

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

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

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

See Also `nndata`, `sim2nndata`, `nnsize`

Purpose

Number of neural data elements, samples, timesteps, and signals

Syntax

```
[N,Q,TS,M] = nnsize(x)
```

Description

nnsize(x) takes neural network data x and returns,

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

If X is a matrix, N is the number of rows of X, Q is the number of columns, and both TS and M are 1.

If X is a cell array, N is an Sx1 vector, where M is the number of rows in X, and N(i) is the number of rows in X{i,1}. Q is the number of columns in the matrices in X.

Examples

This code gets the dimensions of matrix data:

```
x = [1 2 3; 4 7 4]
[n,q,ts,s] = nnsize(x)
```

This code gets the dimensions of cell array data:

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

See Also

nndata, numelements, numsamples, numsignals, numtimesteps

nnstart

Purpose 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 wizards. It also provides links to lists of datasets, demos, and other useful information for getting started.

See Also `nctool`, `nftool`, `nprtool`, `ntstool`

Purpose 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.

nntraintool

Purpose Neural network training tool

Syntax

```
nntraintool
```

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.

Purpose

Remove neural network open- and closed-loop feedback

Syntax

```
net = noloop(net)
```

Description

`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);
nnet = 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`

Purpose

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`

Purpose

Normalized dot product weight function

Syntax

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

Description

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

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

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

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

`normprod(code)` returns information about this function. The following codes are defined:

`'deriv'`	Name of derivative function
`'pfullderiv'`	Full input derivative = 1, linear input derivative = 0
`'wfullderiv'`	Full weight derivative = 1, linear weight derivative = 0
`'name'`	Full name
`'fpnames'`	Returns names of function parameters
`'fpdefaults'`	Returns default function parameters

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

`normprod('dp',W,P,Z,FP)` returns the derivative of `Z` with respect to `P`.

`normprod('size',S,R,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.inputWeight{i,j}.weightFcn to 'normprod'.
For a layer weight, set net.layerWeight{i,j}.weightFcn to 'normprod'.

In either case, call sim to simulate the network with normprod. See newgrnn for simulation examples.

Algorithm

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

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

See Also

dotprod

Purpose

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

Purpose Neural network pattern recognition tool

Syntax

```
nprtool
```

Description `nprtool` opens the neural network pattern-recognition GUI.

Algorithm `nprtool` leads you through solving a pattern-recognition classification problem using a two-layer feed-forward `patternnet` network with sigmoid output neurons.

See Also `nftool`, `nctool`, `ntstool`

Purpose

Neural network time series tool

Syntax

```
nprtool
```

Description

`nprtool` opens the neural network pattern-recognition GUI.

Algorithm

`ntstool` leads you through solving time series problems using `narxnet`, `narnet` and `timedelaynet` neural networks.

See Also

`nftool`, `nctool`, `nprtool`

Purpose

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.

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

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

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);
gwb = num2deriv('dperf_dwb',net,x,t)
jwb = num2deriv('de_dwb',net,x,t)
```

See Also

bttderiv, defaultderiv, fpderiv, num5deriv, staticderiv

num5deriv

Purpose

Numeric five-point stencil neural network derivative function

Syntax

```
num5deriv('dperf_dwb',net,X,T,Xi,Ai,EW)
num5deriv('de_dwb',net,X,T,Xi,Ai,EW)
```

Description

This function calculates derivatives using the five-point numeric derivative rule.

$$y_1 = y(x + 2dx)$$

$$y_2 = y(x + dx)$$

$$y_3 = y(x - dx)$$

$$y_4 = y(x - 2dx)$$

$$\frac{dy}{dx} = \frac{y_1 - 8y_2 + 8y_3 - y_4}{dx}$$

This function is much slower than the analytical (non-numerical) derivative functions, but is provided as a means of checking the analytical derivative functions. The other numerical function, `num2deriv`, is faster but less accurate.

`num5deriv('dperf_dwb',net,X,T,Xi,Ai,EW)` takes these arguments,

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

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);
gwb = num5deriv('dperf_dwb',net,x,t)
jwb = num5deriv('de_dwb',net,x,t)
```

See Also

bttderiv, defaultderiv, fpderiv, num2deriv, staticderiv

Purpose

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 `Sx1` vector, where `S` is the number of signals (i.e., rows of `X`), and each element `S(i)` is the number of elements in each signal `i` (i.e., rows of `x{i,1}`.

Examples

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

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

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

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

See Also

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

numfinite

Purpose 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

Purpose

Number of NaN values in neural network data

Syntax

```
numnan(x)
```

Description

`numnan(x)` takes a matrix or cell array of matrices and returns the number of NaN elements in it.

Examples

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

x = {[1 2; 3 NaN] [5 NaN; NaN 8]}
n = numnan(x)
```

See Also

`numnan`, `nndata`, `nnsize`

Purpose

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

Purpose

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`

Purpose

Number of timesteps 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 timesteps represented by matrix data:

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

This code calculates the number of timesteps 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

Purpose

Convert neural network closed loop feedback to open loop

Syntax

```
net = openloop(net)
```

Description

`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).

Examples

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);
Ycloseed = net(Xs,Xi,Ai);
net = openloop(net)
view(net)
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
Yopen = net(Xs,Xi,Ai)
```

See Also

`closeloop`, `noloop`

Purpose

Pattern recognition network

Syntax

```
patternnet(hiddenSizes,trainFcn)
```

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)` takes these arguments,

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

and returns a pattern recognition neural network.

Examples

Here a pattern recognition network is designed to classify iris flowers.

```
[x,t] = iris_dataset;
net = patternnet(10)
net = train(net,x,t);
view(net)
y = net(x);
perf = perform(net,t,y)
classes = vec2ind(y)
```

See Also

`lvqnet`, `competlayer`, `selforgmap`, `nprtool`

perceptron

Purpose

Perceptron

Syntax

`perceptron(hardlimitTF,perceptronLF)`

Description

Perceptrons are simple single-layer binary classifiers, which divide the input space with a linear decision boundary.

Perceptrons are provide for historical interest. For much better results use `patternnet`, which can solve non-linearly separable problems. Sometimes when people refer to perceptrons they are referring to feed-forward pattern recognition networks, such as `patternnet`. But the original perceptron, described here, can solve only very simple problems.

Perceptrons can learn to solve a narrow class of classification problems. Their significance is they have a simple learning rule and were one of the first neural networks to reliably solve a given class of problems.

`perceptron(hardlimitTF,perceptronLF)` takes these arguments,

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

and returns a perceptron.

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

Examples

Here a perceptron is used to solve a very simple classification logical-OR problem.

```
x = [0 0 1 1; 0 1 0 1];
t = [0 1 1 1];
net = perceptron;
net = train(net,x,t);
view(net)
y = net(x);
```

See Also

`preparets`, `removedelay`, `timedelaynet`, `narnet`, `narxnet`

Purpose

Calculate network performance

Syntax

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

Description

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

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

and returns network performance calculated according to the `net.performFcn` and `net.performParam` property values.

The target and output data must have the same dimensions. The error weights may be the same dimensions as the targets, in the most general case, but may also have any of its dimension be 1. This gives the flexibilty of defining error weights across any dimension desired.

Error weights can be defined by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2]; % Across 4 samples
ew = [0.1; 0.5; 1.0]; % Across 3 elements
ew = {0.1 0.2 0.3 0.5 1.0}; % Across 5 timesteps
ew = {1.0; 0.5}; % Across 2 outputs
```

The may also be defined across any combination, such as across two time-series (i.e. two samples) over four timesteps.

```
ew = {[0.5 0.4],[0.3 0.5],[1.0 1.0],[0.7 0.5]};
```

In the general case, error weights may have exactly the same dimensions as targets, in which case each target value will have an associated error weight.

The default error weight treats all errors the same.

```
ew = {1}
```

Examples

Here a simple fitting problem is solved with a feed-forward network and its performance calculated.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(20);
net = train(net,x,t);
y = net(x);
perf = perform(net,t,y)
```

See Also

train, configure, init

plotconfusion

Purpose

Plot classification confusion matrix

Syntax

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

Description

`plotconfusion(targets,outputs)` displays the classification confusion grid.

`plotconfusion(targets1,outputs1,'name1',...)` plots a series of plots.

Examples

```
load simpleclass_dataset
net = newpr(simpleclassInputs,simpleclassTargets,20);
net = train(net,simpleclassInputs,simpleclassTargets);
simpleclassOutputs = sim(net,simpleclassInputs);
plotconfusion(simpleclassTargets,simpleclassOutputs);
```

plotep

Purpose

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 already created by `plotes`.

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

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

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

`plotep(W,B,E,H)` continues plotting using the vector `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`

Purpose

Plot autocorrelation of error time series

Syntax

```
ploterrcorr(e)
ploterrcorr(...,'outputIndex',outputIndex)
```

Description

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

ploterrcorr may also take an optional property name/value pair 'outputIndex', which defines which output whose error autocorrelation is being plotted. The default is 1.

Examples

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

```
[X,T] = simplenarx_dataset;
net = narxnet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts,Xi,Ai);
Y = net(Xs,Xi,Ai);
E = gsubtract(Ts,Y);
ploterrcorr(E)
```

See Also

plotinerrcorr, plotresponse

ploterrhist

Purpose

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`

Purpose

Plot error surface of single-input neuron

Syntax

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

Description

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

`WV`	`1` x `N` row vector of values of `W`
`BV`	`1` x `M` row vector of values of `B`
`ES`	`M` x `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

Purpose

Plot function fit

Syntax

```
plotfit(net,inputs,targets)
plotfit(net,inputs1,targets1,'name1',inputs2,targets2,'name2',...)
```

Description

`plotfit(NET,INPUTS,TARGETS)` plots the output function of a network across the range of the inputs `X` and also plots target `T` and output data points associated with values in `X`. Error bars show the difference between outputs and `T`.

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,outputs1,'name1',...)` plots a series of plots.

Examples

```
load simplefit_dataset
net = newfit(simplefitInputs,simplefitTargets,20);
[net,tr] = train(net,simplefitInputs,simplefitTargets);
plotfit(net,simplefitInputs,simplefitTargets);
```

See Also

`plottrainstate`

Purpose

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 autocorrelation 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);
ploterrcorr(E)
plotinerrcorr(Xs,E)
```

See Also

`ploterrcorr`, `plotresponse`, `ploterrhist`

Purpose

Plot classification line on perceptron vector plot

Syntax

```
plotpc(W,B)
plotpc(W,B,H)
```

Description

`plotpc(W,B)` takes these inputs,

`W`	`S` x `R` weight matrix (`R` must be 3 or less)
`B`	`S` x `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 with inputs ranging over the values in `P`, assigns values to its weights and biases, and plots the resulting classification line.

```
net = newp(minmax(p),1);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```

See Also

`plotpv`

Purpose

Plot network performance

Syntax

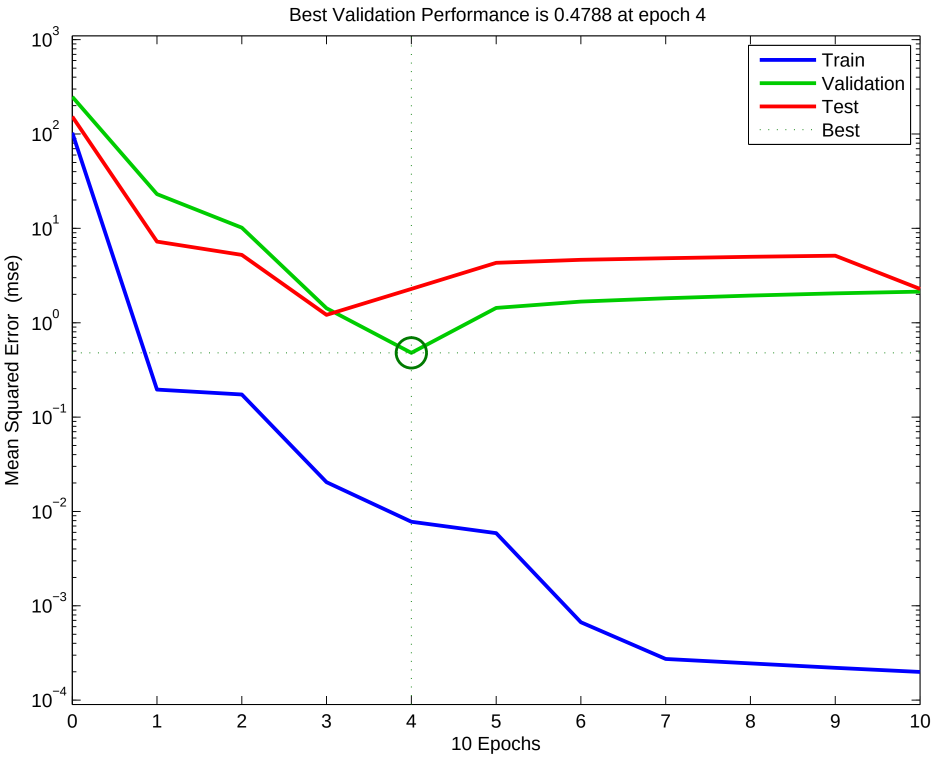
```
plotperform(tr)
```

Description

`plotperform(TR)` plots the training, validation, and test performances given the training record `TR` returned by the function `train`.

Examples

```
load simplefit_dataset
net = newff(simplefitInputs,simplefitTargets,20);
[net,tr] = train(net,simplefitInputs,simplefitTargets);
plotperform(tr);
```



See Also

`plottrainstate`

plotpv

Purpose

Plot perceptron input/target vectors

Syntax

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

Description

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

P	R x Q matrix of input vectors (R must be 3 or less)
T	S x 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

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 with inputs ranging over the values in P, assigns values to its weights and biases, and plots the resulting classification line.

```
net = newp(minmax(p),1);
net.iw{1,1} = [-1.2 -0.5];
net.b{1} = 1;
plotpc(net.iw{1,1},net.b{1})
```

See Also

`plotpc`

Purpose

Plot linear regression

Syntax

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

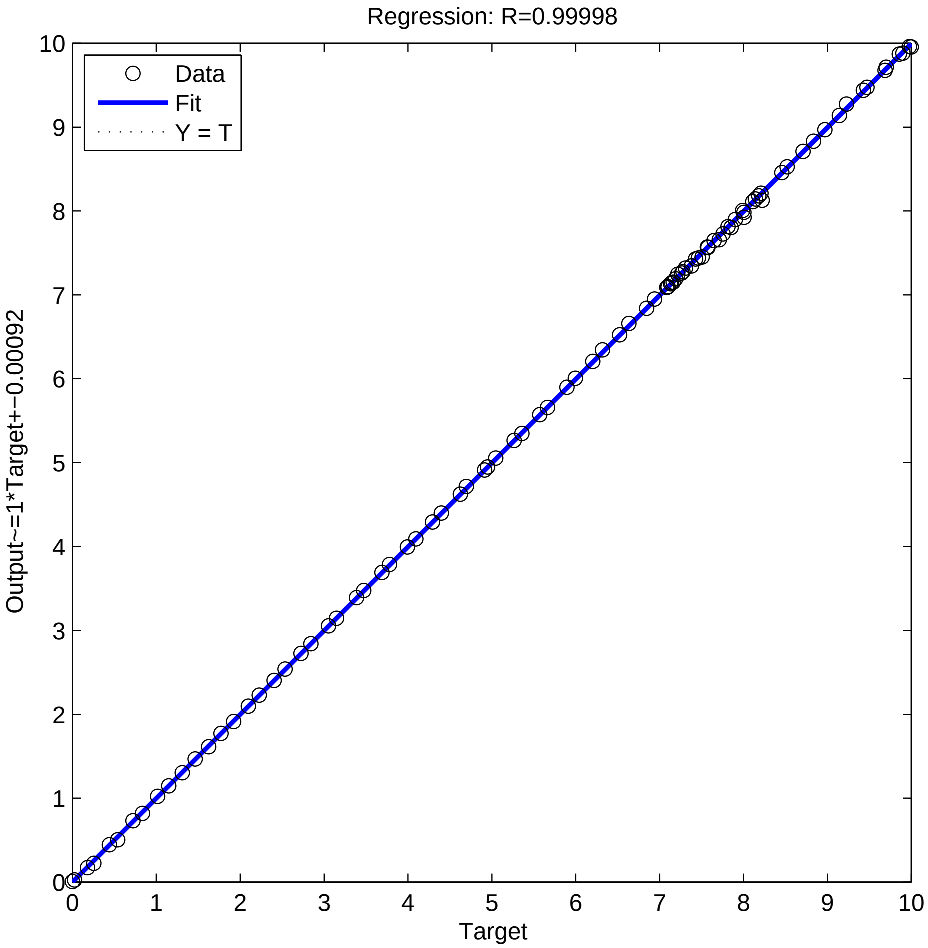
Description

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

`plotregression(targets1,outputs2,'name1',...)` generates multiple plots.

Examples

```
load simplefit_dataset
net = newff(simplefitInputs,simplefitTargets,20);
[net,tr] = train(net,simplefitInputs,simplefitTargets);
simplefitOutputs = sim(net,simplefitInputs);
plotregression(simplefitTargets,simplefitOutputs);
```



See Also `plottrainstate`

Purpose

Plot dynamic network time series response

Syntax

```
plotresponse(t,y)
plotresponse(t1,'name1',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

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);
plotresponse(Ts,Y)
```

See Also

`ploterrcorr`, `plotinerrcorr`, `ploterrhist`

plotroc

Purpose

Plot receiver operating characteristic

Syntax

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

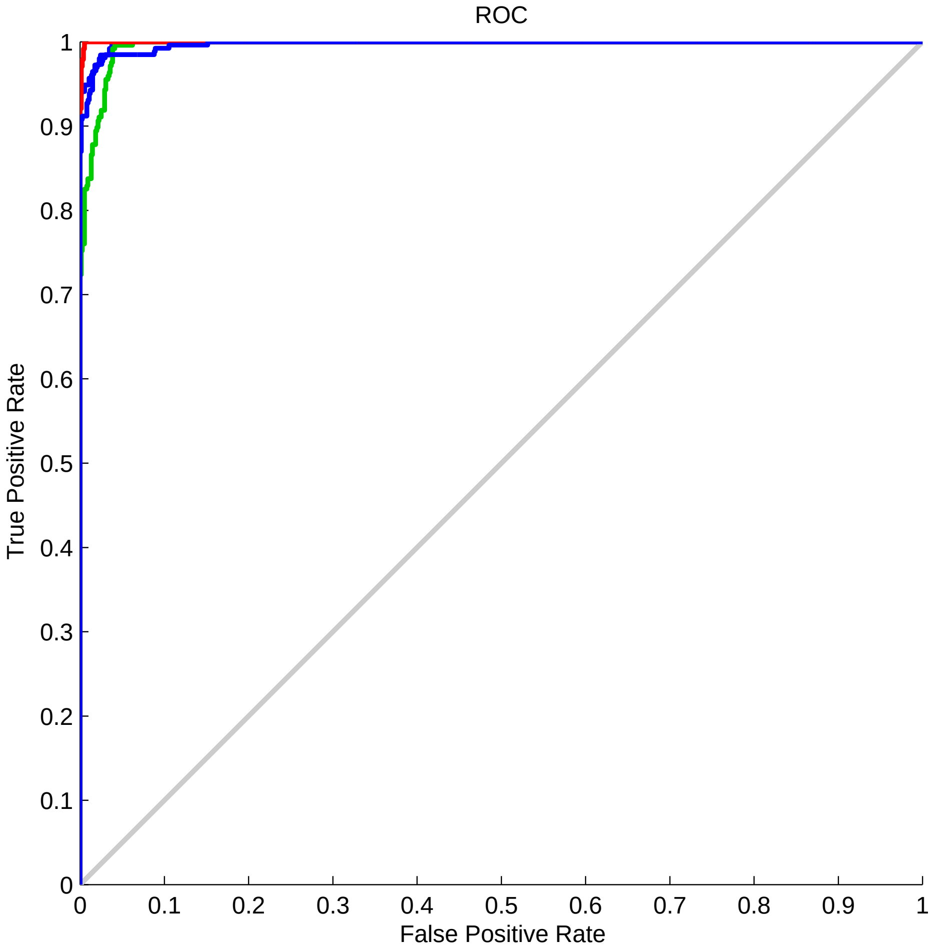
Description

`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

```
load simplecluster_dataset
net = newpr(simpleclusterInputs,simpleclusterTargets,20);
net = train(net,simpleclusterInputs,simpleclusterTargets);
simpleclusterOutputs = sim(net,simpleclusterInputs);
plotroc(simpleclusterTargets,simpleclusterOutputs);
```


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

See Also `roc`

plotsom

Purpose

Plot self-organizing map

Syntax

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

Description

`plotsom(pos)` takes one argument,

`POS`	`N` x `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` x `R` weight matrix
`D`	`S` x `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

Here are some neat 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 `newsom` for an example of plotting a layer's weight vectors with the input vectors they map.

See Also

`initsompc`, `learnsom`

Purpose

Plot self-organizing map sample hits

Syntax

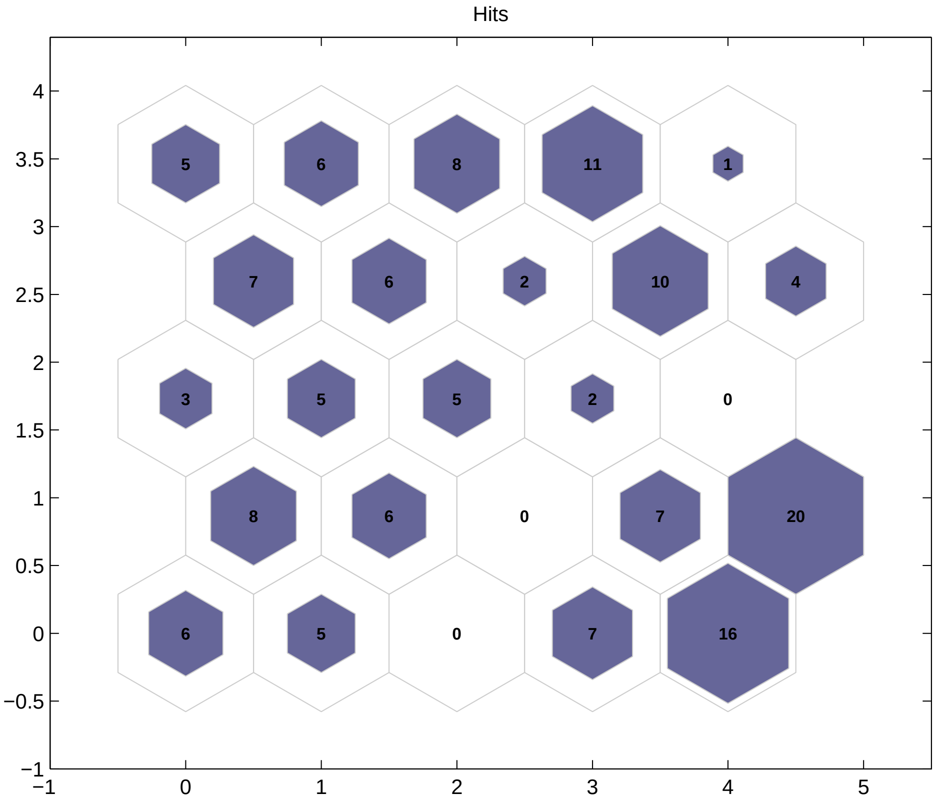
```
plotsomhits(net,inputs)
plotsomhits(net,inputs,targets)
```

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.

Examples

```
load iris_dataset
net = newsom(irisInputs,[5 5]);
[net,tr] = train(net,irisInputs);
plotsomhits(net,irisInputs);
```



See Also

`plotsomplanes`

Purpose

Plot self-organizing map neighbor connections

Syntax

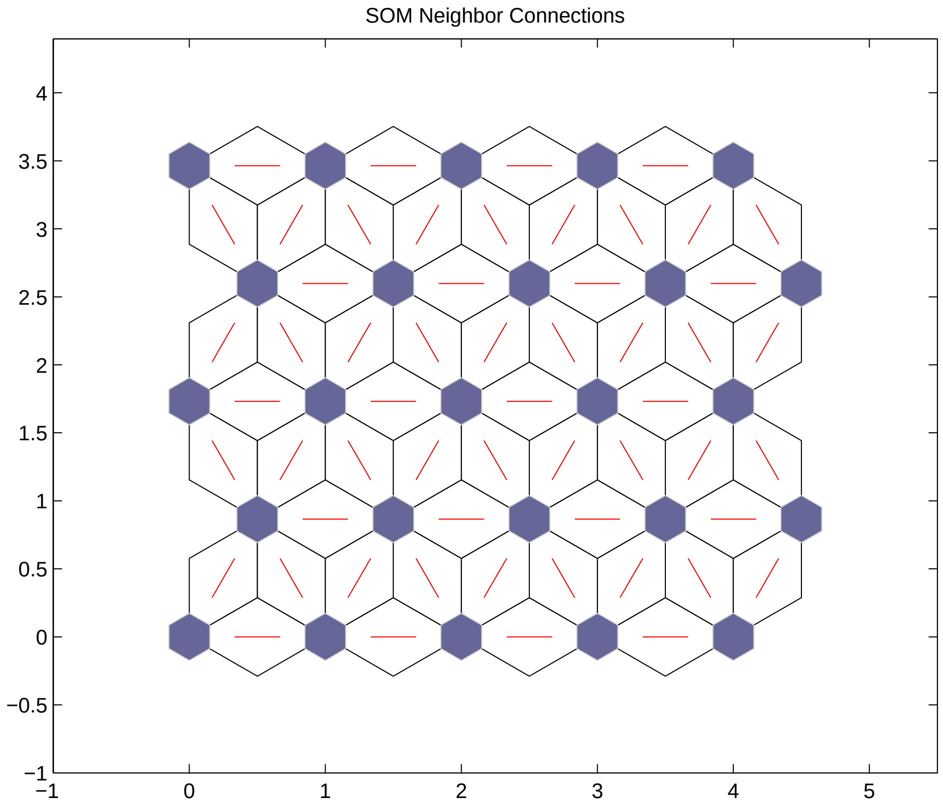
```
plotsomnc(net)
```

Description

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

Examples

```
load iris_dataset
net = newsom(irisInputs,[5 5]);
plotsomnc(net);
```



See Also

`plotsomnd`, `plotsomplanes`, `plotsomhits`

plotsomnd

Purpose

Plot self-organizing map neighbor distances

Syntax

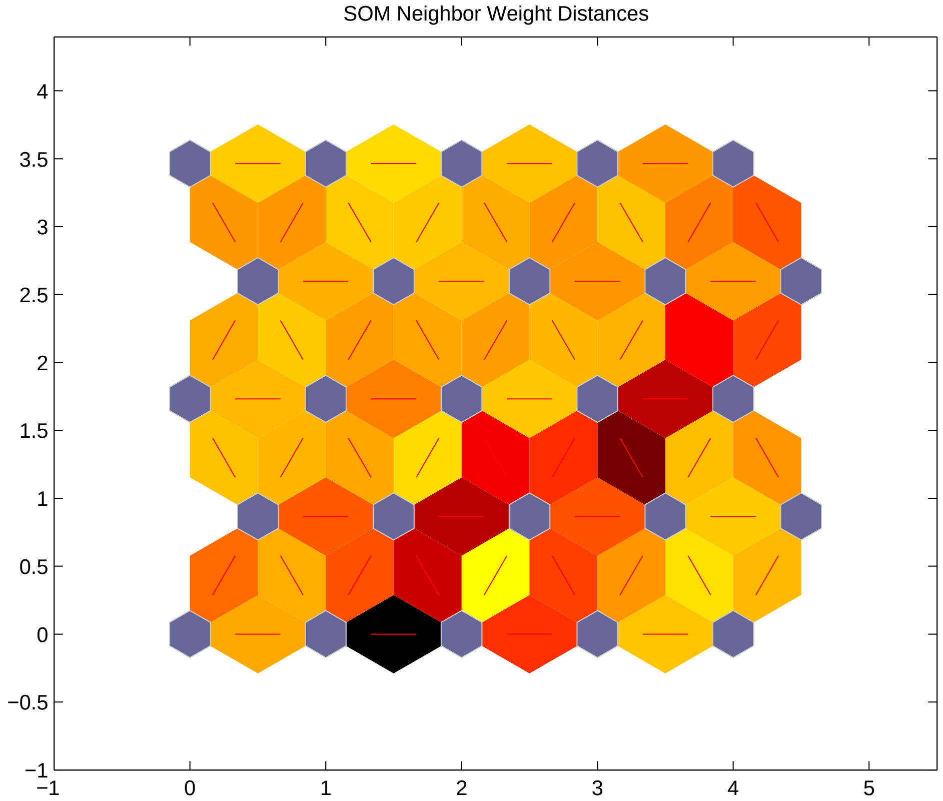
```
plotsomnd(net)
```

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.

Examples

```
load iris_dataset
net = newsom(irisInputs,[5 5]);
[net,tr] = train(net,irisInputs);
plotsomnd(net);
```



See Also

`plotsomhits`, `plotsomnc`, `plotsomplanes`

plotsomplanes

Purpose Plot self-organizing map weight planes

Syntax

```
plotsomplanes(net)
```

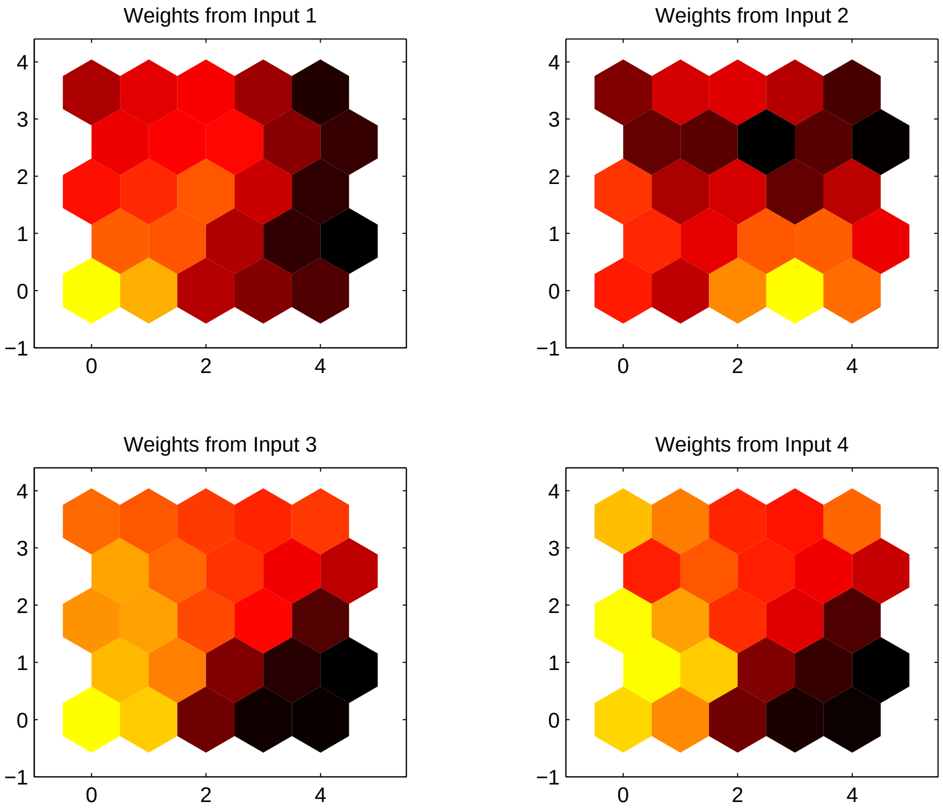
Description `plotsomplanes(net)` generates a set of subplots. Each `i`th subplot shows the weights from the `i`th input to the layer's neurons, with the most negative connections shown as blue, zero connections as black, and the strongest positive connections as red.

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

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

Examples

```
load iris_dataset
net = newsom(irisInputs,[5 5]);
[net,tr] = train(net,irisInputs);
plotsomplanes(net);
```



Weights from Input 1

Weights from Input 2

Weights from Input 3

Weights from Input 4

See Also

`plotsomhits`, `plotsomnc`, `plotsomnd`

plotsompos

Purpose Plot self-organizing map weight positions

Syntax

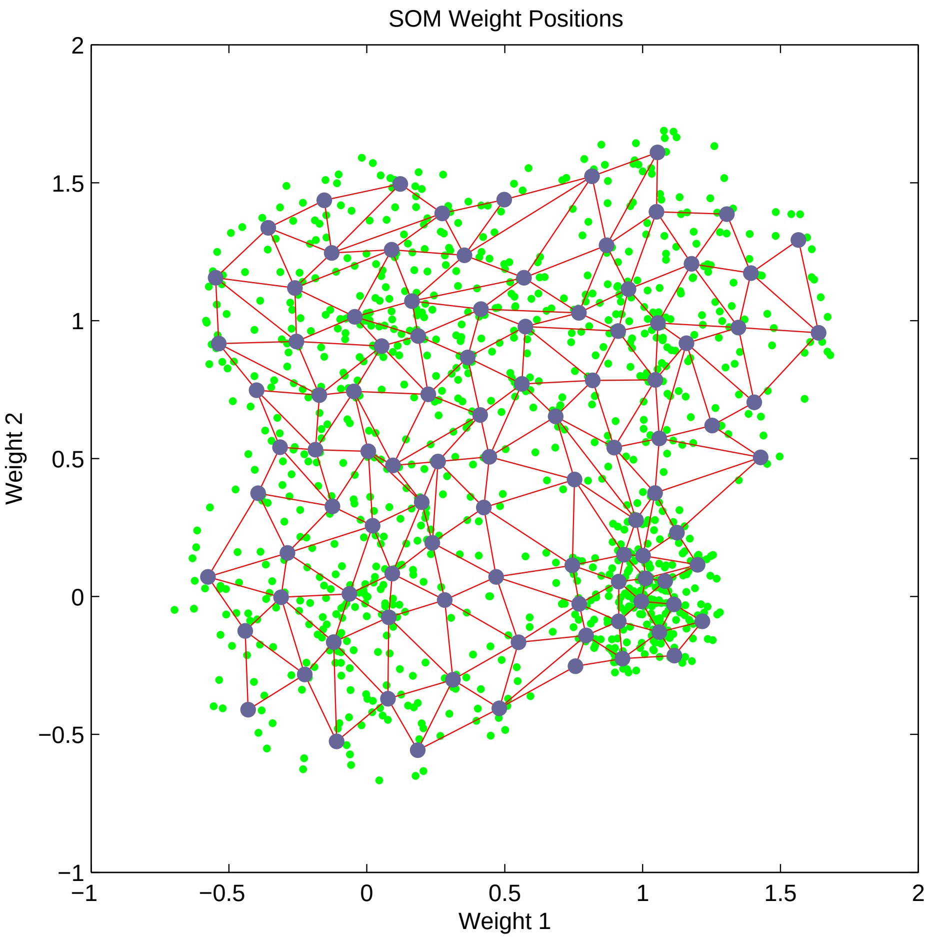
```
plotsompos(net)
plotsompos(net,inputs)
```

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.

Examples

```
load simplecluster_dataset
net = newsom(simpleclusterInputs,[10 10]);
net = train(net,simpleclusterInputs);
plotsompos(net,simpleclusterInputs);
```



See Also `plotsomnd`, `plotsomplanes`, `plotsomhits`

plotsomtop

Purpose

Plot self-organizing map topology

Syntax

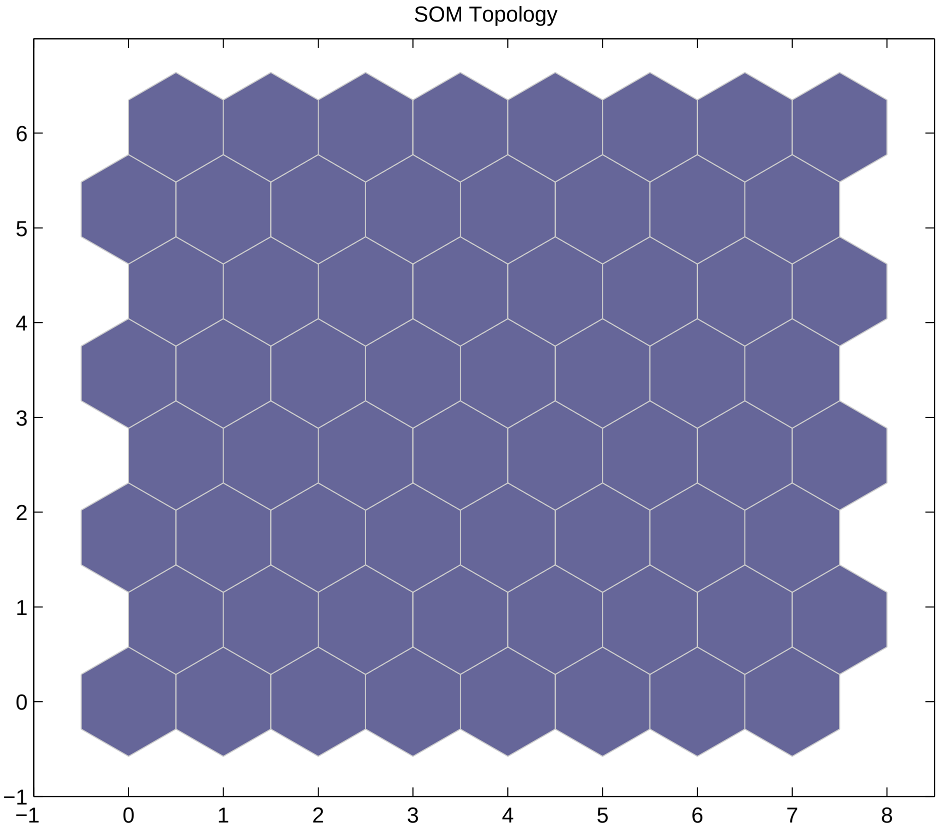
`plotsomtop(net)`

Description

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

Examples

```
load iris_dataset
net = newsom(irisInputs,[8 8]);
plotsomtop(net);
```



See Also

`plotsomnd`, `plotsomplanes`, `plotsomhits`

Purpose

Plot training state values

Syntax

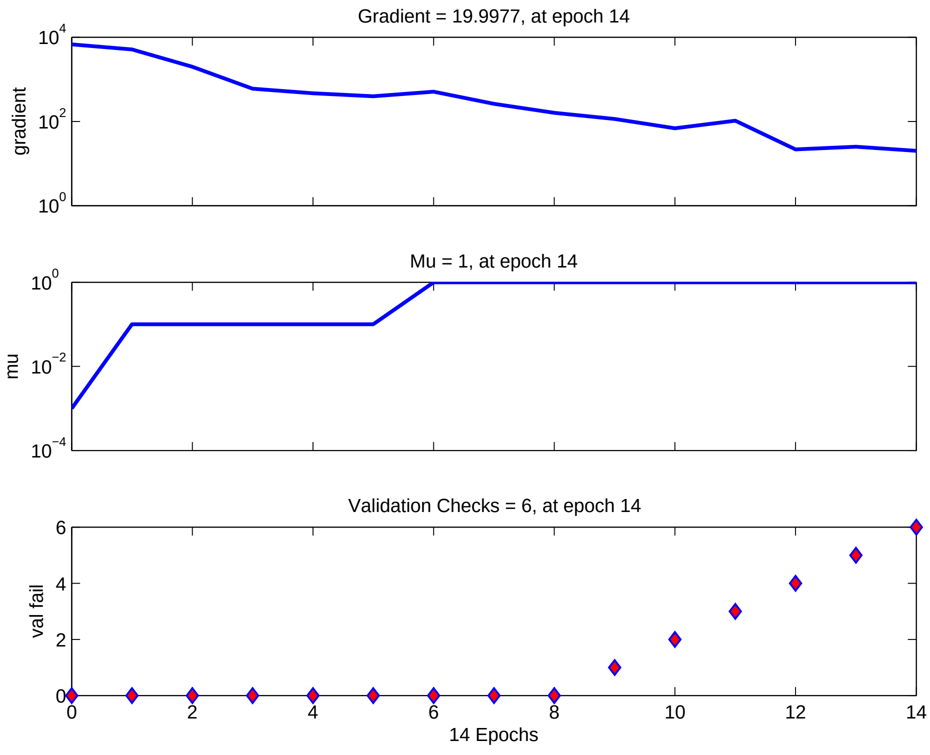
```
plottrainstate(tr)
```

Description

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

Examples

```
load housing
net = newff(p,t,20);
[net,tr] = train(net,p,t);
plottrainstate(tr);
```



See Also

`plotfit`, `plotperform`, `plotregression`

plotv

Purpose

Plot vectors as lines from origin

Syntax

```
plotv(M,T)
```

Description

`plotv(M,T)` takes two inputs,

`M`	`R` x `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

```
plotv([-.4 0.7 .2; -0.5 .1 0.5],'-')
```

plotvec

Purpose

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

```
x = [0 1 0.5 0.7; -1 2 0.5 0.1];
c = [1 2 3 4];
plotvec(x,c)
```

plotwb

Purpose

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(...,'toLayers',toLayers) 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

Purpose

Pseudonormalize columns of matrix

Syntax

`pnormc(X,R)`

Description

`pnormc(X,R)` takes these arguments,

`X`	`M` x `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

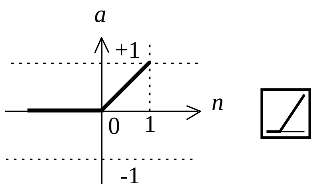
`normc`, `normr`

poslin

Purpose

Positive linear transfer function

Graph and Symbol



a = poslin(n)

Positive Linear Transfer Function

Syntax

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

Description

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

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

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

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

`poslin('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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.

Algorithm

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

Purpose

Prepare input and target time series data for network simulation or training

Syntax

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

Description

This function simplifies the normally complex and error prone task of reformatting input and target timeseries. It automatically shifts input and target time series as many steps as are needed to fill the initial input and layer delay states. If the network has open loop feedback, then it copies feedback targets into the inputs as needed to define the open loop inputs.

Each time a new network is designed, with different numbers of delays or feedback settings, `preparets` can be called to 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.

`preparets(net,Xnf,Tnf,Tf,EW)` takes these arguments,

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

and returns,

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

Examples

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

```
net = timedelaynet(20);
view(net)
[X,T] = simpleseries_dataset;
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts);
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);
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] = preparets(net,X,{},T);
y = net(Xs,Xi,Ai);
```

See Also

adddelay, closeloop, narnet, narxnet, openloop, removedelay, timedelaynet

Purpose

Process columns of matrix with principal component analysis

Syntax

```
[y,ps] = processpca(maxfrac)
[y,ps] = processpca(x,fp)
y = processpca('apply',x,ps)
x = processpca('reverse',y,ps)
dx_dy = processpca('dx',x,y,ps)
dx_dy = processpca('dx',x,[],ps)
name = processpca('name');
fp = processpca('pdefaults');
names = processpca('pnames');
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.

`processpca(X,maxfrac)` takes `X` and an optional parameter,

`X`	`N` x `Q` matrix or a `1` x `TS` row cell array of `N` x `Q` matrices
`maxfrac`	Maximum fraction of variance for removed rows (default is 0)

and returns

`Y`	Each `N` x `Q` matrix with `N - M` rows deleted (optional)
`PS`	Process settings that allow consistent processing of values

`processpca(X,FP)` takes parameters as a struct: `FP.maxfrac`.

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

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

`processpca('dx',X,Y,PS)` returns the `M` x `N` x `Q` derivative of `Y` with respect to `X`.

`processpca('dx',X,[],PS)` returns the derivative, less efficiently.

`processpca('name')` returns the name of this process method.

processpca('pdefaults') returns default process parameter structure.

processpca('pdesc') returns the process parameter descriptions.

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

Examples

Here is how to format a matrix with an independent row, a correlated row, and a completely redundant row so that its rows are uncorrelated and the redundant row is dropped.

```
x1_independent = rand(1,5)
x1_correlated = rand(1,5) + x_independent;
x1_redundant = x_independent + x_correlated
x1 = [x1_independent; x1_correlated; x1_redundant]
[y1,ps] = processpca(x1)
```

Next, apply the same processing settings to new values.

```
x2_independent = rand(1,5)
x2_correlated = rand(1,5) + x_independent;
x2_redundant = x_independent + x_correlated
x2 = [x2_independent; x2_correlated; x2_redundant];
y2 = processpca('apply',x2,ps)
```

Reverse the processing of y1 to get x1 again.

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

Algorithm

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

```
y = 2*(x-minx)/(maxx-minx) - 1;
```

Values in rows with all the same value are set to 0.

See Also

fixunknowns, mapminmax, mapstd

Definition

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. If you wish to add this, you can use the following command:

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

prune

Purpose

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.

`prune(net)` takes a neural network and returns

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

Examples

Here a NARX dynamic network is created which has one external input and a second input which feeds back from the output.

```
net = narxnet(20);
view(net)
```

The network is then trained on a single random time-series problem with 50 timesteps. The external input happens to have no elements.

```
X = nndata(0,1,50);
T = nndata(1,1,50);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts);
```

The network and data are then pruned before generating a Simulink diagram and initializing its input and layer states.

```
[net2,pi,pl,po] = prune(net);
view(net)
```

```
[Xs2,Xi2,Ai2,Ts2] = prunedata(net,pi,pl,po,Xs,Xi,Ai,Ts)
[sysName,netName] = gensim(net);
setsiminit(sysName,netName,Xi2,Ai2)
```

See Also

`prunedata`, `gensim`

prunedata

Purpose

Purpose

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.

`prunedata(pi,pl,po,X,Xi,Ai,T)` takes these arguments,

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

and returns the pruned inputs, input and layer delay states, and targets.

Examples

Here a NARX dynamic network is created which has one external input and a second input which feeds back from the output.

```
net = narxnet(20);
view(net)
```

The network is then trained on a single random time-series problem with 50 timesteps. The external input happens to have no elements.

```
X = nndata(0,1,50);
T = nndata(1,1,50);
[Xs,Xi,Ai,Ts] = preparets(net,X,{},T);
net = train(net,Xs,Ts);
```

The network and data are then pruned before generating a Simulink diagram and initializing its input and layer states.

```
[net2,pi,pl,po] = prune(net);
view(net)
[Xs2,Xi2,Ai2,Ts2] = prunedata(net,pi,pl,po,Xs,Xi,Ai,Ts)
[sysName,netName] = gensim(net);
setsiminit(sysName,netName,Xi2,Ai2)
```

See Also

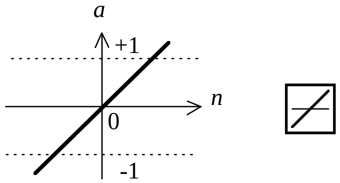
prune, gensim

purelin

Purpose

Linear transfer function

Graph and Symbol



a = purelin(n)

Linear Transfer Function

Syntax

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

Description

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

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

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

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

`purelin('dn',N,A,FP)` returns the `S` x `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`.

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

`purelin('output',FP)` returns the `[min max]` output range.

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

`purelin('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is `S` x `S` x `Q` or `S` x `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';
```

Algorithm

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

See Also

`sim`, `satlin`, `satlins`

Purpose

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 in `X` rounded to nearest multiple of `Q`.

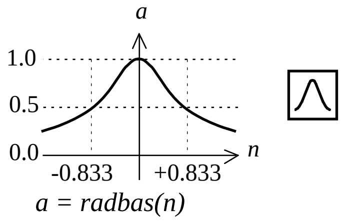
Examples

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

Purpose

Radial basis transfer function

Graph and Symbol



Radial Basis Function

Syntax

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

Description

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

`radbas(N,FP)` takes one or two inputs,

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

and returns `A`, an `S` x `Q` matrix of the radial basis function applied to each element of `N`.

`radbas('da_dn',N,A,FP)` returns the `S` x `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`.

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';
```

Algorithm

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

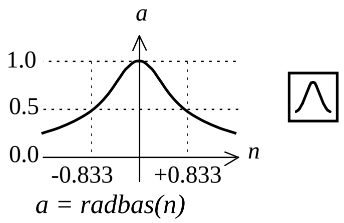
See Also

`sim`, `radbasn`, `tribas`

Purpose

Normalized radial basis transfer function

Graph and Symbol



Radial Basis Function

Syntax

```
A = radbasn(N,FP)
da_dn = radbasn('da_dn',N,A,FP)
```

Description

`radbasn` is a neural transfer function. Transfer functions calculate a layer's output from its net input. This function is equivalent to radbas, except that output vectors are normalized by dividing by the sum of the pre-normalized values.

`radbasn(N,FP)` takes one or two inputs,

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

and returns `A`, an `S` x `Q` matrix of the radial basis function applied to each element of `N`.

`radbasn('da_dn',N,A,FP)` returns the `S` x `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`.

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';
```

Algorithm

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

See Also

`sim`, `radbas`, `tribas`

Purpose

Normalized column weight initialization function

Syntax

```
W = randnc(S,PR)
W = randnc(S,R)
```

Description

`randnc` is a weight initialization function.

`randnc(S,P)` takes two inputs,

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

and returns an `S` x `R` random matrix with normalized columns.

Can also be called as `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

Purpose

Normalized row weight initialization function

Syntax

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

Description

`randnr` is a weight initialization function.

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

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

and returns an `S` x `R` random matrix with normalized rows.

Can also be called as `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`

Purpose

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.

`rands(S,PR)` takes

`S`	Number of neurons
`PR`	`R` x 2 matrix of `R` input ranges

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

`rands(S,R)` returns an `S`-by-`R` matrix of random values. `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'`. Set each `net.layerWeights{i,j}.initFcn` to `'rands'`. Set each `net.biases{i}.initFcn` to `'rands'`.

To initialize the network, call `init`.

See Also

`randsmall`, `randnr`, `randnc`, `initwb`, `initlay`, `init`

randsmall

Purpose

Small random weight/bias initialization function

Syntax

```
W = randsmall(S,PR)
M = randsmall(S,R)
v = randsmall(S);
```

Description

`randsmall` is a weight/bias initialization function.

`randsmall(S,PR)` takes

`S`	Number of neurons
`PR`	`R` x 2 matrix of `R` input ranges

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

`rands(S,R)` returns an `S`-by-`R` matrix of random values. `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'`.
   Set each `net.layerWeights{i,j}.initFcn` to `'randsmall'`.
   Set each `net.biases{i}.initFcn` to `'randsmall'`.

To initialize the network, call `init`.

See Also

`rands`, `randnr`, `randnc`, `initwb`, `initlay`, `init`

Purpose

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.

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

`dimi`	Length of layer in dimension `i`

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

Examples

This code creates and displays a two-dimensional layer with 192 neurons arranged in a 16-by-12 random pattern.

```
  pos = randtop(8,5);
  plotsompos(pos)
```

See Also

`gridtop`, `hextop`, `tritop`

regression

Purpose

Linear regression

Syntax

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

Description

`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

`regression(t,y,'one')` combines all matrix rows before regressing, returning single scalar regression, slope and offset values.

Examples

Here a feedforward network is trained and regression performed on 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)
```

See Also

`plotregression`, `confusion`

removeconstantrows

Purpose

Process matrices by removing rows with constant values

Syntax

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

Description

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

`removeconstantrows(X,max_range)` takes `X` and an optional parameter,

`X`	Single `N` x `Q` matrix or a `1` x `TS` row cell array of `N` x `Q` matrices
`max_range`	Maximum range of values for row to be removed (default is 0)

and returns

`Y`	Each `M` x `Q` matrix with `N - M` rows deleted (optional)
`PS`	Process settings that allow consistent processing of values

`removeconstantrows(X,FP)` takes parameters as a struct: `FP.max_range`.

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

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

`removeconstantrows('dy_dx',X,Y,PS)` returns the `M` x `N` x `Q` derivative of `Y` with respect to `X`.

`removeconstantrows('dx_dy',X,[],PS)` returns the reverse derivative.

Examples

Here is how to 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)
```

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)
```

Reverse the processing of y1 to get x1 again.

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

See Also

fixunknowns, mapminmax, mapstd, processpca

Purpose

Remove delay to neural network's response

Syntax

```
net = removedelay(net)
net = removedelay(net,delay)
```

Description

`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 later.

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

Examples

Here a time delay network is created, trained and simulated in its original form on an input time series `X` and target series `T`. It is then with a delay removed and then added back. These first and third outputs will be identical, while the second will be shifted by one timestep.

```
[X,T] = simpleseries_dataset;
net = timedelaynet(1:2,20);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi);
y1 = net(Xs)
net2 = removedelay(net);
[Xs,Xi,Ai,Ts] = preparets(net2,X,T);
y2 = net2(Xs,Xi)
net3 = adddelay(net2)
[Xs,Xi,Ai,Ts] = preparets(net3,X,T);
y3 = net3(Xs,Xi)
```

See Also

`adddelay`, `closeloop`, `openloop`

Purpose

Process matrices by removing rows with specified indices

Syntax

```
[y,ps] = removerows(x,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('pnames');
removerows('pcheck',fp);
```

Description

`removerows` processes matrices by removing rows with the specified indices.

`removerows(X,ind)` takes `X` and an optional parameter,

`X`	`N` x `Q` matrix or a `1` x `TS` row cell array of `N` x `Q` matrices
`ind`	Vector of row indices to remove (default is `[]`)

and returns

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

`removerows(X,FP)` takes parameters as a struct: `FP.ind`.

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

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

`removerows('dx',X,Y,PS)` returns the `M` x `N` x `Q` derivative of `Y` with respect to `X`.

`removerows('dx',X,[],PS)` returns the derivative, less efficiently.

`removerows('name')` returns the name of this process method.

`removerows('pdefaults')` returns the default process parameter structure.

`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,[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)
```

Algorithm

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

See Also

`fixunknowns`, `mapminmax`, `mapstd`, `processpca`

Purpose

Change network weights and biases to previous initialization values

Syntax

```
net = revert(net)
```

Description

`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 a two-element input (with ranges of 0 to 1 and -2 to 2) and one neuron. Once it is created, you can display the neuron's weights and bias.

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

Purpose

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`.

`roc(targets,outputs)` takes these arguments:

`targets`	`S` x `Q` matrix, where each column vector contains a single `1` value, with all other elements `0`. The index of the `1` indicates which of `S` categories that vector represents.
`outputs`	`S` x `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` x `Q` vector, where values greater or equal to `0.5` indicate class membership, and values below `0.5`, nonmembership.

and returns these values:

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

`roc(targets,outputs)` takes these arguments:

`targets`	`1` x `Q` matrix of Boolean values indicating class membership.
`outputs`	`S` x `Q` matrix, of values in `[0.1]` interval, where values greater than or equal to `0.5` indicate class membership.

and returns these values:

tpr	1 x N vector of true-positive/positive ratios.
fpr	1 x N vector of false-positive/negative ratios.
thresholds	1 x N vector of thresholds over interval [0,1].

Examples

```
load iris_dataset

net = newpr(irisInputs,irisTargets,20);

net = train(net,irisInputs,irisTargets);

irisOutputs = sim(net,irisInputs);

[tpr,fpr,thresholds] = roc(irisTargets,irisOutputs)
```

See Also

plotroc, confusion

Purpose

Sum absolute error performance function

Syntax

```
perf = sae(net.t,y,ew)
dPerf_dy = sae('dperf_dy',t,y,ew);
dPerf_dx = sae('dperf_dwb',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.

`sae(net,t,y,ew)` takes `E` and optional function parameters,

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

and returns the sum squared error.

`sae('dperf_dy',E,Y,X,perf,FP)` returns the derivative of `perf` with respect to `Y`.

`sae('dperf_dwb',E,Y,X,perf,FP)` returns the derivative of `perf` with respect to `X`.

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.

- `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);
net.performFcn = 'sae';
net = train(net,x,t)
y = sim(net,p)
e = t-y
perf = sae(net,t,y)
```

Network Use

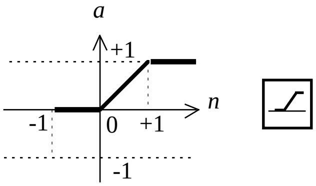
To prepare a custom network to be trained with sae, set net.performFcn to 'sae'. This automatically sets net.performParam to the default function parameters.

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

Purpose

Saturating linear transfer function

Graph and Symbol



a = satlin(n)

Satlin Transfer Function

Syntax

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

Description

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

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

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

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

`satlin('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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';
```

Algorithm

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

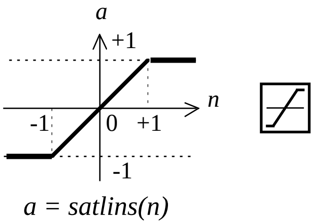
See Also

sim, poslin, satlins, purelin

Purpose

Symmetric saturating linear transfer function

Graph and Symbol



Satlins Transfer Function

Syntax

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

Description

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

`satlins(N,FP)` takes `N` and an optional argument,

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

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

`satlins('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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)
```

Algorithm

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

See Also

`sim`, `satlin`, `poslin`, `purelin`

Purpose

Scalar product weight function

Syntax

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

Description

scalprod is the scalar product weight function. Weight functions apply weights to an input to get weighted inputs.

scalprod(W,P) takes these inputs,

W	1 x 1 weight matrix
P	R x Q matrix of Q input (column) vectors

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

scalprod(code) returns information about this function. The following codes are defined:

'deriv'	Name of derivative function
'fullderiv'	Reduced derivative = 2, full derivative = 1, linear derivative = 0
'pfullderiv'	Input: reduced derivative = 2, full derivative = 1, linear derivative = 0
'wfullderiv'	Weight: reduced derivative = 2, full derivative = 1, linear derivative = 0
'name'	Full name
'fpnames'	Returns the names of function parameters
'fpdefaults'	Returns the default function parameters

scalprod('size',S,R,FP) takes the layer dimension S, input dimension R, and function parameters, and returns the weight size [1 x 1].

scalprod('dp',W,P,Z,FP) returns the derivative of Z with respect to P.

`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.inputWeight{i,j}.weightFcn` to `'scalprod'`.
For a layer weight, set `net.layerWeight{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`

Purpose

Self-organizing map

Syntax

```
selforgmap(dimensions,coverSteps,initNeighbor,topologyFcn,distance
   Fcn)
```

Description

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

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

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

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

and returns a self-organizing map.

Examples

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

```
x = simplecluster_dataset;
net = selforgmap([8 8])
net = train(net,x);
view(net)
y = net(x);
classes = vec2ind(y)
```

See Also

`lvqnet`, `competlayer`, `selforgmap`, `nctool`

separatewb

Purpose

Separate biases and weight values from a weight/bias vector

Syntax

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

Description

`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`

Purpose

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.

seq2con(S) takes one input,

s	N x TS cell array of matrices with M columns

and returns

b	N x 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

Purpose

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`

Purpose

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

Purpose

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`

Purpose

Set neural network Simulink block initial conditions

Syntax

```
setsimitinit(sysName,netName,net,xi,ai,Q)
```

Description

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

`sysName`	The name of the Simulink system containing the neural network block
`netName`	The name of the Simulink neural network block
`net`	The original neural network
`xi`	Initial input delay states
`ai`	Initial layer delay states

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`

Purpose

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

Purpose

Set all network weight and bias values with single vector

Syntax

`net = setwb(net,wb)`

Description

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

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

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

Examples

Here you create a network with a two-element input and one layer of three neurons.

```
net = newff([0 1; -1 1],[3]);
```

The network has six weights (3 neurons * 2 input elements) and three biases (3 neurons) for a total of nine weight and bias values. You can set them to random values as follows:

```
net = setwb(net,rand(9,1));
```

You can then view the weight and bias values as follows:

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

See Also

`getwb`, `formwb`, `separatewb`

Purpose

Simulate neural network

Syntax

```
[Y,Pf,Af,E,perf] = sim(net,P,Pi,Ai,T)
[Y,Pf,Af,E,perf] = sim(net,{Q TS},Pi,Ai,T)
[Y,Pf,Af,E,perf] = sim(net,Q,Pi,Ai,T)
```

To Get Help

Type `help network/sim`.

Description

`sim` simulates neural networks.

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

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

and returns

`Y`	Network outputs
`Pf`	Final input delay conditions
`Af`	Final layer delay conditions
`E`	Network errors
`perf`	Network performance

Note that arguments `Pi`, `Ai`, `Pf`, and `Af` are optional and need only be used for networks that have input or layer delays.

`sim`'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 x TS cell array	Each element P{i,ts} is an Ri x Q matrix.
Pi	Ni x ID cell array	Each element Pi{i,k} is an Ri x Q matrix.
Ai	Nl x LD cell array	Each element Ai{i,k} is an Si x Q matrix.
T	No x TS cell array	Each element P{i,ts} is a Ui x Q matrix.
Y	No x TS cell array	Each element Y{i,ts} is a Ui x Q matrix.
Pf	Ni x ID cell array	Each element Pf{i,k} is an Ri x Q matrix.
Af	Nl x LD cell array	Each element Af{i,k} is an Si x Q matrix.
E	Nt x TS cell array	Each element P{i,ts} is a Vi x Q matrix.

where

Ni = net.numInputs

Nl = net.numLayers

No = net.numOutputs

D = net.numInputDelays

LD = net.numLayerDelays

TS = Number of time steps

Q = Batch size

Ri = net.inputs{i}.size

Si = net.layers{i}.size

Ui = net.outputs{i}.size

The columns of Pi, Ai, Pf, 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 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:

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

[Y,Pf,Af] = sim(net,{Q TS},Pi,Ai) is used for networks that do not have an input, such as Hopfield networks, when cell array notation is used.

Examples

Here newp is used to create a perceptron layer with a two-element input (with ranges of [0 1]) and a single neuron.

```
net = newp([0 1;0 1],1);
```

Here the perceptron is simulated for an individual vector, a batch of three vectors, and a sequence of three vectors.

```
p1 = [.2; .9]; a1 = sim(net,p1)
p2 = [.2 .5 .1; .9 .3 .7]; a2 = sim(net,p2)
p3 = {[.2; .9] [.5; .3] [.1; .7]}; a3 = sim(net,p3)
```

Here newlin is used to create a linear layer with a three-element input and two neurons.

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

The linear layer is simulated with a sequence of two input vectors using the default initial input delay conditions (all zeros).

```
p1 = {[2; 0.5; 1] [1; 1.2; 0.1]};
[y1,pf] = sim(net,p1)
```

The layer is simulated for three more vectors, using the previous final input delay conditions as the new initial delay conditions.

```
p2 = {[0.5; 0.6; 1.8] [1.3; 1.6; 1.1] [0.2; 0.1; 0]};
[y2,pf] = sim(net,p2,pf)
```

Here newelm is used to create an Elman network with a one-element input, and a layer 1 with three tansig neurons followed by a layer 2 with two purelin neurons. Because it is an Elman network, it has a tapped delay line with a delay of 1 going from layer 1 to layer 1.

```
net = newelm([0 1],[3 2],{'tansig','purelin'});
```

The Elman network is simulated for a sequence of three values, using default initial delay conditions.

```
p1 = {0.2 0.7 0.1};
[y1,pf,af] = sim(net,p1)
```

The network is simulated for four more values, using the previous final delay conditions as the new initial delay conditions.

```
p2 = {0.1 0.9 0.8 0.4};
[y2,pf,af] = sim(net,p2,pf,af)
```

Algorithm

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 Chapter 2, "Network Objects, Data and Training Styles," for more information on network simulation.

See Also

`init`, `adapt`, `train`, `revert`

sim2nndata

Purpose

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)
```

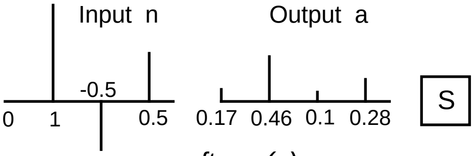
See Also

`nndata`, `nndata2sim`

Purpose

Soft max transfer function

Graph and Symbol



a = softmax(n)

Softmax Transfer Function

Syntax

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

Description

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

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

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

and returns `A`, the `S` x `Q` matrix of the softmax competitive function applied to each column of `N`.

`softmax('dn',N,A,FP)` returns the `S` x `S` x `Q` derivative of `A` with respect to `N`. If `A` or `FP` are not supplied or are set to `[]`, `FP` reverts to the default parameters, and `A` is calculated from `N`.

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

`softmax('output',FP)` returns the `[min max]` output range.

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

`softmax('fullderiv')` returns 1 or 0, depending on whether `dA_dN` is `S` x `S` x `Q` or `S` x `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';
```

Algorithm

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

See Also

sim, compet

Purpose

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.

`srchbac(net,X,Pd,Tl,Ai,Q,TS,dX,gX,perf,dperf,delta,TOL,ch_perf)` takes these inputs,

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

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point

`retcode` Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

`0` Normal

`1` Minimum step taken

`2` Maximum step taken

`3` Beta condition not met

`delta` New initial step size, based on the current step size

`tol` New tolerance on search

Parameters used for the backstepping algorithm are

`alpha` Scale factor that determines sufficient reduction in `perf`

`beta` Scale factor that determines sufficiently large step size

`low_lim` Lower limit on change in step size

`up_lim` Upper limit on change in step size

`maxstep` Maximum step length

`minstep` Minimum step length

`scale_tol` Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

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

where

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

Examples

Here is a problem consisting of inputs p and targets t to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

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

Create and Test a Network

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

Train and Retest the Network

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

Network Use

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

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

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

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

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

Algorithm

`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.

Reference

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`

Definition

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 [DeSc83]. It is used as the default line search for the quasi-Newton algorithms, although it might not be the best technique for all problems.

Purpose

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.

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

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

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point

`retcode` Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

- `0` Normal
- `1` Minimum step taken
- `2` Maximum step taken
- `3` Beta condition not met

`delta` New initial step size, based on the current step size

`tol` New tolerance on search

Parameters used for the Brent algorithm are

`alpha` Scale factor that determines sufficient reduction in `perf`

`beta` Scale factor that determines sufficiently large step size

`bmax` Largest step size

`scale_tol` Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

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

where

`Ni = net.numInputs`

`Nl = net.numLayers`

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

Examples

Here is a problem consisting of inputs p and targets t to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

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

Create and Test a Network

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

Train and Retest the Network

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

Network Use

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

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

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

Algorithm

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.

Reference

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

See Also

srchbac, srchcha, srchgol, srchhyb

Definition

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.

Purpose

1-D minimization using Charalambous' method

Syntax

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

Description

`srchcha` is a linear search routine. It searches in a given direction to locate the minimum of the performance function in that direction. It uses a technique based on Charalambous' method.

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

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

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point

`retcode` Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

`0` Normal

`1` Minimum step taken

`2` Maximum step taken

`3` Beta condition not met

`delta` New initial step size, based on the current step size

`tol` New tolerance on search

Parameters used for the Charalambous algorithm are

`alpha` Scale factor that determines sufficient reduction in `perf`

`beta` Scale factor that determines sufficiently large step size

`gama` Parameter to avoid small reductions in performance, usually set to 0.1

`scale_tol` Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

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

where

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

Examples

Here is a problem consisting of inputs p and targets t to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

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

Create and Test a Network

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

Train and Retest the Network

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

Network Use

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

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

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

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

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

Algorithm

`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).

Reference

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`

Definition

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.

Purpose

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.

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

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

and returns

`a`	Step size that minimizes performance
`gX`	Gradient at new minimum point
`perf`	Performance value at new minimum point

`retcode` Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

- `0` Normal
- `1` Minimum step taken
- `2` Maximum step taken
- `3` Beta condition not met

`delta` New initial step size, based on the current step size

`tol` New tolerance on search

Parameters used for the golden section algorithm are

`alpha` Scale factor that determines sufficient reduction in `perf`

`bmax` Largest step size

`scale_tol` Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

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

where

```
Ni   =  net.numInputs
Nl   =  net.numLayers
LD   =  net.numLayerDelays
```

```
Ri   =  net.inputs{i}.size
Si   =  net.layers{i}.size
Vi   =  net.targets{i}.size
Dij  =  Ri * length(net.inputWeights{i,j}.delays)
```

Examples

Here is a problem consisting of inputs p and targets t to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

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

Create and Test a Network

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

Train and Retest the Network

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

Network Use

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

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

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

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

Algorithm

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).

Reference

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

See Also

srchbac, srchbre, srchcha, srchhyb

Definition

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.

Purpose

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.

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

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

and returns

a	Step size that minimizes performance
gX	Gradient at new minimum point
perf	Performance value at new minimum point

`retcode` Return code that has three elements. The first two elements correspond to the number of function evaluations in the two stages of the search. The third element is a return code. These have different meanings for different search algorithms. Some might not be used in this function.

- 0 Normal
- 1 Minimum step taken
- 2 Maximum step taken
- 3 Beta condition not met

`delta` New initial step size, based on the current step size

`tol` New tolerance on search

Parameters used for the hybrid bisection-cubic algorithm are

`alpha` Scale factor that determines sufficient reduction in `perf`

`beta` Scale factor that determines sufficiently large step size

`bmax` Largest step size

`scale_tol` Parameter that relates the tolerance `tol` to the initial step size `delta`, usually set to 20

The defaults for these parameters are set in the training function that calls them. See `traincgf`, `traincgb`, `traincgp`, `trainbfg`, and `trainoss`.

Dimensions for these variables are

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

where

```
Ni  =  net.numInputs
Nl  =  net.numLayers
```

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

Examples

Here is a problem consisting of inputs `p` and targets `t` to be solved with a network.

```
p = [0 1 2 3 4 5];
t = [0 0 0 1 1 1];
```

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

Create and Test a Network

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

Train and Retest the Network

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

Network Use

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

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

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

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

Algorithm

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).

Reference

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

See Also

srchbac, srchbre, srchcha, srchgol

Definition

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.

Purpose

Sum squared error performance function

Syntax

```
perf = sse(net.t,y,ew)
dPerf_dy = sse('dperf_dy',t,y,ew);
dPerf_dx = sse('dperf_dwb',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.

`sse(net,t,y,ew)` takes `E` and optional function parameters,

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

and returns the sum squared error.

`sse('dperf_dy',E,Y,X,perf,FP)` returns the derivative of `perf` with respect to `Y`.

`sse('dperf_dwb',E,Y,X,perf,FP)` returns the derivative of `perf` with respect to `X`.

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.

- `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 = sim(net,p)
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.

Purpose

Static derivative function

Syntax

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

Description

This function calculates derivatives using the chain rule from the networks performance or outputs back to its inputs. For time series data and dynamic networks this function ignores the delay connections resulting in a approximation (which may be good or not) of the actual derivative. This function is used by Elman networks (elmannet) which is a dynamic network trained by the static derivative approximation when full derivative calculations are not available. As full derivatives are calculated by all the other derivative functions, this function is not recommended for dynamic networks except for research into training algorithms.

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

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

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, staticderiv

Purpose

Sum of absolute elements of matrix or matrices

Syntax

```
[s,n] = sumabs(x)
```

Description

`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

Purpose

Sum of squared elements of matrix or matrices

Syntax

```
[s,n] = sumsqr(x)
```

Description

`sumsqr(x)` takes a matrix or cell array of matrices and returns,

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

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

Examples

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

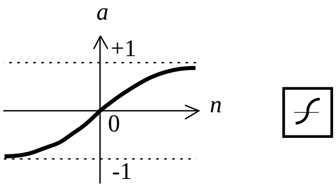
See Also

`meanabs`, `meansqr`, `sumabs`

Purpose

Hyperbolic tangent sigmoid transfer function

Graph and Symbol



a = tansig(n)

Tan-Sigmoid Transfer Function

Syntax

```
A = tansig(N,FP)
dA_dN = tansig('dn',N,A,FP)
```

Description

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

`tansig(N,FP)` takes `N` and optional function parameters,

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

and returns `A`, the `S` x `Q` matrix of `N`'s elements squashed into `[-1 1]`.

`tansig('dn',N,A,FP)` returns the 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`.

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';
```

Algorithm

```
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.

Reference

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`

Purpose

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

Purpose

Time delay neural network

Syntax

```
timedelaynet(delays,hiddenSizes,trainFcn)
```

Description

Time delay networks are similar to feedforward networks, except that the input weight has a tap delay line associated with it. This allows the network to have a finite dynamic response to time series input data. This network is also similar to the distributed delay neural network (`distdelaynet`), which has delays on the layer weights in addition to the input weight.

`timedelaynet(inputDelays,hiddenSizes,trainFcn)` takes these arguments,

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

and returns a time delay neural network.

Examples

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

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

See Also

`preparets`, `removedelay`, `distdelaynet`, `narnet`, `narxnet`

tonndata

Purpose

Convert data to standard neural network cell array form

Syntax

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

Description

`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 cell, false if they are store in matrix

and returns

`y`	Original data transformed into standard neural network cell array form
`wasMatrix`	True if original data was a matrix (as apposed to cell array)

If `columnSamples` is false, then matrix `x` or matrices in cell array `x` will be transposed, so row samples will now be stored as column vectors.

If `cellTime` is false, then matrix samples will be separated into columns of a cell array so time originally represented as vectors in a matrix will now be represented as columns of a cell array.

The returned value `wasMatrix` can be used by `fromnndata` to reverse the transformation.

Examples

Here data consisting of six timesteps of 5-element vectors is originally represented as a matrix with six columns is converted to standard neural network representation and back.

```
x = rand(5,6)
     [y,wasMatrix] = tonndata(x,true,false)
     x2 = fromnndata(y,wasMatrix,columnSamples,cellTime)
```

See Also

`nndata`, `fromnndata`, `nndata2sim`, `sim2nndata`

Purpose

Train neural network

Syntax

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

To Get Help

Type `help network/train`.

Description

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

`train(net,P,T,Pi,Ai)` takes

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

and returns

`net`	New network
`tr`	Training record (`epoch` and `perf`)
`Y`	Network outputs
`E`	Network errors
`Pf`	Final input delay conditions
`Af`	Final layer delay conditions

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

`train`'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 x TS cell array	Each element P{i,j,ts} is an Ni x Q matrix.
T	Nl x TS cell array	Each element T{i,ts} is a Ui x Q matrix.
Pi	Ni x ID cell array	Each element Pi{i,k} is an Ri x Q matrix.
Ai	Nl x LD cell array	Each element Ai{i,k} is an Si x Q matrix.
Y	No x TS cell array	Each element Y{i,ts} is a Ui x Q matrix.
E	No x TS cell array	Each element E{i,ts} is a Ui x Q matrix.
Pf	Ni x ID cell array	Each element Pf{i,k} is an Ri x Q matrix.
Af	Nl x LD cell array	Each element Af{i,k} is an Si x Q matrix.

where

Ni	=	net.numInputs
Nl	=	net.numLayers
ID	=	net.numInputDelays
LD	=	net.numLayerDelays
TS	=	Number of time steps
Q	=	Batch size
Ri	=	net.inputs{i}.size
Si	=	net.layers{i}.size

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

Pi{i,k}	=	Input i at time ts = k - ID
Pf{i,k}	=	Input i at time ts = TS + k - D
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)` x `Q` matrix
`T`	`(sum of Ui)` x `Q` matrix
`Pi`	`(sum of Ri)` x `(ID*Q)` matrix
`Ai`	`(sum of Si)` x `(LD*Q)` matrix
`Y`	`(sum of Ui)` x `Q` matrix
`E`	`(sum of Ui)` x `Q` matrix
`Pf`	`(sum of Ri)` x `(ID*Q)` matrix
`Af`	`(sum of Si)` x `(LD*Q)` matrix

Examples

Here input `P` and targets `T` define a simple function that you can plot:

```
p = [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(p,t,'o')
```

Here `newff` is used to create a two-layer feed-forward network. The network has one hidden layer with ten neurons.

```
net = feedforwardnet(10);
net = configure(net,p,t);
y1 = sim(net,p)
plot(p,t,'o',p,y1,'x')
```

The network is trained for up to 50 epochs to an error goal of 0.01 and then resimulated.

```
net.trainParam.epochs = 50;
net.trainParam.goal = 0.01;
net = train(net,p,t);
y2 = sim(net,p)
plot(p,t,'o',p,y1,'x',p,y2,'*')
```

Algorithm

`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 each epoch from concurrent input vectors (or sequences). `newc` and `newsom` return networks that use `trainr`, a training function that does this.

See Also

`init`, `revert`, `sim`, `adapt`

trainb

Purpose

Batch training with weight and bias learning rules

Syntax

```
[net,TR] = trainb(net,TR,trainV,valV,testV)
info = trainb('info')
```

Description

`trainb` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'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.

`trainb(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Initial input conditions
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `ts` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

Tl Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.

Ai Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to trainb's training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showWindow	true	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

trainb('info') returns useful information about this function.

Network Use

You can create a standard network that uses trainb by calling newlin.

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`.

See `newlin` for training examples.

Algorithm

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

`train`

Purpose

BFGS quasi-Newton backpropagation

Syntax

```
[net,TR] = trainbfg(net,TR,trainV,valV,testV)
info = trainbfg('info')
```

Description

`trainbfg` is a network training function that updates weight and bias values according to the BFGS quasi-Newton method.

`trainbfg(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Initial input conditions
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch:

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `ts` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.
`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `trainbfg`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.showWindow`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showGUI`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to 0.1 (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size

`net.trainParam.up_lim`	0.5	Upper limit on change in step size
`net.trainParam.maxstep`	100	Maximum step length
`net.trainParam.minstep`	1.0e-6	Minimum step length
`net.trainParam.bmax`	26	Maximum step size
`net.trainParam.batch_frag`	0	In case of multiple batches, they are considered independent. Any nonzero value implies a fragmented batch, so the final layer's conditions of a previous trained epoch are used as initial conditions for the next epoch.

`trainbfg('info')` returns useful information about this function.

Network Use

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

Here a feed-forward network is created with one hidden layer of 2 neurons.

```
net = newff(P,T,2,{},'trainbfg');
a = sim(net,P)
```

Here the network is trained and tested.

```
net = train(net,P,T);
a = sim(net,P)
```

Algorithm

`trainbfg` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

```
dX = -H\gX;
```

where `gX` is the gradient and `H` is a approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (*Practical Optimization*, 1981) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

Reference

Gill, Murray, & Wright, *Practical Optimization*, 1981

See Also

`traingdm`, `traingda`, `traingdx`, `trainlm`, `trainrp`, `traincgf`, `traincgb`, `trainscg`, `traincgp`, `trainoss`

Definition

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 following code trains a network using the BFGS quasi-Newton algorithm. The training parameters for `trainbfg` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainbfg');
net = train(net,p,t);
y = sim(net,p)
```

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is n x n, where n is equal to the number of weights and biases in the network. For very large networks it might be better to use Rprop or one of the conjugate gradient algorithms. For smaller networks, however, `trainbfg` can be an efficient training function.

trainbfgc

Purpose

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.

`trainbfgc(net,P,T,Pi,Ai,epochs,TS,Q)` takes these inputs,

`net`	Neural network
`P`	Delayed input vectors
`T`	Layer target vectors
`Pi`	Initial input delay conditions
`Ai`	Initial layer delay conditions
`epochs`	Number of iterations for training
`TS`	Time steps
`Q`	Batch size

and returns

`net`	Trained network	
`TR`	Training record of various values over each epoch:	
	`TR.epoch`	Epoch number
	`TR.perf`	Training performance
	`TR.vperf`	Validation performance
	`TR.tperf`	Test performance
`Y`	Network output for last epoch	
`E`	Layer errors for last epoch	
`Pf`	Final input delay conditions	

`Af`	Collective layer outputs for last epoch
`flag_stop`	Indicates if the user stopped the training

Training occurs according to `trainbfgc`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchbacxc'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to 0.1 (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size

`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length
`net.trainParam.bmax`	`26`	Maximum step size

`trainbfgc(code)` returns useful information for each `code` string:

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

Algorithm

`trainbfgc` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed according to the following formula:

```
dX = -H\gX;
```

where `gX` is the gradient and `H` is an approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (*Practical Optimization*, 1981) for a more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of `epochs` (repetitions) is reached.
- The maximum amount of `time` is exceeded.
- Performance is minimized to the `goal`.
- The performance gradient falls below `min_grad`.
- Precision problems have occurred in the matrix inversion.

Reference

Gill, Murray, and Wright, *Practical Optimization*, 1981

Purpose

Bayesian regulation backpropagation

Syntax

```
[net,TR] = trainbr(net,TR,trainV,valV,testV)
info = trainbr('info')
```

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.

trainbr(net,TR,trainV,valV,testV) takes these inputs,

net	Neural network
TR	Initial training record created by train
trainV	Training data created by train
valV	Validation data created by train
testV	Test data created by train

and returns

net	Trained network
TR	Training record of various values over each epoch

Each argument trainV, valV and testV is a structure of these fields:

X	N x TS cell array of inputs for N inputs and TS time steps. X{i,ts} is an Ri x Q matrix for the ith input and ts time step.
Xi	N x Nid cell array of input delay states for N inputs and Nid delays. Xi{i,j} is an Ri x Q matrix for the ith input and jth state.
Pd	N x S x Nid cell array of delayed input states.
T	No x TS cell array of targets for No outputs and TS time steps. T{i,ts} is an Si x Q matrix for the ith output and TS time step.

Tl Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.

Ai Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to trainbr's training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.mu	0.005	Marquardt adjustment parameter
net.trainParam.mu_dec	0.1	Decrease factor for mu
net.trainParam.mu_inc	10	Increase factor for mu
net.trainParam.mu_max	1e10	Maximum value for mu
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.mem_reduc	1	Factor to use for memory/speed tradeoff
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

trainbr('info') returns useful information about this function.

Network Use

You can create a standard network that uses trainbr with newff, newcf, or newelm. To prepare a custom network to be trained with trainbr,

1. Set NET.trainFcn to 'trainlm'. 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 newff, newcf, and newelm 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 = newff(p,t,2,{},'trainbr');
a = sim(net,p)
```

Here the network is trained and tested.

```
net = train(net,p,t);
a = sim(net,p)
```

Algorithm

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.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

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

traingdm, traingda, traingdx, trainlm, trainrp, traincgf, traincgb, trainscg, traincgp, trainbfg

trainbu

Purpose

Batch unsupervised weight/bias training

Syntax

```
[net,TR] = trainbu(net,TR,trainV,valV,testV)
```

Description

`trainbuwb` 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'`.

`trainbu(net,TR,trainV,valV,testV)` takes these inputs:

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns the following:

`NET`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV` and `testV` is a structure of these fields:

`X` — `N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.

`Xi` — `N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.

`Pd` — `N` x `S` x `Nid` cell array of delayed input states.

`T` — `No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

Tl — Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.

Ai — Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to trainbuwb's training parameters, shown here with the following default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showGUI	true	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

Validation and test vectors have no impact on training for this function, but act as independent measures of network generalization.

Network Use

You can create a standard network that uses trainbuwb by calling newsom. To prepare a custom network to be trained with trainb:

1. Set NET.trainFcn to 'trainbu'. (This option sets NET.trainParam to trainbuwb'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 newsom for training examples.

Algorithm

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

Purpose

Cyclical order weight/bias training

Syntax

```
[net,TR] = trainc(net,TR,trainV,valV,testV)
info = trainc('info')
```

Description

`trainc` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'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.

`trainc(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch:

Each argument `trainV`, `valV` and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

Tl	Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.
Ai	Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to trainc's training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showWindow	true	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

trainc('info') returns useful information about this function.

Network Use

You can create a standard network that uses trainc by calling newp. 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 newp for training examples.

Algorithm

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

train

traincgb

Purpose

Conjugate gradient backpropagation with Powell-Beale restarts

Syntax

```
[net,TR] = traincgb(net,TR,trainV,valV,testV)
info = traincgb('info')
```

Description

`traincgb` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Powell-Beale restarts.

`traincgb(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch:

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `traincgb`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step

`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length
`net.trainParam.bmax`	`26`	Maximum step size

`traincgb('info')` returns useful information about this function.

Network Use

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

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 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 feed-forward network is created with a hidden layer of 2 neurons.

```
net = newff(p,t,2,{},'traincgb');
a = sim(net,p)
```

Here the network is trained and tested.

```
net = train(net,p,t);
```

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

Algorithm

traincgb can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. See Powell, *Mathematical Programming,* Vol. 12, 1977, pp. 241 to 254, for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

Reference

Powell, M.J.D., “Restart procedures for the conjugate gradient method,” *Mathematical Programming*, Vol. 12, 1977, pp. 241–254

See Also

traingdm, traingda, traingdx, trainlm, traincgp, traincgf, trainscg, trainoss, trainbfg

Definition

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [Powe77], based on an earlier version proposed by Beale [Beal72]. This technique restarts if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality:

$$\left|\mathbf{g}_{k-1}^{T}\mathbf{g}_{k}\right| \geq 0.2\left\|\mathbf{g}_{k}\right\|^{2}$$

If this condition is satisfied, the search direction is reset to the negative of the gradient.

The following code recreates the previous network and trains it using the Powell-Beale version of the conjugate gradient algorithm. The training parameters for traincgb are the same as those for traincgf. The default line search routine srchcha is used in this example.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traincgb');
net = train(net,p,t);
y = sim(net,p)
```

The traincgb routine has somewhat better performance than traincgp for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribiére (four vectors).

traincgf

Purpose

Conjugate gradient backpropagation with Fletcher-Reeves updates

Syntax

```
[net,TR] = traincgf(net,TR,trainV,valV,testV)
info = traincgf('info')
```

Description

`traincgf` is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Fletcher-Reeves updates.

`traincgf(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `traincgf`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step

`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length
`net.trainParam.bmax`	`26`	Maximum step size

`traincgf('info')` returns useful information about this function.

Network Use

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

To prepare a custom network to be trained with `traincgf`,

1. Set `net.trainFcn` to `'traincgf'`. This sets `net.trainParam` to `traincgf`'s default parameters.
2. Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincgf`.

Examples

Here 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 feed-forward network is created with a hidden layer of 2 neurons.

```
net = newff(p,t,2,{},'traincgf');
a = sim(net,p)
```

Here the network is trained and tested.

```
net = train(net,p,t);
a = sim(net,p)
```

Algorithm

traincgf can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where dX is the search direction. The parameter a is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction, according to the formula

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

```
Z=normnew_sqr/norm_sqr;
```

where norm_sqr is the norm square of the previous gradient and normnew_sqr is the norm square of the current gradient. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

Reference

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

See Also

traingdm, traingda, traingdx, trainlm, traincgb, trainscg, traincgp, trainoss, trainbfg

Definition

All the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

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

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of the conjugate gradient algorithm are distinguished by the manner in which the constant β_k is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

The following code reinitializes the previous network and retrains it using the Fletcher-Reeves version of the conjugate gradient algorithm. The default line search routine srchcha is used in this example. traincgf generally converges in fewer iterations than trainrp (although there is more computation required in each iteration).

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traincgf');
net = train(net,p,t);
y = sim(net,p)
```

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.

traincgp

Purpose

Conjugate gradient backpropagation with Polak-Ribiére updates

Syntax

```
[net,TR] = traincgp(net,TR,trainV,valV,testV)
info = traincgp('info')
```

Description

`traincgp` is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Polak-Ribiére updates.

`traincgp(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X` — `N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.

`Xi` — `N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.

`Pd` — `N` x `S` x `Nid` cell array of delayed input states.

`T` — `No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl` — `Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.

`Ai` — `Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `traincgp`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size
`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size

`net.trainParam.maxstep`	100	Maximum step length
`net.trainParam.minstep`	1.0e-6	Minimum step length
`net.trainParam.bmax`	26	Maximum step size

Network Use

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

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 feed-forward network is created with a hidden layer of 2 neurons.

```
net = newff(p,t,2,{},'traincgp');
a = sim(net,p)
```

Here the network is trained and tested.

```
net = train(net,p,t);
a = sim(net,p)
```

Algorithm

`traincgp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`. Each variable is adjusted according to the following:

```
X = X + a*dX;
```

where `dX` is the search direction. The parameter `a` is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

```
dX = -gX + dX_old*Z;
```

where gX is the gradient. The parameter Z can be computed in several different ways. For the Polak-Ribiére variation of conjugate gradient, it is computed according to

```
Z = ((gX - gX_old)'*gX)/norm_sqr;
```

where norm_sqr is the norm square of the previous gradient, and gX_old is the gradient on the previous iteration. See page 78 of Scales (*Introduction to Non-Linear Optimization*, 1985) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

Reference

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

See Also

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

Definition

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, traincgf, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant β_k is computed by

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

The following code recreates the previous network and trains it using the Polak-Ribiére version of the conjugate gradient algorithm. The training parameters for `traincgp` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epochs` are set to the same values as they were for `traincgf`.

```
net=newff(p,t,3,{},'traincgp');
[net,tr]=train(net,p,t);
```

The `traincgp` routine has performance similar to `traincgf`. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

Purpose

Gradient descent backpropagation

Syntax

```
[net,TR] = traingd(net,TR,trainV,valV,testV)
info = traingd('info')
```

Description

traingd is a network training function that updates weight and bias values according to gradient descent.

traingd(net,TR,trainV,valV,testV) takes these inputs,

net	Neural network
TR	Initial training record created by train
trainV	Training data created by train
valV	Validation data created by train
testV	Test data created by train

and returns

net	Trained network
TR	Training record of various values over each epoch

Each argument trainV, valV, and testV is a structure of these fields:

X — N x TS cell array of inputs for N inputs and TS time steps. X{i,ts} is an Ri x Q matrix for the ith input and TS time step.

Xi — N x Nid cell array of input delay states for N inputs and Nid delays. Xi{i,j} is an Ri x Q matrix for the ith input and jth state.

Pd — N x S x Nid cell array of delayed input states.

T — No x TS cell array of targets for No outputs and TS time steps. T{i,ts} is an Si x Q matrix for the ith output and TS time step.

Tl — Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.

Ai — Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to traingd's training parameters, shown here with their default values:

net.trainParam.epochs	10	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.lr	0.01	Learning rate
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.time	inf	Maximum time to train in seconds

Network Use

You can create a standard network that uses traingd with newff, newcf, or newelm. 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 newff, help newcf, and help newelm for examples.

Algorithm

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

Definition

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” on page 9-34.

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 = newff(p,t,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 = sim(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.

Purpose

Gradient descent with adaptive learning rate backpropagation

Syntax

```
[net,TR] = traingda(net,TR,trainV,valV,testV)
info = traingda('info')
```

Description

traingda is a network training function that updates weight and bias values according to gradient descent with adaptive learning rate.

traingda(net,TR,trainV,valV,testV) takes these inputs,

net	Neural network
TR	Initial training record created by train
trainV	Training data created by train
valV	Validation data created by train
testV	Test data created by train

and returns

net	Trained network
TR	Training record of various values over each epoch

Each argument trainV, valV, and testV is a structure of these fields:

X	N x TS cell array of inputs for N inputs and TS time steps. X{i,ts} is an Ri x Q matrix for the ith input and TS time step.
Xi	N x Nid cell array of input delay states for N inputs and Nid delays. Xi{i,j} is an Ri x Q matrix for the ith input and jth state.
Pd	N x S x Nid cell array of delayed input states.
T	No x TS cell array of targets for No outputs and TS time steps. T{i,ts} is an Si x Q matrix for the ith output and TS time step.
Tl	Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.
Ai	Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to traingda's training parameters, shown here with their default values:

net.trainParam.epochs	10	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.lr	0.01	Learning rate
net.trainParam.lr_inc	1.05	Ratio to increase learning rate
net.trainParam.lr_dec	0.7	Ratio to decrease learning rate
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.max_perf_inc	1.04	Maximum performance increase
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

traingda('info') returns useful information about this function.

Network Use

You can create a standard network that uses traingda with newff, newcf, or newelm. 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 newff, help newcf, and help newelm for examples.

Algorithm

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

Definition

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

You can improve the performance of the steepest descent algorithm if you allow the learning rate to change during the training process. An adaptive learning rate attempts to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface.

An adaptive learning rate requires some changes in the training procedure used by traingd. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated.

As with momentum, if the new error exceeds the old error by more than a predefined ratio, max_perf_inc (typically 1.04), the new weights and biases are discarded. In addition, the learning rate is decreased (typically by multiplying by lr_dec = 0.7). Otherwise, the new weights, etc., are kept. If the new error is less than the old error, the learning rate is increased (typically by multiplying by lr_inc = 1.05).

This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. Thus, a near-optimal learning rate is obtained for the local terrain. When a larger learning rate could result in stable learning, the learning rate is increased. When the learning rate is too high to guarantee a decrease in error, it is decreased until stable learning resumes.

Try the *Neural Network Design* demonstration nnd12vl [HDB96] for an illustration of the performance of the variable learning rate algorithm.

Backpropagation training with an adaptive learning rate is implemented with the function traingda, which is called just like traingd, except for the additional training parameters max_perf_inc, lr_dec, and lr_inc. Here is how it is called to train the previous two-layer network:

```
p = [-1 -1 2 2; 0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'traingda');
net.trainParam.lr = 0.05;
net.trainParam.lr_inc = 1.05;
net = train(net,p,t);
y = sim(net,p)
```

Purpose

Gradient descent with momentum backpropagation

Syntax

```
[net,TR] = traingdm(net,TR,trainV,valV,testV)
info = traingdm('info')
```

Description

`traingdm` is a network training function that updates weight and bias values according to gradient descent with momentum.

`traingdm(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X` — `N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.

`Xi` — `N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.

`Pd` — `N` x `S` x `Nid` cell array of delayed input states.

`T` — `No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl` — `Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.

`Ai` — `Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to traingdm's training parameters, shown here with their default values:

net.trainParam.epochs	10	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.lr	0.01	Learning rate
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.mc	0.9	Momentum constant
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between showing progress
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

traingdm('info') returns useful information about this function.

Network Use

You can create a standard network that uses traingdm with newff, newcf, or newelm. 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 newff, help newcf, and help newelm for examples.

Algorithm

traingdm can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent with momentum,

```
dX = mc*dXprev + lr*(1-mc)*dperf/dX
```

where dXprev is the previous change to the weight or bias.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min_grad.
- Validation performance has increased more than max_fail times since the last time it decreased (when using validation).

See Also

traingd, traingda, traingdx, trainlm

Definition

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 = newff(p,t,3,{},'traingdm');
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net = train(net,p,t);
y = sim(net,p)
```

Try the *Neural Network Design* demonstration nnd12mo [HDB96] for an illustration of the performance of the batch momentum algorithm.

Purpose

Gradient descent with momentum and adaptive learning rate backpropagation

Syntax

```
[net,TR] = traingdx(net,TR,trainV,valV,testV)
info = traingdx('info')
```

Description

`traingdx` is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

`traingdx(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.
`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to traingdx's training parameters, shown here with their default values:

net.trainParam.epochs	10	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.lr	0.01	Learning rate
net.trainParam.lr_inc	1.05	Ratio to increase learning rate
net.trainParam.lr_dec	0.7	Ratio to decrease learning rate
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.max_perf_inc	1.04	Maximum performance increase
net.trainParam.mc	0.9	Momentum constant
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

traingdx('info') returns useful information about this function.

Network Use

You can create a standard network that uses traingdx with newff, newcf, or newelm. 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 newff, help newcf, and help newelm for examples.

Algorithm

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

Definition

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.

trainlm

Purpose Levenberg-Marquardt backpropagation

Syntax

```
[net,TR] = trainlm(net,TR,trainV,valV,testV)
info = trainlm('info')
```

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.

`trainlm(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

Tl	Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.
Ai	Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to trainlm's training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.mem_reduc	1	Factor to use for memory/speed tradeoff
net.trainParam.min_grad	1e-10	Minimum performance gradient
net.trainParam.mu	0.001	Initial mu
net.trainParam.mu_dec	0.1	mu decrease factor
net.trainParam.mu_inc	10	mu increase factor
net.trainParam.mu_max	1e10	Maximum mu
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.time	inf	Maximum time to train in seconds

Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for max_fail epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

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

trainlm('info') returns useful information about this function.

Network Use

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

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 newff, help newcf, and help newelm for examples.

Algorithm

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).

Definition

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}$$

where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and $\mathbf{e}$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [HaMe94]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^T\mathbf{e}$$

When the scalar μ is zero, this is just Newton’s method, using the approximate Hessian matrix. When μ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus, μ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

The following code reinitializes the previous network and retrains it using the Levenberg-Marquardt algorithm. The training parameters for trainlm are epochs, show, goal, time, min_grad, max_fail, mu, mu_dec, mu_inc, mu_max, and mem_reduc. The first six parameters were discussed earlier. The parameter mu is the initial value for μ. This value is multiplied by mu_dec whenever the performance function is reduced by a step. It is multiplied by mu_inc whenever a step would increase the performance function. If mu becomes larger than mu_max, the algorithm is stopped. The parameter mem_reduc is used to control the amount of memory used by the algorithm. It is discussed in the next section.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainlm');
net = train(net,p,t);
y = sim(net,p)
```

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.

Purpose

One-step secant backpropagation

Syntax

```
[net,TR,Ac,El] = trainoss(net,TR,trainV,valV,testV)
info = trainoss('info')
```

Description

`trainoss` is a network training function that updates weight and bias values according to the one-step secant method.

`trainoss(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.
`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `trainoss`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.searchFcn`	`'srchcha'`	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

`net.trainParam.scal_tol`	`20`	Divide into `delta` to determine tolerance for linear search.
`net.trainParam.alpha`	`0.001`	Scale factor that determines sufficient reduction in `perf`
`net.trainParam.beta`	`0.1`	Scale factor that determines sufficiently large step size
`net.trainParam.delta`	`0.01`	Initial step size in interval location step
`net.trainParam.gama`	`0.1`	Parameter to avoid small reductions in performance, usually set to `0.1` (see `srch_cha`)
`net.trainParam.low_lim`	`0.1`	Lower limit on change in step size

`net.trainParam.up_lim`	`0.5`	Upper limit on change in step size
`net.trainParam.maxstep`	`100`	Maximum step length
`net.trainParam.minstep`	`1.0e-6`	Minimum step length
`net.trainParam.bmax`	`26`	Maximum step size

`trainoss('info')` returns useful information about this function.

Network Use

You can create a standard network that uses `trainoss` with `newff`, `newcf`, or `newelm`. 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 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 `trainoss` network training function is to be used.

Create and test a network.

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

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 = sim(net,p)
```

Algorithm

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).

Reference

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

Definition

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 following code trains a network using the one-step secant algorithm. The training parameters for `trainoss` are the same as those for `traincgf`. The default line search routine `srchbac` is used in this example. The parameters `show` and `epochs` are set to 5 and 300, respectively.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainoss');
net = train(net,p,t);
y = sim(net,p)
```

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.

trainr

Purpose

Random order incremental training with learning functions

Syntax

```
[net,TR,Ac,El] = trainr(net,TR,trainV,valV,testV)
```

Description

`trainr` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'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.

`trainr(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `trainr`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

You can create a standard network that uses `trainr` by calling `newc` or `newsom`. 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 newc` and `help newsom` for training examples.

Algorithm

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`

Purpose

Resilient backpropagation

Syntax

```
[net,TR,Ac,El] = trainrp(net,TR,trainV,valV,testV)
info = trainrp('info')
```

Description

trainrp is a network training function that updates weight and bias values according to the resilient backpropagation algorithm (Rprop).

trainrp(net,TR,trainV,valV,testV) takes these inputs,

net	Neural network
TR	Initial training record created by train
trainV	Training data created by train
valV	Validation data created by train
testV	Test data created by train

and returns

net	Trained network
TR	Training record of various values over each epoch

Each argument trainV, valV, and testV is a structure of these fields:

X — N x TS cell array of inputs for N inputs and TS time steps. X{i,ts} is an Ri x Q matrix for the ith input and TS time step.

Xi — N x Nid cell array of input delay states for N inputs and Nid delays. Xi{i,j} is an Ri x Q matrix for the ith input and jth state.

Pd — N x S x Nid cell array of delayed input states.

T — No x TS cell array of targets for No outputs and TS time steps. T{i,ts} is an Si x Q matrix for the ith output and TS time step.

Tl — Nl x TS cell array of targets for Nl layers and TS time steps. Tl{i,ts} is an Si x Q matrix for the ith layer and TS time step.

Ai — Nl x TS cell array of layer delays states for Nl layers, TS time steps. Ai{i,j} is an Si x Q matrix of delayed outputs for layer i, delay j.

Training occurs according to `trainrp`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.time`	`inf`	Maximum time to train in seconds
`net.trainParam.min_grad`	`1e-6`	Minimum performance gradient
`net.trainParam.max_fail`	`5`	Maximum validation failures
`net.trainParam.lr`	`0.01`	Learning rate
`net.trainParam.delt_inc`	`1.2`	Increment to weight change
`net.trainParam.delt_dec`	`0.5`	Decrement to weight change
`net.trainParam.delta0`	`0.07`	Initial weight change
`net.trainParam.deltamax`	`50.0`	Maximum weight change

`trainrp('info')` returns useful information about this function.

Network Use

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

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 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 `trainrp` network training function is to be used.

Create and test a network.

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

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 = sim(net,p)
```

See `help newff`, `help newcf`, and `help newelm` for other examples.

Algorithm

`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, *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, 1993, pp. 586 to 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).

Reference

Riedmiller, *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, 1993, pp. 586–591

See Also

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

Definition

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 [ReBr93].

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 = newff(p,t,3,{},'trainrp');
net = train(net,p,t);
y = sim(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.

trainru

Purpose

Unsupervised random order weight/bias training

Syntax

```
[net,TR,Ac,El] = trainr(net,TR,trainV,valV,testV)
```

Description

`trainru` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'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.

`trainru(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X`	`N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.
`Xi`	`N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.
`Pd`	`N` x `S` x `Nid` cell array of delayed input states.
`T`	`No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl`	`Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.
`Ai`	`Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to `trainr`'s training parameters, shown here with their default values:

`net.trainParam.epochs`	`100`	Maximum number of epochs to train
`net.trainParam.goal`	`0`	Performance goal
`net.trainParam.show`	`25`	Epochs between displays (`NaN` for no displays)
`net.trainParam.showCommandLine`	`0`	Generate command-line output
`net.trainParam.showWindow`	`1`	Show training GUI
`net.trainParam.time`	`inf`	Maximum time to train in seconds

Network Use

To prepare a custom network to be trained with `trainru`,

1. Set `net.trainFcn` to `'trainr'`. 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`.

Algorithm

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`

Purpose

Sequential order incremental training with learning functions

Syntax

```
[net,TR,Ac,El] = trains(net,Pd,Tl,Ai,Q,TS)
info = trains(code)
```

Description

`trains` is not called directly. Instead it is called by `train` for networks whose `net.trainFcn` property is set to `'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.

`trains` takes these inputs:

`net`	Neural network
`Pd`	Delayed inputs
`Tl`	Layer targets
`Ai`	Initial input conditions
`Q`	Batch size
`TS`	Time steps

and after training the network with its weight and bias learning functions returns

`net`	Updated network	
`TR`	Training record:	
	`TR.timesteps`	Number of time steps
	`TR.perf`	Performance for each time step
`Ac`	Collective layer outputs	
`El`	Layer errors	

Training occurs according to `trains`'s training parameter, shown here with its default value:

`net.trainParam.passes`	`1`	Number of times to present sequence

Dimensions for these variables are

`Pd`	`Nl` x `Ni` x `TS` cell array	Each `Pd{i,j,ts}` is a `Dij` x `Q` matrix.
`Tl`	`Nl` x `TS` cell array	Each `Tl{i,ts}` is a `Ui` x `Q` matrix or `[]`.
`Ai`	`Nl` x `LD` cell array	Each `Ai{i,k}` is an `Si` x `Q` matrix.
`Ac`	`Nl` x `(LD+TS)` cell array	Each `Ac{i,k}` is an `Si` x `Q` matrix.
`El`	`Nl` x `TS` cell array	Each `El{i,k}` is an `Si` x `Q` matrix or `[]`.

where

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

`trains(code)` returns useful information for each `code` string:

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

Network Use

You can create a standard network that uses `trains` for adapting by calling `newp` or `newlin`.

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 newp` and `help newlin` for adaption examples.

Algorithm

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

Purpose

Scaled conjugate gradient backpropagation

Syntax

```
[net,TR,Ac,El] = trainscg(net,TR,trainV,valV,testV)
info = trainscg('info')
```

Description

`trainscg` is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

`trainscg(net,TR,trainV,valV,testV)` takes these inputs,

`net`	Neural network
`TR`	Initial training record created by `train`
`trainV`	Training data created by `train`
`valV`	Validation data created by `train`
`testV`	Test data created by `train`

and returns

`net`	Trained network
`TR`	Training record of various values over each epoch

Each argument `trainV`, `valV`, and `testV` is a structure of these fields:

`X` — `N` x `TS` cell array of inputs for `N` inputs and `TS` time steps. `X{i,ts}` is an `Ri` x `Q` matrix for the `i`th input and `TS` time step.

`Xi` — `N` x `Nid` cell array of input delay states for `N` inputs and `Nid` delays. `Xi{i,j}` is an `Ri` x `Q` matrix for the `i`th input and `j`th state.

`Pd` — `N` x `S` x `Nid` cell array of delayed input states.

`T` — `No` x `TS` cell array of targets for `No` outputs and `TS` time steps. `T{i,ts}` is an `Si` x `Q` matrix for the `i`th output and `TS` time step.

`Tl` — `Nl` x `TS` cell array of targets for `Nl` layers and `TS` time steps. `Tl{i,ts}` is an `Si` x `Q` matrix for the `i`th layer and `TS` time step.

`Ai` — `Nl` x `TS` cell array of layer delays states for `Nl` layers, `TS` time steps. `Ai{i,j}` is an `Si` x `Q` matrix of delayed outputs for layer `i`, delay `j`.

Training occurs according to trainscg's training parameters, shown here with their default values:

net.trainParam.epochs	100	Maximum number of epochs to train
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.showCommandLine	0	Generate command-line output
net.trainParam.showWindow	1	Show training GUI
net.trainParam.goal	0	Performance goal
net.trainParam.time	inf	Maximum time to train in seconds
net.trainParam.min_grad	1e-6	Minimum performance gradient
net.trainParam.max_fail	5	Maximum validation failures
net.trainParam.sigma	5.0e-5	Determine change in weight for second derivative approximation
net.trainParam.lambda	5.0e-7	Parameter for regulating the indefiniteness of the Hessian

trainscg('info') returns useful information about this function.

Network Use

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

Here 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 `trainscg` network training function is to be used.

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

Here the network is trained and retested.

```
net = train(net,p,t);
a = sim(net,p)
```

See `help newff`, `help newcf`, and `help newelm` for other examples.

Algorithm

`trainscg` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `X`.

The scaled conjugate gradient algorithm is based on conjugate directions, as in `traincgp`, `traincgf`, and `traincgb`, but this algorithm does not perform a line search at each iteration. See Moller (*Neural Networks*, Vol. 6, 1993, pp. 525 to 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).

Reference

Moller, *Neural Networks*, Vol. 6, 1993, pp. 525–533

See Also

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

Each of the other conjugate gradient algorithms requires a line search at each iteration. This line search is computationally expensive, because it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller [Moll93], was designed to avoid the time-consuming line search. This algorithm combines the model-trust region approach (used in the Levenberg-Marquardt algorithm, trainlm), with the conjugate gradient approach. See {Moll93] for a detailed explanation of the algorithm.

The following code trains a network using the scaled conjugate gradient algorithm. The training parameters for trainscg are epochs, show, goal, time, min_grad, max_fail, sigma, and lambda. The parameter sigma determines the change in the weight for the second derivative approximation. The parameter lambda regulates the indefiniteness of the Hessian.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net = newff(p,t,3,{},'trainscg');
net = train(net,p,t);
y = sim(net,p)
```

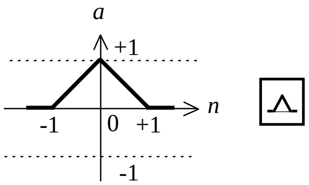
The trainscg routine can require more iterations to converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed. The storage requirements for the scaled conjugate gradient algorithm are about the same as those of Fletcher-Reeves.

tribas

Purpose

Triangular basis transfer function

Graph and Symbol



a = tribas(n)

Triangular Basis Function

Syntax

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

Description

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

`tribas(N,FP)` takes `N` and optional function parameters,

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

and returns `A`, an `S` x `Q` matrix of the triangular basis function applied to each element of `N`.

`tribas('dn',N,A,FP)` returns the `S` x `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`.

`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` x `S` x `Q` or `S` x `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';
```

Algorithm

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

See Also

`sim`, `radbas`

Purpose

Triangle layer topology function

Syntax

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

Description

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

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

`dimi`	Length of layer in dimension `i`

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

Examples

This code creates and displays a two-dimensional layer with 40 neurons arranged in an 8-by-5 triangular grid.

```
pos = tritop(8,5);
plotsompos(pos)
```

See Also

`gridtop`, `hextop`, `randtop`

Purpose

Unconfigure network inputs and outputs

Syntax

```
unconfigure(net)
unconfigure(net,'inputs')
unconfigure(net,'inputs',i)
unconfigure(net,'outputs')
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

Purpose

Convert vectors to indices

Syntax

```
ind = vec2ind(vec)
```

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.

`vec2ind(vec)` takes one argument,

`vec`	Matrix of vectors, each containing a single 1

and returns the indices of the 1s.

Examples

Here four vectors (each containing only one “1” element) are defined, and the indices of the 1s are found.

```
vec = [1 0 0 0; 0 0 1 0; 0 1 0 1]
ind = vec2ind(vec)
```

See Also

`ind2vec`

Purpose View neural network

Syntax

```
view(net)
```

Description Use this function to launch a window that shows your neural network (specified in `net`) as a graphical diagram.

Mathematical Notation

Mathematical Notation for Equations and Figures

Basic Concepts

	Description	Example
Scalars	Small *italic* letters	a, b, c
Vectors	Small **bold** nonitalic letters	$\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$
Matrices	Capital **BOLD** nonitalic letters	$\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$

Language

Vector means a column of numbers.

Weight Matrices

Scalar element	$w_{i,j}$	
Matrix	$\mathbf{W}$	
Column vector	$\mathbf{w}_j$	
Row vector	${}_i\mathbf{w}$	Vector made of ith row of weight matrix $\mathbf{W}$

Bias Elements and Vectors

Scalar element	b_i
Bias vector	$\mathbf{b}$

Time and Iteration

Weight matrix at time t	$\mathbf{W}(t)$
Weight matrix on iteration k	$\mathbf{W}(k)$

Layer Notation

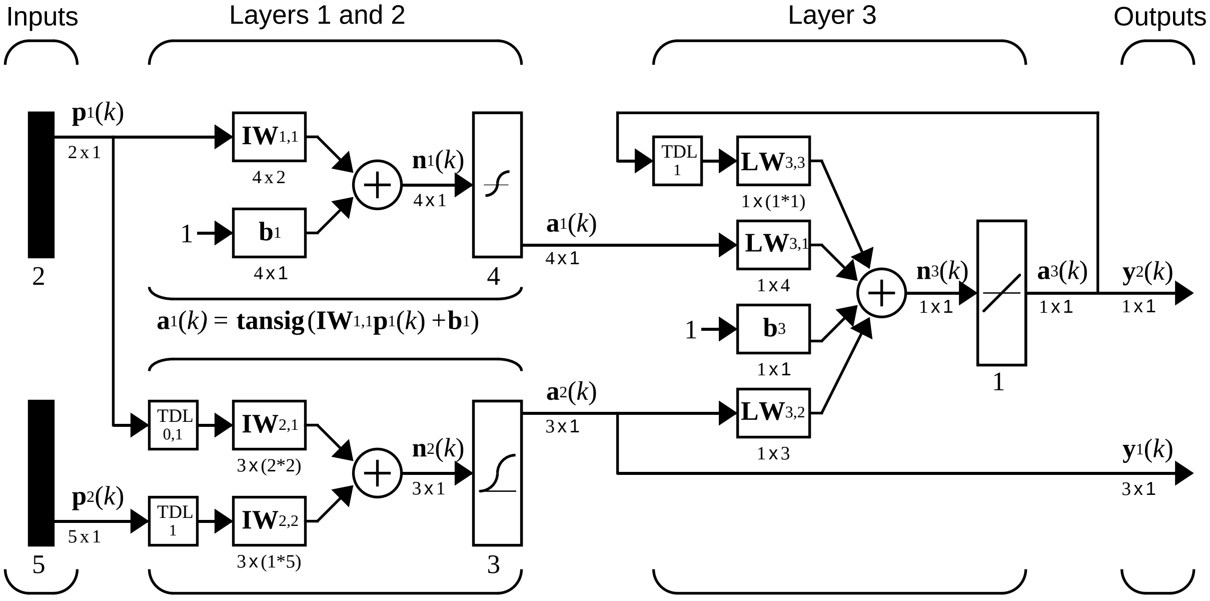
A single superscript is used to identify elements of a layer. For instance, the net input of layer 3 would be shown as $\mathbf{n}^3$.

Superscripts k, l are used to identify the source (l) connection and the destination (k) connection of layer weight matrices and input weight matrices. For instance, the layer weight matrix from layer 2 to layer 4 would be shown as $\mathbf{LW}^{4,2}$.

Input weight matrix	$\mathbf{IW}^{k,\,l}$
Layer weight matrix	$\mathbf{LW}^{k,\,l}$

Figure and Equation Examples

The following figure, taken from Chapter 9, "Advanced Topics," illustrates notation used in such advanced figures.



$$\mathbf{a}^2(k) = \mathbf{logsig}(\mathbf{IW}^{2,1}[\mathbf{p}^1(k);\mathbf{p}^1(k\text{-}1)]+\mathbf{IW}^{2,2}\mathbf{p}^2(k\text{-}1))$$

$$\mathbf{a}^3(k)=\mathbf{purelin}(\mathbf{LW}^{3,3}a^3(k\text{-}1)+\mathbf{LW}^{3,1}\ \mathbf{a}\mathbf{\mathit{1}}\ (k)+\mathbf{b}^3+\mathbf{LW}^{3,2}\mathbf{a}^2\ (\mathrm{k}))$$

Mathematics and Code Equivalents

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,

  $ab \rightarrow a*\mathrm{b}$

Figure Notation

The following equations illustrate the notation used in figures.

$$n = w_{1,1}p_{1} + w_{1,2}p_{2} + \ldots + w_{1,R}p_{R} + b$$

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

Blocks for the Simulink® Environment

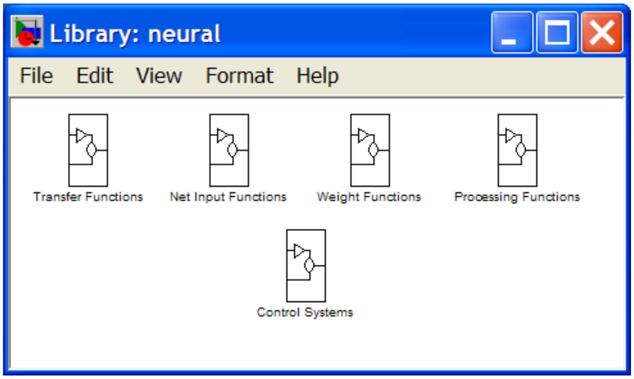
Blockset

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.

Bring up the Neural Network Toolbox blockset with this command:

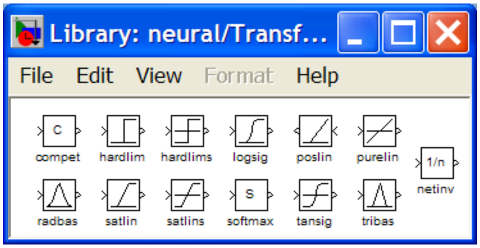
```
neural
```

The result is a window that contains five blocks. Each of these blocks contains additional blocks.



Transfer Function Blocks

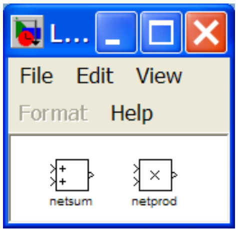
Double-click the Transfer Functions block in the Neural window to bring up a window containing several transfer function blocks.



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

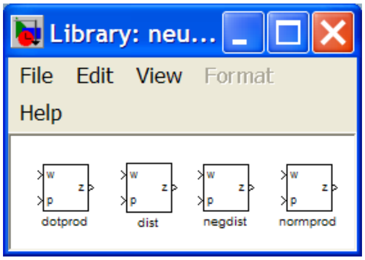
Double-click the Net Input Functions block in the Neural window to bring up a window containing two net-input function blocks.



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 window to bring up 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.

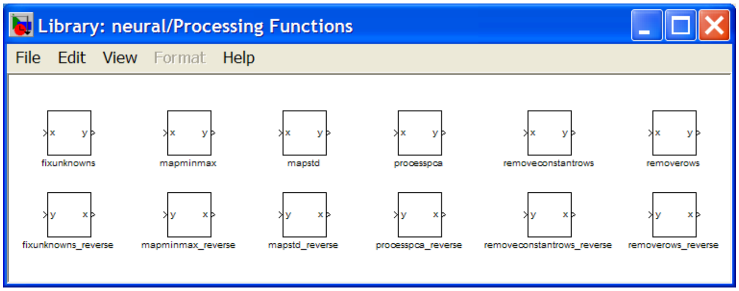
It is important to note that the blocks above expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

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 window to bring up a window containing five processing blocks and their corresponding reverse-processing blocks.



Each of these blocks can be used to pre-process inputs and post-process outputs.

Block Generation

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 tells 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 our original inputs with sim.

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

The results show the network has solved the problem.

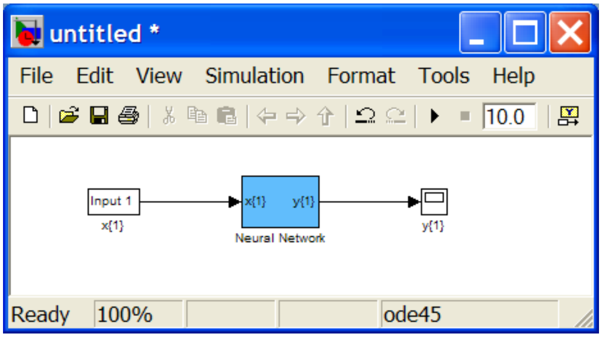
```
y =
      1.0000    3.0000    5.0000    7.0000    9.0000
```

Call gensim as follows to generate a Simulink version of the network.

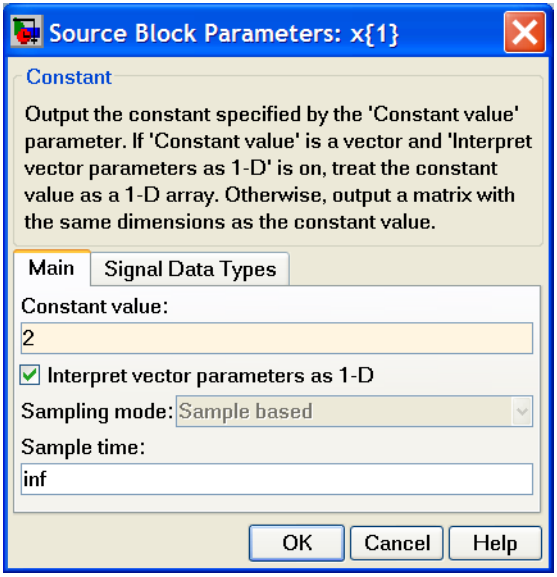
```
gensim(net,-1)
```

The second argument is -1, so the resulting network block samples continuously.

The call to gensim results in the following screen. It contains a Simulink system consisting of the linear network connected to a sample input and a scope.



To test the network, double-click the Input 1 block at left.


Source Block Parameters: x{1}

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Constant value: 2

☑ Interpret vector parameters as 1-D

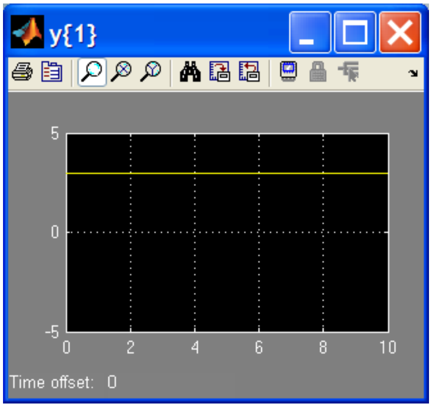
Sampling mode: Sample based

Sample time: inf

The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to 2, and then click **Close**.

Select **Start** from the **Simulation** menu. Simulink momentarily pauses as it simulates the system.

When the simulation is over, double-click the scope at 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.

Exercises

Here are a couple of exercises you can try.

Changing Input Signal

Replace the constant input block with a signal generator from the standard Simulink blockset Sources. Simulate the system and view the network's response.

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.

Code Notes

Dimensions

The following code dimensions are used in describing both the network signals that users commonly see, and those used by the utility functions:

`Ni` = Number of network inputs = `net.numInputs`

`Ri` = Number of elements in input `i` = `net.inputs{i}.size`

`Nl` = Number of layers = `net.numLayers`

`Si` = Number of neurons in layer `i` = `net.layers{i}.size`

`Nt` = Number of targets

`Vi` = Number of elements in target `i`, equal to `Sj`, where `j` is the `i`th layer with a target. (A layer `n` has a target if `net.targets(n) == 1`.)

`No` = Number of network outputs

`Ui` = Number of elements in output `i`, equal to `Sj`, where `j` is the `i`th layer with an output (A layer `n` has an output if `net.outputs(n) == 1`.)

`ID` = Number of input delays = `net.numInputDelays`

`LD` = Number of layer delays = `net.numLayerDelays`

`TS` = Number of time steps

`Q` = 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 `Ri`-by-`Q` matrix
`Pi`	Initial input delay conditions	`Ni`-by-`ID` cell array, where each element `Pi{i,k}` is an `Ri`-by-`Q` matrix
`Ai`	Initial layer delay conditions	`Nl`-by-`LD` cell array, where each element `Ai{i,k}` is an `Si`-by-`Q` matrix
`T`	Network targets	`Nt`-by-`TS` cell array, where each element `P{i,ts}` is a `Vi`-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 `Ui`-by-`Q` matrix
`E`	Network errors	`Nt`-by-`TS` cell array, where each element `P{i,ts}` is a `Vi`-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

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

Equivalently,

```
IWD(i,j) = length(net.inputWeights{i,j}.delays)
```

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

Each matrix is simply Q copies of the 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 `Tl` contains empty matrices `[]` in rows of layers `i` not associated with targets, indicated by `net.targets(i) == 0`.
`El`	Layer errors	`Nl`-by-`TS` cell array, where each element `El{i,ts}` is a `Si`-by-`Q` matrix `El` contains empty matrices `[]` in rows of layers `i` not associated with targets, indicated by `net.targets(i) == 0`.
`X`	Column vector of all weight and bias values	

Functions

The following functions are the utility functions that you can call to perform a lot of the work of simulating or training a network. You can read about them in their respective help comments.

These functions calculate signals.

```
calcpd, calca, calca1, calce, calce1, calcperf
```

These functions calculate derivatives, Jacobians, and values associated with Jacobians.

```
calcgx, calcjx, calcjejj
```

`calcgx` is used for gradient algorithms; `calcjx` and `calcjejj` can be used for calculating approximations of the Hessian for algorithms like Levenberg-Marquardt.

These functions allow network weight and bias values to be accessed and altered in terms of a single vector X.

```
setx, getx, formx
```

Code Efficiency

The functions `sim`, `train`, and `adapt` all convert a network object to a structure,

```
net = struct(net);
```

before simulation and training, and then recast the structure back to a network.

```
net = class(net,'network')
```

This is done for speed efficiency since structure fields are accessed directly, while object fields are accessed using the MATLAB® object method handling system. If users write any code that uses utility functions outside of `sim`, `train`, or `adapt`, they should use the same technique.

Argument Checking

These functions are only recommended for advanced users.

None of the utility functions do any argument checking, which means that the only feedback you get from calling them with incorrectly sized arguments is an error.

The lack of argument checking allows these functions to run as fast as possible.

For “safer” simulation and training, use `sim`, `train`, and `adapt`.

D

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] Foresee, 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*, Drodrecht, 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 demonstration 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 Chapter 9 of this 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.

[McPi43] 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.

[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 demonstrate 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. Hilton, 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.

Glossary

ADALINE Acronym for a linear neuron: ADAptive LINear Element.

adaption 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.

adaptive filter 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.

adaptive learning rate Learning rate that is adjusted according to an algorithm during training to minimize training time.

architecture 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.

backpropagation learning rule 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 *generalized delta rule*.

backtracking search Linear search routine that begins with a step multiplier of 1 and then backtracks until an acceptable reduction in performance is obtained.

batch 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 *batch* is being replaced by the more descriptive expression "concurrent vectors.")

batching Process of presenting a set of input vectors for simultaneous calculation of a matrix of output vectors and/or new weights and biases.

Bayesian framework Assumes that the weights and biases of the network are random variables with specified distributions.

BFGS quasi-Newton algorithm 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.

bias 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.

bias vector Column vector of bias values for a layer of neurons.

Brent's search Linear search that is a hybrid of the golden section search and a quadratic interpolation.

cascade-forward network Layered network in which each layer only receives inputs from previous layers.

Charalambous' search Hybrid line search that uses a cubic interpolation together with a type of sectioning.

classification Association of an input vector with a particular target vector.

competitive layer 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.

competitive learning 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.

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 the net input **n**.

concurrent input vectors 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.

conjugate gradient algorithm 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.

connection One-way link between neurons in a network.

connection strength Strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.

cycle Single presentation of an input vector, calculation of output, and new weights and biases.

dead neuron 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.

decision boundary Line, determined by the weight and bias vectors, for which the net input *n* is zero.

delta rule See **Widrow-Hoff learning rule**.

delta vector The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.

distance Distance between neurons, calculated from their positions with a distance function.

distance function Particular way of calculating distance, such as the Euclidean distance between two vectors.

early stopping 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.

epoch 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.

error jumping Sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.

error ratio Training parameter used with adaptive learning rate and momentum training of backpropagation networks.

error vector Difference between a network's output vector in response to an input vector and an associated target output vector.

feedback network Network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.

feedforward network Layered network in which each layer only receives inputs from previous layers.

Fletcher-Reeves update Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

function approximation Task performed by a network trained to respond to inputs with an approximation of a desired function.

generalization 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.

generalized regression network Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.

global minimum 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.

golden section search Linear search that does not require the calculation of the slope. The interval containing the minimum of the performance is subdivided at each iteration of the search, and one subdivision is eliminated at each iteration.

gradient descent 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.

hard-limit transfer function Transfer function that maps inputs greater than or equal to 0 to 1, and all other values to 0.

Hebb learning rule Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and postweight neurons.

hidden layer Layer of a network that is not connected to the network output (for instance, the first layer of a two-layer feedforward network).

home neuron Neuron at the center of a neighborhood.

hybrid bisection-cubic search Line search that combines bisection and cubic interpolation.

initialization Process of setting the network weights and biases to their original values.

input layer Layer of neurons receiving inputs directly from outside the network.

input space Range of all possible input vectors.

input vector Vector presented to the network.

input weight vector Row vector of weights going to a neuron.

input weights Weights connecting network inputs to layers.

Jacobian matrix Contains the first derivatives of the network errors with respect to the weights and biases.

Kohonen learning rule Learning rule that trains a selected neuron's weight vectors to take on the values of the current input vector.

layer Group of neurons having connections to the same inputs and sending outputs to the same destinations.

layer diagram 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 Chapter 2, "Network Objects, Data and Training Styles.")

layer weights Weights connecting layers to other layers. Such weights need to have nonzero delays if they form a recurrent connection (i.e., a loop).

learning Process by which weights and biases are adjusted to achieve some desired network behavior.

learning rate Training parameter that controls the size of weight and bias changes during learning.

learning rule 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.

Levenberg-Marquardt 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.

line search function Procedure for searching along a given search direction (line) to locate the minimum of the network performance.

linear transfer function Transfer function that produces its input as its output.

link distance Number of links, or steps, that must be taken to get to the neuron under consideration.

local minimum 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 = sum(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 μ.

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^*:

$$N_i(d) = \{j, d_{ij} \le d\}$$

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

neuron 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.

neuron diagram Network architecture figure showing the neurons and the weights connecting them. Each neuron's transfer function is indicated with a symbol.

ordering phase Period of training during which neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

output layer Layer whose output is passed to the world outside the network.

output vector Output of a neural network. Each element of the output vector is the output of a neuron.

output weight vector Column vector of weights coming from a neuron or input. (See also **outstar learning rule**.)

outstar learning rule 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.

overfitting 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.

pass Each traverse through all the training input and target vectors.

pattern A vector.

pattern association Task performed by a network trained to respond with the correct output vector for each input vector presented.

pattern recognition 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.

perceptron Single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.

perceptron learning rule 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.

performance Behavior of a network.

performance function Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions. Type `nnets` and look under performance functions.

Polak-Ribiére update Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

positive linear transfer function Transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.

postprocessing Converts normalized outputs back into the same units that were used for the original targets.

Powell-Beale restarts 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.

preprocessing Transformation of the input or target data before it is presented to the neural network.

principal component analysis Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.

quasi-Newton algorithm 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.

radial basis networks Neural network that can be designed directly by fitting special response elements where they will do the most good.

radial basis transfer function The transfer function for a radial basis neuron is

$$radbas(n) = e^{-n^2}$$

regularization 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.

resilient backpropagation Training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid squashing transfer functions.

saturating linear transfer function Function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1. (The toolbox function is `satlin`.)

scaled conjugate gradient algorithm Avoids the time-consuming line search of the standard conjugate gradient algorithm.

sequential input vectors 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.

sigma parameter Determines the change in weight for the calculation of the approximate Hessian matrix in the scaled conjugate gradient algorithm.

sigmoid Monotonic S-shaped function that maps numbers in the interval (-∞,∞) to a finite interval such as (-1,+1) or (0,1).

simulation Takes the network input **p**, and the network object `net`, and returns the network outputs **a**.

spread constant Distance an input vector must be from a neuron's weight vector to produce an output of 0.5.

squashing function Monotonically increasing function that takes input values between -∞ and +∞ and returns values in a finite interval.

star learning rule 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.

sum-squared error Sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.

supervised learning 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.

symmetric hard-limit transfer function Transfer that maps inputs greater than or equal to 0 to +1, and all other values to -1.

symmetric saturating linear transfer function 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.

tan-sigmoid transfer function Squashing function of the form shown below that maps the input to the interval (-1,1). (The toolbox function is `tansig`.)

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

tapped delay line Sequential set of delays with outputs available at each delay output.

target vector Desired output vector for a given input vector.

test vectors Set of input vectors (not used directly in training) that is used to test the trained network.

topology functions Ways to arrange the neurons in a grid, box, hexagonal, or random topology.

training 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.

training vector Input and/or target vector used to train a network.

transfer function Function that maps a neuron's (or layer's) net output **n** to its actual output.

tuning phase 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.

underdetermined system System that has more variables than constraints.

unsupervised learning 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.

update 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.

validation vectors Set of input vectors (not used directly in training) that is used to monitor training progress so as to keep the network from overfitting.

weight function Weight functions apply weights to an input to get weighted inputs, as specified by a particular function.

weight matrix 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`.

weighted input vector 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.

Index