

# NEURAL NETWORKS using MATLAB. CLUSTER ANALYSIS and CLASSIFICATION

K. Taylor

**NEURAL  
NETWORKS  
USING MATLAB.  
CLUSTER  
ANALYSIS AND**

# CLASSIFICATION

K. TAYLOR



# **CONTENTS**

---

**NEURAL NETWORKS WITH  
MATLAB**

## 1.1 NEURAL NETWORK TOOLBOX

## 1.2 USING NEURAL NETWORK TOOLBOX

## 1.3 AUTOMATIC SCRIPT GENERATION

## 1.4 NEURAL NETWORK TOOLBOX APPLICATIONS

## 1.5 NEURAL NETWORK DESIGN STEPS

# CLUSTER DATA WITH A SELF- ORGANIZING MAP

## 2.1 INTRODUCTION

## 2.2 USING THE NEURAL NETWORK CLUSTERING TOOL

## 2.3 USING COMMAND-LINE FUNCTIONS

# CLUSTER WITH SELF-ORGANIZING MAP NEURAL NETWORK

3.7.1 One-Dimensional Self-Organizing Map

3.7.2 Two-Dimensional Self-Organizing Map

3.7.3 Training with the Batch Algorithm

## SELF-ORGANIZING MAPS. FUNCTIONS

4.1 FUNCTIONS

4.2 NNSTART

4.3 VIEW

4.4 SELFORGMAP

4.5 TRAIN

4.6 PLOTSOMHITS

4.7 PLOTSOMNC

4.8 PLOTSOMND

4.9 PLOTSOMPLANES

4.10 PLOTSOMPOS

4.11 PLOTSOMTOP

4.12 GENFUNCTION

## **COMPETITIVE NEURAL NETWORKS**

5.1 CREATE A COMPETITIVE NEURAL NETWORK

5.1.1 Kohonen Learning Rule (learnk)

5.1.2 Bias Learning Rule (learncon)

5.1.3 Training

5.1.4 Graphical Example

## 5.2 COMPETITIVE LAYERS

5.2.1 Functions

5.3 NNSTART

5.4 VIEW

5.5 SELFORGMAP

5.6 TRAIN

5.7 PLOTSOMHITS

5.8 PLOTSOMNC

5.9 PLOTSOMND

5.10 PLOTSOMPLANES

5.11 PLOTSOMPOS

5.12 PLOTSOMTOP

5.13 GENFUNCTION

## COMPETITIVE LAYERS

6.1 FUNCTIONS

6.2 COMPETLAYER

6.3 VIEW

6.4 TRAINRU

6.5 LEARNK

6.6 LEARNCON

## CLASSIFY PATTERNS WITH A NEURAL NETWORK

7.1 INTRODUCTION

7.2 USING THE NEURAL  
NETWORK PATTERN  
RECOGNITION TOOL

7.3 USING COMMAND-LINE  
FUNCTIONS

## WORKFLOW FOR NEURAL NETWORK DESIGN

## 8.1 INTRODUCTION

## 8.2 FOUR LEVELS OF NEURAL NETWORK DESIGN

## 8.3 MULTILAYER NEURAL NETWORKS AND BACKPROPAGATION TRAINING

## 8.4 MULTILAYER NEURAL NETWORK ARCHITECTURE

### 8.4.1 Neuron Model (logsig, tansig, purelin)

### 8.4.2 Feedforward Neural Network

## 8.5 UNDERSTANDING NEURAL NETWORK TOOLBOX DATA STRUCTURES

### 8.5.1 Simulation with Concurrent Inputs in a Static Network

## 8.5.2 Simulation with Sequential Inputs in a Dynamic Network

## 8.5.3 Simulation with Concurrent Inputs in a Dynamic Network

# 8.6 NEURAL NETWORK OBJECT PROPERTIES

## 8.6.1 General

## 8.6.2 Architecture

## 8.6.3 Subobject Structures

## 8.6.4 Functions

## 8.6.5 Weight and Bias Values

# 8.7 NEURAL NETWORK SUBOBJECT PROPERTIES

## 8.7.1 Inputs

## 8.7.2 Layers

## 8.7.3 Outputs

8.7.4 Biases

8.7.5 Input Weights

8.7.6 Layer Weights

## FUNCTIONS FOR PATTERN RECOGNITION AND CLASSIFICATION

9.1 INTRODUCTION

9.2 VIEW NEURAL NETWORK

9.3 PATTERN RECOGNITION AND LEARNING VECTOR QUANTIZATION

9.3.1 Pattern recognition network:  
patternnet

9.3.2 Learning vector quantization neural network: lvqnet

9.4 TRAINING OPTIONS AND

## NETWORK PERFORMANCE

9.4.1 Receiver operating characteristic: roc

9.4.2 Plot receiver operating characteristic: plotroc

9.4.3 Plot classification confusion matrix: plotconfusion

9.4.4 Neural network performance: crossentropy

9.4.5 Construct and Train a Function Fitting Network

9.4.6 Create and train Feedforward Neural Network

9.4.7 Create and Train a Cascade Network

## 9.5 NETWORK PERFORMANCE

## 9.5.1 Description

## 9.5.2 Examples

# 9.6 FIT REGRESSION MODEL AND PLOT FITTED VALUES VERSUS TARGETS

## 9.6.1 Description

## 9.6.2 Examples

# 9.7 PLOT OUTPUT AND TARGET VALUES

## 9.7.1 Description

## 9.7.2 Examples

# 9.8 PLOT TRAINING STATE VALUES

# 9.9 PLOT PERFORMANCES

# 9.10 PLOT HISTOGRAM OF ERROR VALUES

## 9.10.1 Syntax

## 9.10.2 Description

## 9.10.3 Examples

# 9.11 GENERATE MATLAB FUNCTION FOR SIMULATING NEURAL NETWORK

## 9.11.1 Create Functions from Static Neural Network

## 9.11.2 Create Functions from Dynamic Neural Network

# 9.12 A COMPLETE EXAMPLE: HOUSE PRICE ESTIMATION

## 9.12.1 The Problem: Estimate House Values

## 9.12.2 Why Neural Networks?

## 9.12.3 Preparing the Data

## 9.12.4 Fitting a Function with a

## Neural Network

### 9.12.5 Testing the Neural Network

## 9.13 AUTOENCODER CLASS

### 9.13.1 trainAutoencoder

### 9.13.2 Construct Deep Network Using Autoencoders

### 9.13.3 decode

### 9.13.4 encode

### 9.13.5 predict

### 9.13.6 stack

## **MULTILAYER NEURAL NETWORK**

### 10.1 CREATE, CONFIGURE, AND INITIALIZE MULTILAYER NEURAL NETWORKS

#### 10.1.1 Other Related Architectures

## 10.2 FUNCTIONS FOR CREATE, CONFIGURE, AND INITIALIZE MULTILAYER NEURAL NETWORKS

10.2.1 Initializing Weights (init)

10.2.2 feedforwardnet

10.2.3 configure

10.2.4 init

10.2.5 train

10.2.6 trainlm

10.2.7 tansig

10.2.8 purelin

10.2.9 cascadeforwardnet

10.2.10 patternnet

## 10.3 TRAIN AND APPLY MULTILAYER NEURAL

# NETWORKS

10.3.1 Training Algorithms

10.3.2 Training Example

10.3.3 Use the Network

10.4 TRAIN ALGORITHMS IN  
MULTILAYER NEURAL  
NETWORKS

10.4.1 trainbr: Bayesian  
Regularization

10.4.2 trainscg: Scaled conjugate  
gradient backpropagation

10.4.3 trainrp: Resilient  
backpropagation

10.4.4 trainbfg: BFGS quasi-Newton  
backpropagation

10.4.5 traincgb: Conjugate gradient

backpropagation with Powell-Beale  
restarts

10.4.6 traincfgf: Conjugate gradient  
backpropagation with Fletcher-  
Reeves updates

10.4.7 traincgp: Conjugate gradient  
backpropagation with Polak-Ribière  
updates

10.4.8 trainoss: One-step secant  
backpropagation

10.4.9 traingdx: Gradient descent  
with momentum and adaptive learning  
rate backpropagation

10.4.10 traingdm: Gradient descent  
with momentum backpropagation

10.4.11 traingd: Gradient descent

## backpropagation

# ANALYZE AND DEPLOY TRAINED NEURAL NETWORK

## 11.1 ANALYZE NEURAL NETWORK PERFORMANCE

## 11.2 IMPROVING RESULTS

## 11.3 DEPLOYMENT FUNCTIONS AND TOOLS FOR TRAINED NETWORKS

## 11.4 GENERATE NEURAL NETWORK FUNCTIONS FOR APPLICATION DEPLOYMENT

## 11.5 DEPLOY NEURAL NETWORK SIMULINK DIAGRAMS

### 11.5.1 Example

### 11.5.2 Suggested Exercises

## 11.6 DEPLOY TRAINING OF NEURAL NETWORKS

### TRAINING SCALABILITY AND EFICIENCY

#### 12.1 NEURAL NETWORKS WITH PARALLEL AND GPU COMPUTING

12.1.1 Modes of Parallelism

12.1.2 Distributed Computing

12.1.3 Single GPU Computing

12.1.4 Distributed GPU Computing

12.1.5 Deep Learning

12.1.6 Parallel Time Series

12.1.7 Parallel Availability, Fallbacks, and Feedback

## 12.2 AUTOMATICALLY SAVE CHECKPOINTS DURING NEURAL NETWORK TRAINING

## 12.3 OPTIMIZE NEURAL NETWORK TRAINING SPEED AND MEMORY

### 12.3.1 Memory Reduction

### 12.3.2 Fast Elliot Sigmoid

## OPTIMAL SOLUTIONS

## 13.1 REPRESENTING UNKNOWN OR DON'T-CARE TARGETS

### 13.1.1 Choose Neural Network Input-Output Processing Functions

### 13.1.2 Representing Unknown or Don't-Care Targets

## 13.2 CONFIGURE NEURAL

## NETWORK INPUTS AND OUTPUTS

### 13.3 DIVIDE DATA FOR OPTIMAL NEURAL NETWORK TRAINING

### 13.4 CHOOSE A MULTILAYER NEURAL NETWORK TRAINING FUNCTION

13.4.1 SIN Data Set

13.4.2 PARITY Data Set

13.4.3 ENGINE Data Set

13.4.4 CANCER Data Set

13.4.5 CHOLESTEROL Data Set

13.4.6 DIABETES Data Set

13.4.7 Summary

### 13.5 IMPROVE NEURAL NETWORK GENERALIZATION AND AVOID OVERFITTING

13.5.1 Retraining Neural Networks

13.5.2 Multiple Neural Networks

13.5.3 Early Stopping

13.5.4 Index Data Division (divideind)

13.5.5 Random Data Division  
(dividerand)

13.5.6 Block Data Division  
(divideblock)

13.5.7 Interleaved Data Division  
(divideint)

13.5.8 Regularization

13.5.9 Modified Performance Function

13.5.10 Automated Regularization  
(trainbr)

13.5.11 Summary and Discussion of  
Early Stopping and Regularization

## 13.5.12 Posttraining Analysis (regression)

## 13.6 TRAIN NEURAL NETWORKS WITH ERROR WEIGHTS

## 13.7 NORMALIZE ERRORS OF MULTIPLE OUTPUTS

# CLASSIFICATION WITH NEURAL NETWORKS. EXAMPLES

## 14.1 CRAB CLASSIFICATION

### 14.1.1 Why Neural Networks?

### 14.1.2 Preparing the Data

### 14.1.3 Building the Neural Network Classifier

### 14.1.4 Testing the Classifier

## 14.2 WINE CLASSIFICATION

## 14.2.1 The Problem: Classify Wines

### 14.2.2 Why Neural Networks?

### 14.2.3 Preparing the Data

### 14.2.4 Pattern Recognition with a Neural Network

### 14.2.5 Testing the Neural Network

## 14.3 CANCER DETECTION

### 14.3.1 Formatting the Data

### 14.3.2 Ranking Key Features

### 14.3.3 Classification Using a Feed Forward Neural Network

## 14.4 CHARACTER RECOGNITION

### 14.4.1 Creating the First Neural Network

### 14.4.2 Training the first Neural Network

## 14.4.3 Training the Second Neural Network

## 14.4.4 Testing Both Neural Networks

# AUTOENCODERS AND CLUSTERING WITH NEURAL NETWORKS. EXAMPLES

## 15.1 TRAIN STACKED AUTOENCODERS FOR IMAGE CLASSIFICATION

### 15.1.1 Data set

### 15.1.2 Training the first autoencoder

### 15.1.3 Visualizing the weights of the first autoencoder

### 15.1.4 Training the second autoencoder

15.1.5 Training the final softmax layer

15.1.6 Forming a stacked neural network

15.1.7 Fine tuning the deep neural network

15.1.8 Summary

15.2 TRANSFER LEARNING USING CONVOLUTIONAL NEURAL NETWORKS

15.3 IRIS CLUSTERING

15.3.1 Why Self-Organizing Map Neural Networks?

15.3.2 Preparing the Data

15.3.3 Clustering with a Neural Network

## 15.4 GENE EXPRESSION ANALYSIS

15.4.1 The Problem: Analyzing Gene Expressions in Baker's Yeast (*Saccharomyces Cerevisiae*)

15.4.2 The Data

15.4.3 Filtering the Genes

15.4.4 Principal Component Analysis

15.4.5 Cluster Analysis: Self-Organizing Maps

## **SELF-ORGANIZING NETWORKS.** **EXAMPLES**

16.1 COMPETITIVE LEARNING

16.2 ONE-DIMENSIONAL SELF-ORGANIZING MAP

16.3 TWO-DIMENSIONAL SELF-

## ORGANIZING MAP

## BIBLIOGRAPHY

### 17.1 NEURAL NETWORK

## BIBLIOGRAPHY

# **Chapter 1**

# **NEURAL NETWORKS WITH MATLAB**

---

# 1.1 NEURAL NETWORK TOOLBOX

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

The toolbox includes convolutional neural network and autoencoder deep learning algorithms for image classification and feature learning tasks. To speed up training of large data sets,

you can distribute computations and data across multicore processors, GPUs, and computer clusters using Parallel Computing Toolbox.

The more important features are the following:

- Deep learning, including convolutional neural networks and autoencoders
- Parallel computing and GPU support for accelerating training (with Parallel Computing Toolbox)
- Supervised learning algorithms, including multilayer, radial basis, learning vector quantization (LVQ),

time-delay, nonlinear autoregressive (NARX), and recurrent neural network (RNN)

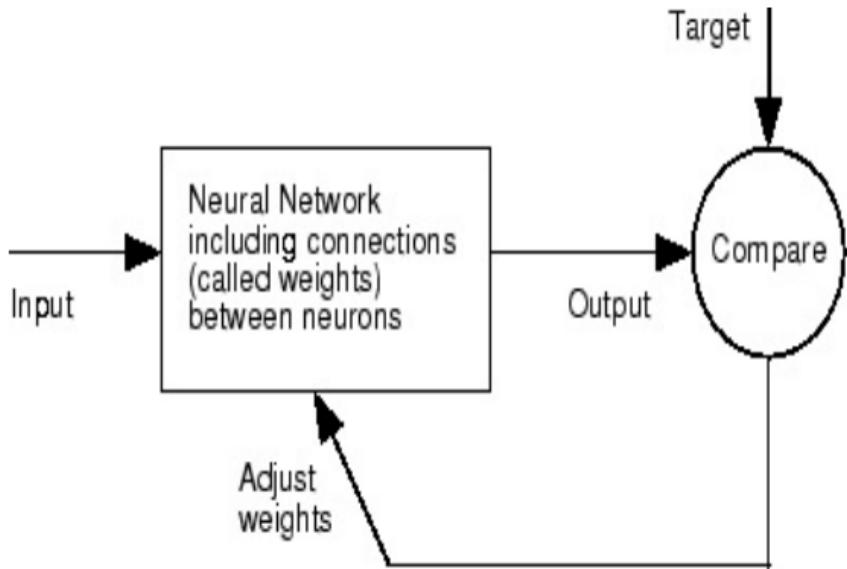
- Unsupervised learning algorithms, including self-organizing maps and competitive layers
- Apps for data-fitting, pattern recognition, and clustering
- Preprocessing, postprocessing, and network visualization for improving training efficiency and assessing network performance
- Simulink® blocks for building and evaluating neural networks and for control systems applications

Neural networks are composed of

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

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

Typically, many such input/target pairs are needed to train a network.



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

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

# 1.2 USING NEURAL NETWORK TOOLBOX

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

- The first way is through its tools. You can open any of these tools from a master tool started by the command [nnstart](#). These tools provide a convenient way to access the capabilities of the toolbox for the following tasks:
  - Function fitting ([nftool](#))
  - Pattern recognition ([nprtool](#))

- o Data clustering ([nctool](#))
- o Time-series analysis ([ntstool](#))

The second way to use the toolbox is through basic command-line operations. The command-line operations offer more flexibility than the tools, but with some added complexity. If this is your first experience with the toolbox, the tools provide the best introduction. In addition, the tools can generate scripts of documented MATLAB code to provide you with templates for creating your own customized command-line functions. The process of using the tools first, and then

generating and modifying MATLAB scripts, is an excellent way to learn about the functionality of the toolbox.

The third way to use the toolbox is through customization. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox. You can create networks with arbitrary connections, and you still be able to train them using existing toolbox training functions (as long as the network components are differentiable).

The fourth way to use the toolbox is through the ability to modify any of

the functions contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

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

# 1.3 AUTOMATIC SCRIPT GENERATION

The tools themselves form an important part of the learning process for the Neural Network Toolbox software. They guide you through the process of designing neural networks to solve problems in four important application areas, without requiring any background in neural networks or sophistication in using MATLAB. In addition, the tools can automatically generate both simple and advanced MATLAB scripts that can reproduce the steps performed by the tool, but with the option to override default settings. These scripts can provide you

with templates for creating customized code, and they can aid you in becoming familiar with the command-line functionality of the toolbox. It is highly recommended that you use the automatic script generation facility of these tools.

# 1.4 NEURAL NETWORK TOOLBOX APPLICATIONS

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

Industry	Business Applications

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

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

processing systems

Transportation      Truck brake diagnosis systems, vehicle scheduling, and routing systems

# **1.5 NEURAL NETWORK DESIGN STEPS**

The standard steps for designing neural networks to solve problems are the following:

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

There are four typical neural networks application areas: function fitting, pattern recognition, clustering, and time-series analysis.

# **Chapter 2**

## **CLUSTER DATA WITH A SELF-ORGANIZING MAP**

---

## 2.1 INTRODUCTION

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

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

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

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

- Use the [nctool](#) GUI.
- Use a command-line solution.

To define a clustering problem, simply arrange Q input vectors to be clustered as columns in an input matrix (see ["Data Structures"](#) for a detailed description of data formatting for static and time-series

data). For instance, you might want to cluster this set of 10 two-element vectors:

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

The next section shows how to train a network using the [nctool](#) GUI.

## **2.2 USING THE NEURAL NETWORK CLUSTERING TOOL**

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.

Getting Started Wizards

More Information

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 app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(npctool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

Click **Clustering Tool** to open  
the Neural Network Clustering Tool.  
(You can also use the command [nctool](#).)



## Welcome to the Neural Clustering app.

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

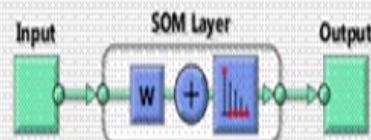
### Introduction

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 Clustering app 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 ([selforgmap](#)) 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 ([trainSOM](#), [learnSOM](#)).

To continue, click [Next].

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

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



## 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

## Summary

No inputs selected.

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

[Load Example Data Set](#)



Select inputs, then click [Next].

[Neural Network Start](#)

Welcome

Back

Next

Cancel

**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](#)

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

This dataset can be used to demonstrate how a neural network can be trained develop its own classification system for a set of examples.

LOAD [simplecluster dataset](#).MAT loads these two variables:

simplecluster/inputs - a 2x1000 matrix of 1000 two-element vectors.

[X,T] = [simplecluster dataset](#) loads the inputs and targets into variables of your own choosing.

For an intro to clustering with the [Neural Clustering app](#), click "Load Example Data Set" in the second panel and pick this dataset.

Here is how to design an 8x8 clustering neural network with this data at the command line. See [selforoma](#) for more details.

Import

Cancel

In this window, select Simple Clusters, and click Import. You return to the Select

Data window.

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. 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.



## Network Architecture

Set the number of neurons in the self-organizing map network.

### Self-Organizing Map

Define a self-organizing map. (selforgmap)

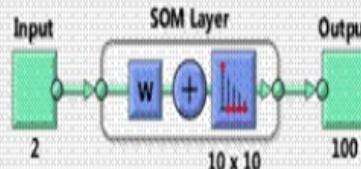
Size of two-dimensional Map:

### Recommendation

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

[Restore Defaults](#)

### Neural Network



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

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

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



## Train Network

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

### Train Network

Train using batch SOM algorithm. (trainbu) (learnsomb)



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

### Notes

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

### Results

[Plot SOM Neighbor Distances](#)

[Plot SOM Sample Hits](#)

[Plot SOM Weight Planes](#)

[Plot SOM Weight Positions](#)

 Train network, then click [Next].

 Neural Network Start

 Welcome

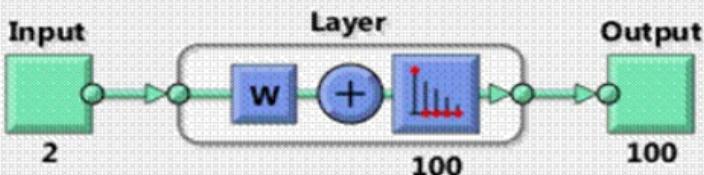
 Back

 Next

 Cancel

**Click Train.**

## Neural Network



## Algorithms

Training: Batch Weight/Bias Rules (trainbu)

Performance: Mean Squared Error (mse)

Calculations: MATLAB

## Progress

Epoch: 0 200 iterations 200

Time: 0:00:02

## Plots

[SOM Topology](#) (plotsomtop)[SOM Neighbor Connections](#) (plotsomnc)[SOM Neighbor Distances](#) (plotsomnd)[SOM Input Planes](#) (plotsomplanes)[SOM Sample Hits](#) (plotsomhits)[SOM Weight Positions](#) (plotsompos)

Plot Interval: 1 epochs



Maximum epoch reached.



Stop Training



Cancel

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

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.

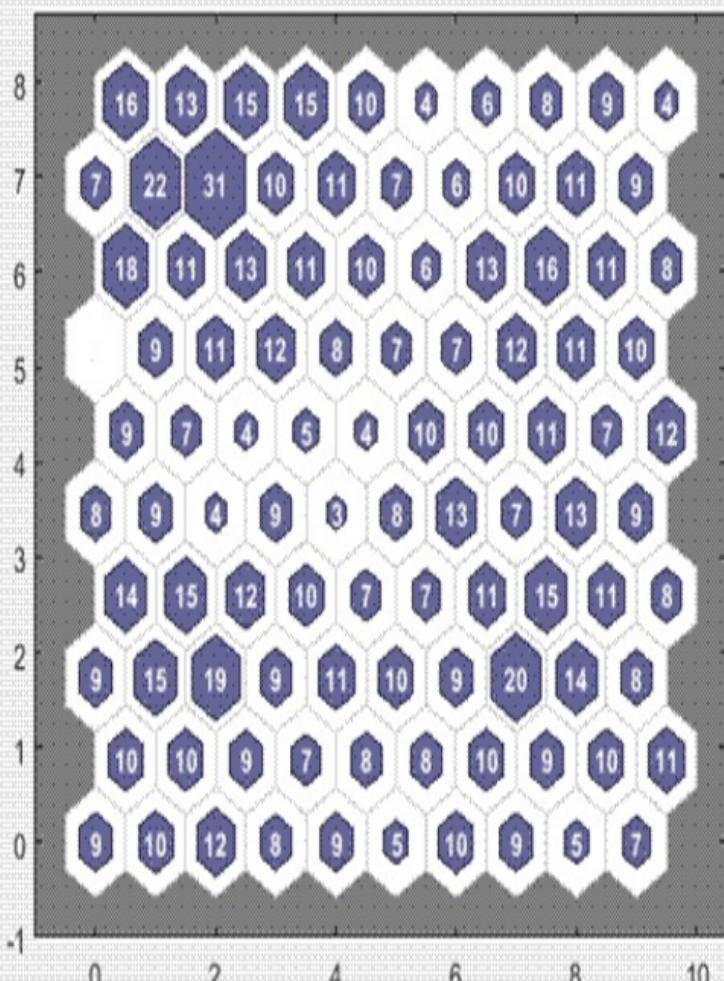
Under the **Plots** pane, click **SOM Sample Hits**.

# SOM Sample Hits (plotsomhits)



File Edit View Insert Tools Desktop Window Help

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.

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.

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.

Iterate for improved performance

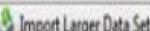
Try training again if a first try did not generate good results or you require marginal improvement.



Increase network size if retraining did not help.



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



Optionally perform additional tests



(none)



Samples are:

Matrix columns  Matrix rows

No inputs selected.



Plot SOM Neighbor Distances

Plot SOM Weight Planes

Plot SOM Sample Hits

Plot SOM Weight Positions

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

Neural Network Start

Welcome

Back

Next

Cancel

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.

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

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

Compiler tools and other MATLAB and Simulink code generation tools.



## Deploy Solution

Generate deployable versions of your trained neural network.

### Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

Generate a MATLAB function with matrix and cell array argument support:

(genFunction)



### Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



### Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

Generate a Simulink diagram:

(gensim)



### Graphics

Generate a graphical diagram of the neural network:

(network/view)



Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

Use the buttons on this screen to save your results.



## Save Results

Generate MATLAB scripts, save results and generate diagrams.

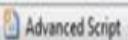
### Generate Scripts

**Recommended >>** Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool:



Generate a script with additional options and example code:



### 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

**Save Results**

Save results and click [Finish].

**Neural Network Start**

**Welcome**

**Back**

**Next**

**Finish**

You can click **Simple Script** or **Advanced Script** to create MATLAB<sup>®</sup> 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.

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.

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



## 2.3 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.

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

```
load iris_dataset  
inputs = irisInputs;
```

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. When creating the network with [selforgmap](#), you specify the number of rows and columns in the

grid:

```
dimension1 = 10;
```

```
dimension2 = 10;
```

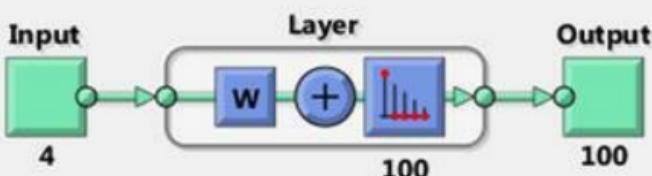
```
net = selforgmap([dimension1  
dimension2]);
```

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

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

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

## Neural Network



## Algorithms

Training: Batch Weight/Bias Rules (trainbu)

Performance: Mean Squared Error (mse)

Calculations: MATLAB

## Progress

Epoch: 0 200

Time: 0:00:00

## Plots

SOM Topology

(plotsomtop)

SOM Neighbor Connections

(plotsomnc)

SOM Neighbor Distances

(plotsomnd)

SOM Input Planes

(plotsomplanes)

SOM Sample Hits

(plotsomhits)

SOM Weight Positions

(plotsompos)

Plot Interval: 1 epochs



Maximum epoch reached.



Stop Training



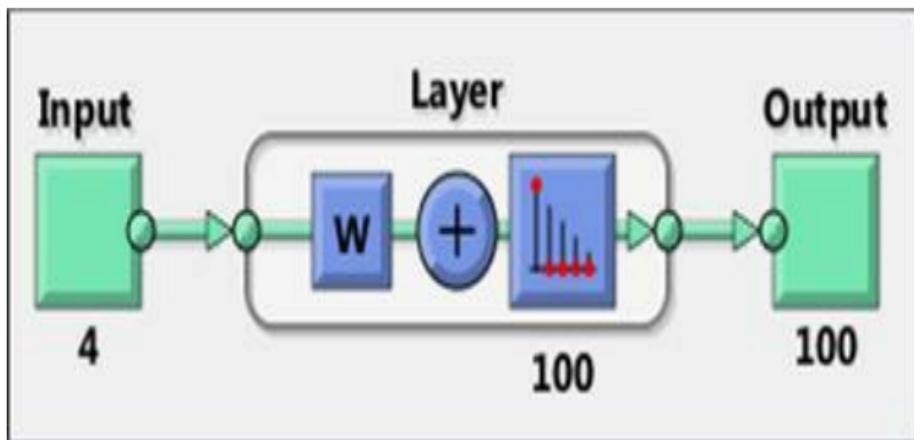
Cancel

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

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

View the network diagram.

```
view(net)
```

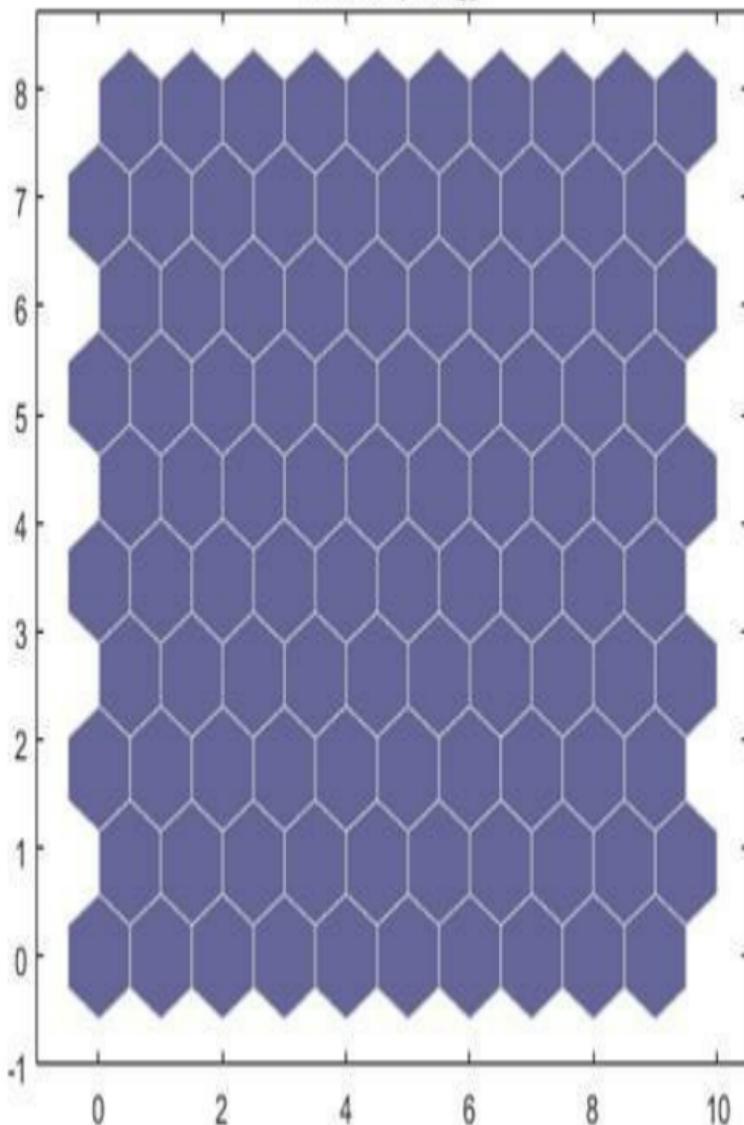


For SOM training, the weight vector

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)

### SOM Topology



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*).

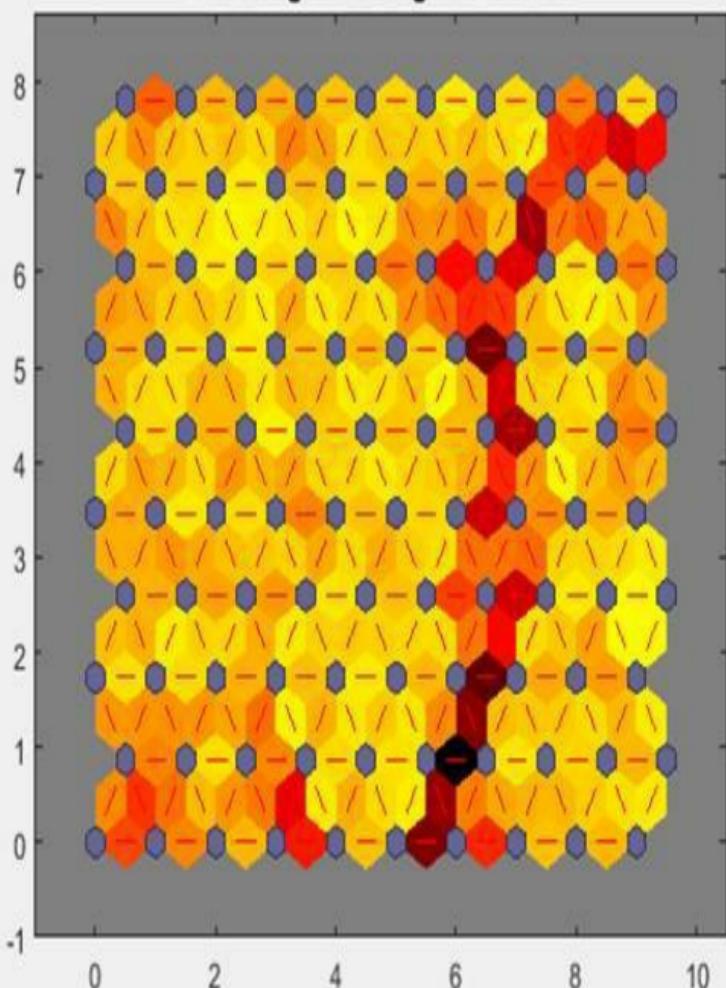
To view the U-matrix, click **SOM Neighbor Distances** in the training window.

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.

# Neural Network Training SOM Neighbor Distances (plotsomnd), Epoch 200, Maxi...

File Edit View Insert Tools Desktop Window Help

## SOM Neighbor Weight Distances



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](#), [pl](#) and [plotsomtop](#).

# **Chapter 3**

## **CLUSTER WITH SELF-ORGANIZING MAP NEURAL NETWORK**

---

## 3.1 CLUSTER WITH SELF-ORGANIZING MAP NEURAL NETWORK

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.

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:

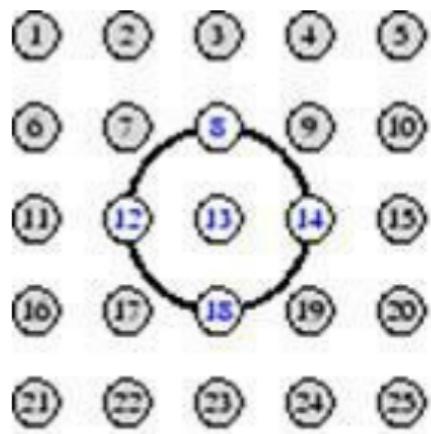
or

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

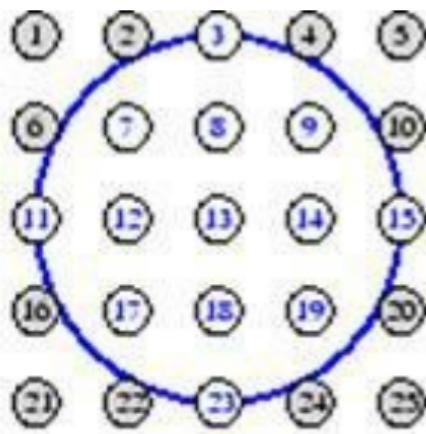
Thus, when a vector  $\mathbf{p}$  is presented, the weights of the winning neuron *and* its close neighbors move toward  $\mathbf{p}$ . Consequently, after many presentations, neighboring neurons have learned vectors similar to each other.

Another version of SOFM training, called the *batch algorithm*, presents the whole data set to the network before any weights are updated. The algorithm then determines a winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner, or for which it is in the neighborhood of a winner.

To illustrate the concept of neighborhoods, consider the figure below. The left diagram shows a two-dimensional neighborhood of radius  $d = 1$  around neuron 13. The right diagram shows a neighborhood of radius  $d = 2$ .



$$N_{13}(1)$$



$$N_{13}(2)$$

These neighborhoods could be written as  $N_{13}(1) = \{8, 12, 13, 14, 18\}$  and  $N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$ .

The neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional

arrangement, or three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

## **3.2 TOPOLOGIES (GRIDTOP, HEXTOP, RA)**

You can specify different topologies for the original neuron locations with the

functions [gridtop](#), [hextop](#), and [randtop](#).

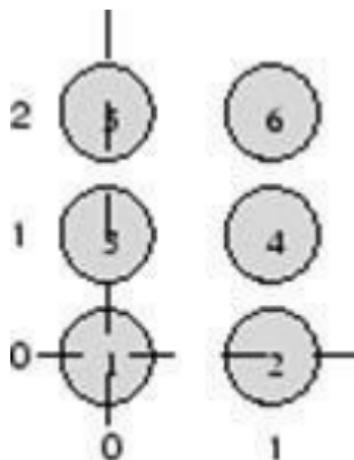
The [gridtop](#) topology starts with neurons in a rectangular grid similar to that shown in the previous figure. For example, suppose that you want a 2-by-3 array of six neurons. You can get this with

```
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)`

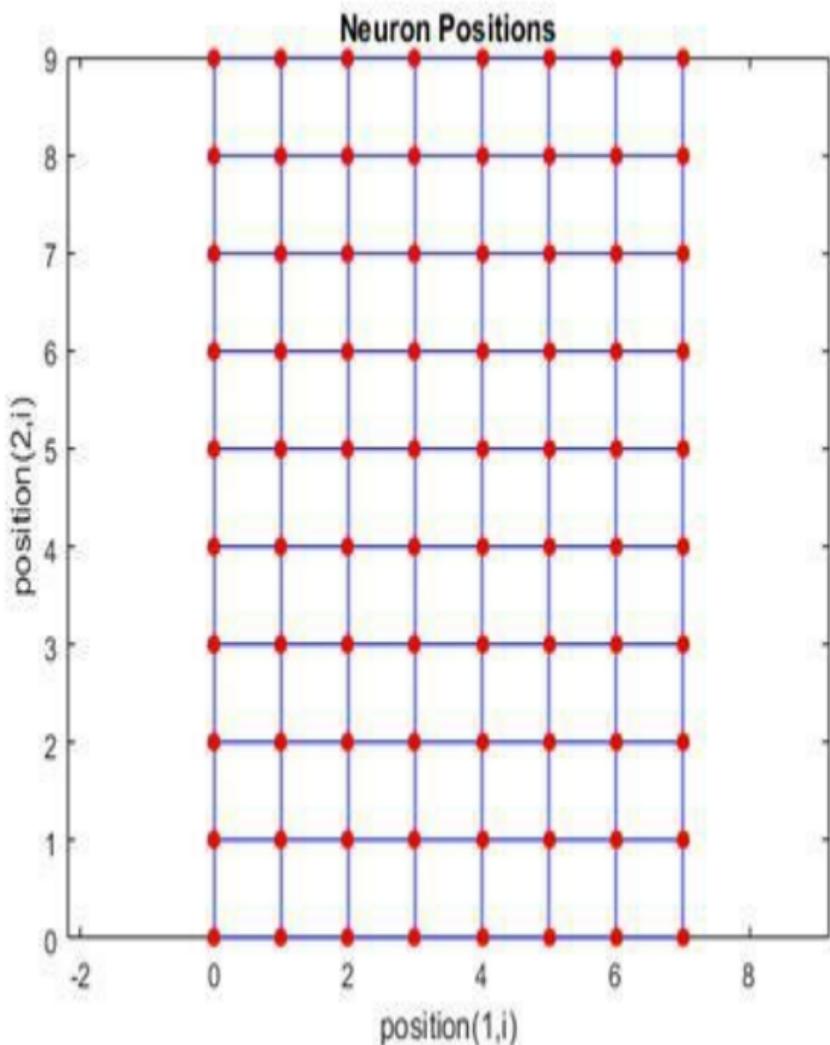
`pos =`

0    1    2    0    1    2

0 0 0 1 1 1

You can create an 8-by-10 set of neurons  
in a [gridtop](#) topology with the following  
code:

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



As shown, the neurons in

the [gridtop](#) topology do indeed lie on a grid.

The [hextop](#) function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of [hextop](#) neurons is generated as follows:

```
pos = hextop(2,3)
```

```
pos =
```

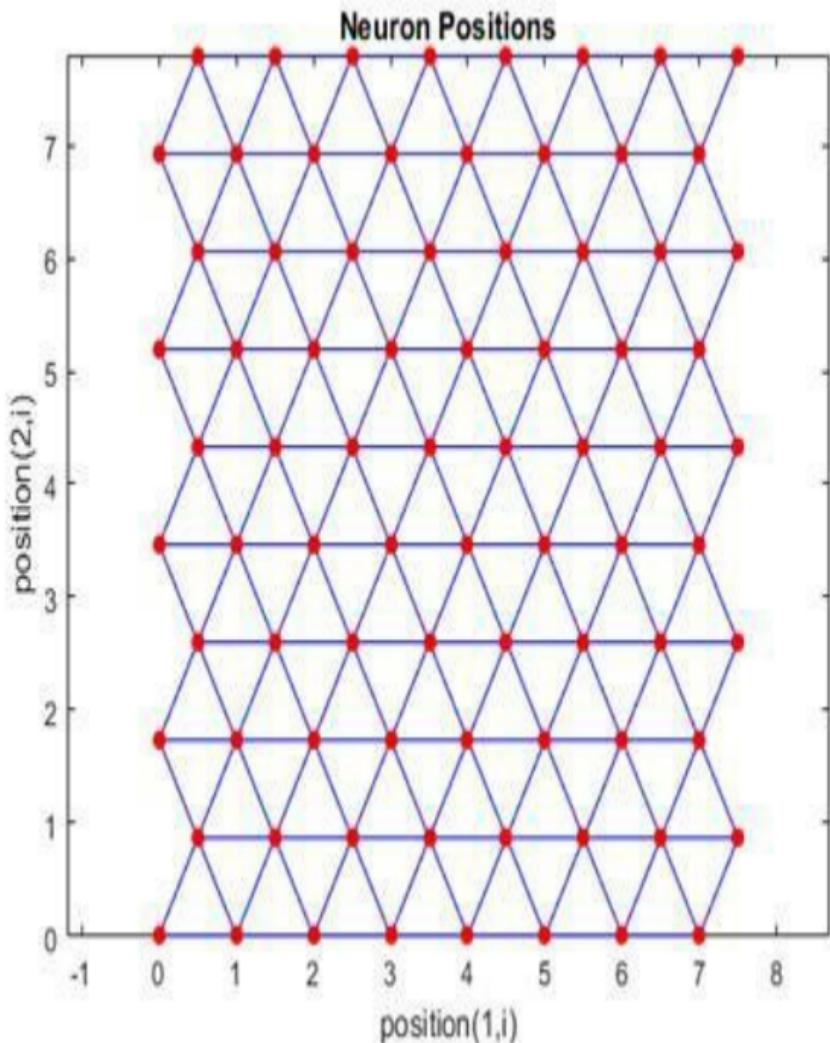
0	1.0000	0.5000
1.5000	0	1.0000
0	0	0.8660
1.7321	1.7321	0.8660

Note that [hextop](#) is the default pattern for SOM networks generated

with [selforgmap](#).

You can create and plot an 8-by-10 set of neurons in a [hextop](#) topology with the following code:

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



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

Note the positions of the neurons in a hexagonal arrangement.

Finally, the [randtop](#) function creates neurons in an N-dimensional random pattern. The following code generates a random pattern of neurons.

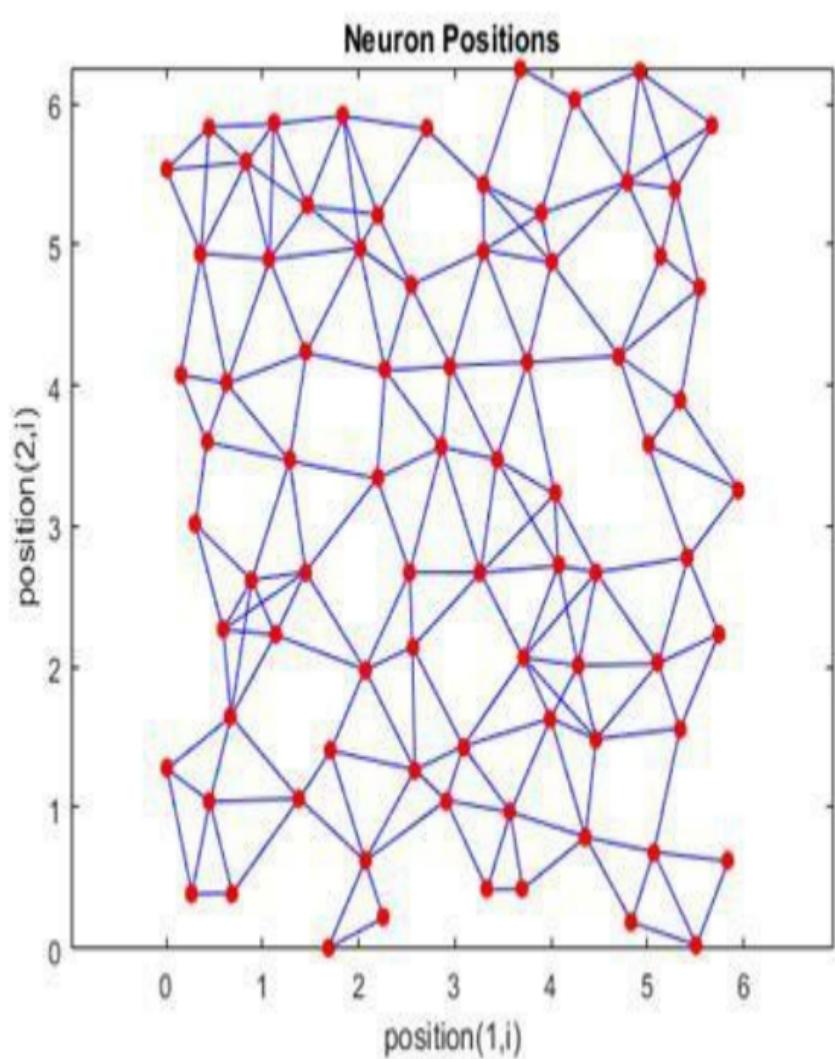
```
pos = randtop(2,3)
```

```
pos =
```

0	0.7620	0.6268
1.4218	0.0663	0.7862
0.0925	0	0.4984
0.6007	1.1222	1.4228

You can create and plot an 8-by-10 set of neurons in a [randtop](#) topology with the following code:

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



For examples, see the help for these topology functions.

## 3.3 DISTANCE FUNCTIONS (DIST, LINKDIST, MAND)

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:

$$\text{pos2} = [0 \ 1 \ 2; 0 \ 1 \ 2]$$

$$\text{pos2} =$$

$$\begin{matrix} 0 & 1 & 2 \end{matrix}$$

$$\begin{matrix} 0 & 1 & 2 \end{matrix}$$

You find the distance from each neuron to the other with

$$\text{D2} = \text{dist}(\text{pos2})$$

$$\text{D2} =$$

$$\begin{matrix} 0 & 1.4142 & 2.8284 \end{matrix}$$

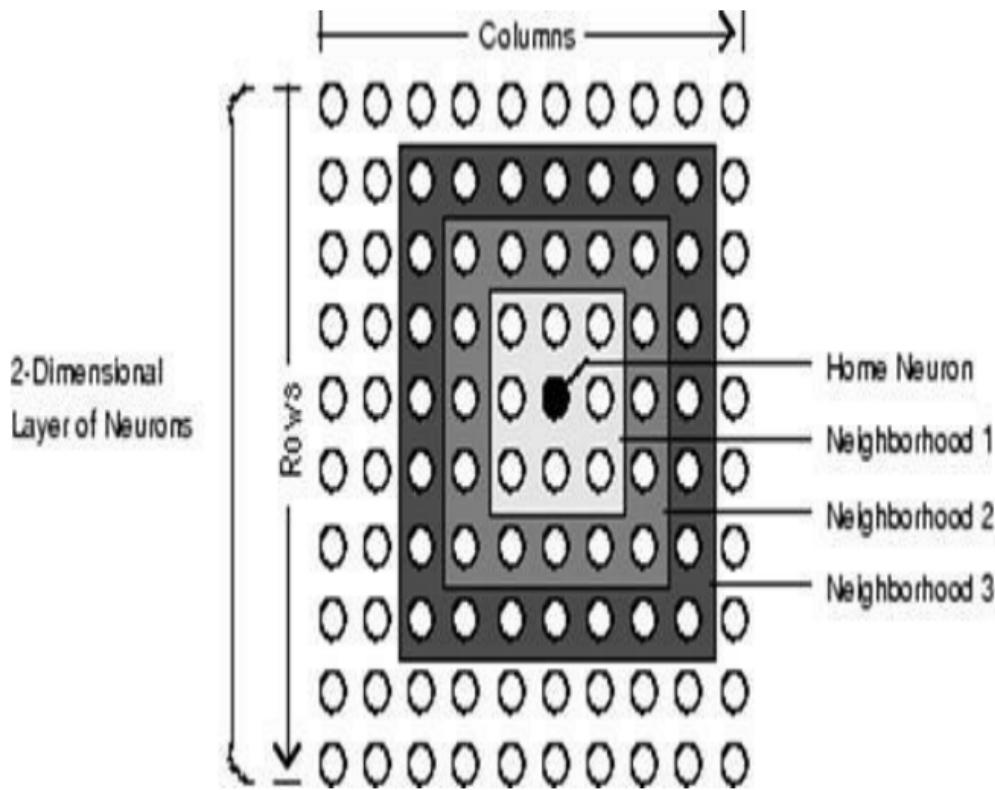
$$\begin{matrix} 1.4142 & 0 & 1.4142 \end{matrix}$$

$$\begin{matrix} 2.8284 & 1.4142 & 0 \end{matrix}$$

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.

$\text{pos} = \text{gridtop}(2,3)$

$\text{pos} =$

0	1	0	1	0	1
0	0	1	1	2	2

Then the box distances are

$\text{d} = \text{boxdist}(\text{pos})$

$\text{d} =$

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  $\mathbf{x}$  and  $\mathbf{y}$  is calculated as

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

Thus if you have

$$\mathbf{W1} = [1 \ 2; 3 \ 4; 5 \ 6]$$

$W1 =$

1 2

3 4

5 6

and

$P1 = [1;1]$

$P1 =$

1

1

then you get for the distances

$Z1 = \text{mandist}(W1, P1)$

$Z1 =$

1

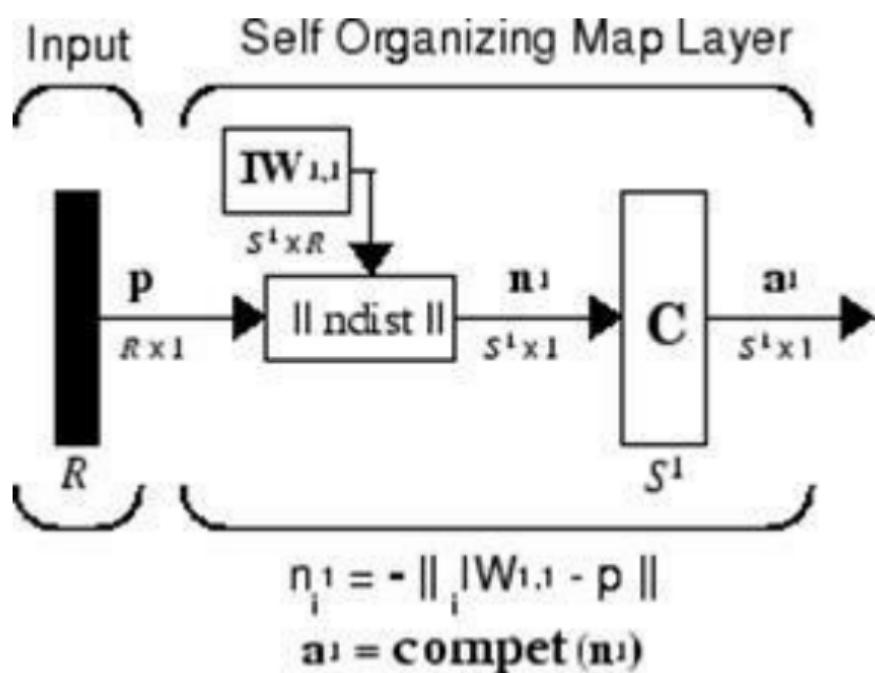
5

9

The distances calculated with [mandist](#) do indeed follow the mathematical expression given above.

### **3.4 ARCHITECTURE**

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element  $a^1_i$  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.

### **3.5 CREATE A SELF-ORGANIZING MAP NEURAL NETWORK (SELFORGMAP)**

You can create a new SOM network with the function [selforgmap](#). This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that you want to create a

network having input vectors with two elements, and that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is:

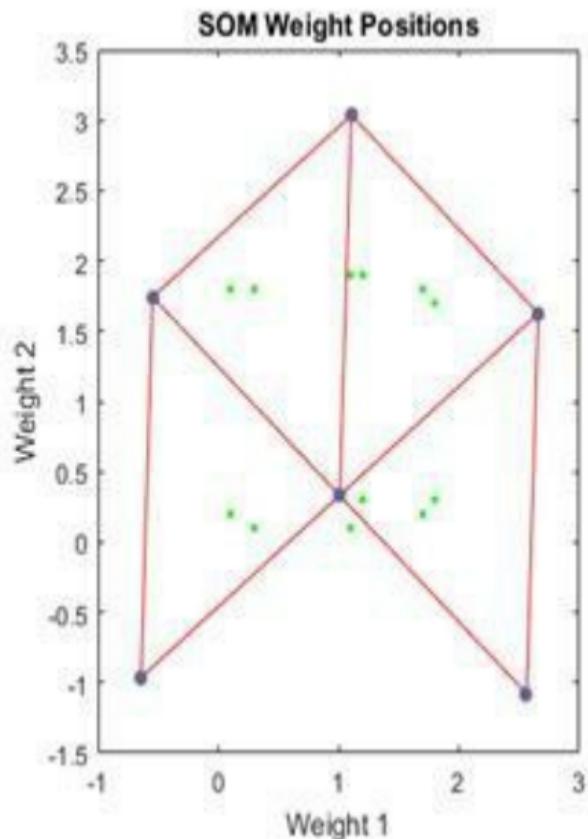
```
net = selforgmap([2,3]);
```

Suppose that the vectors to train on are:

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1  
1.8 1.7;...  
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9  
1.7 1.8];
```

You can configure the network to input the data and plot all of this with:

```
net = configure(net,P);  
plotsompos(net,P)
```



The green spots are the training vectors. The initialization for selforgmap spreads the initial weights across the input space. Note that they are initially some distance from the training vectors.

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated ([negdist](#)) to get the weighted inputs. The weighted inputs are also the net inputs ([netsum](#)). The net inputs compete ([compet](#)) so that only the neuron with the most positive net input will output a 1.

## 3.6 TRAINING (LEARNSONMB)

The default learning in a self-organizing feature map occurs in the batch mode (trainbu). The weight learning function for the self-organizing map is [learnsomb](#).

First, the network identifies the winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner or for which it is in the neighborhood of a winner. The distance that defines the size of the neighborhood is altered during training through two phases.

## Ordering Phase

This phase lasts for the given number of steps. The neighborhood distance starts at a given initial distance, and decreases to the tuning neighborhood distance (1.0). As the neighborhood distance decreases over this phase, the neurons of the network typically order themselves

in the input space with the same topology in which they are ordered physically.

## Tuning Phase

This phase lasts for the rest of training or adaption. The neighborhood size has decreased below 1 so only the winning neuron learns for each sample.

Now take a look at some of the specific values commonly used in these networks.

Learning occurs according to the [learnsomb](#) learning parameter, shown here with its default value.

Learning Parameter	Default Value

LP.init_neighborhood	3	Ini
LP.steps	100	Or

The neighborhood size NS is altered through two phases: an ordering phase and a tuning phase.

The ordering phase lasts as many steps as LP.steps. During this phase, the algorithm adjusts ND from the initial neighborhood size LP.init\_neighborhood down to 1. It is during this phase that neuron weights order themselves in the input space consistent with the associated neuron positions.

During the tuning phase, ND is less than 1. During this phase, the weights are expected to spread out relatively evenly

over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron's weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size decreases to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered. The training continues in order to give the neurons time to spread out evenly across the input vectors.

As with competitive layers, the neurons of a self-organizing map will order

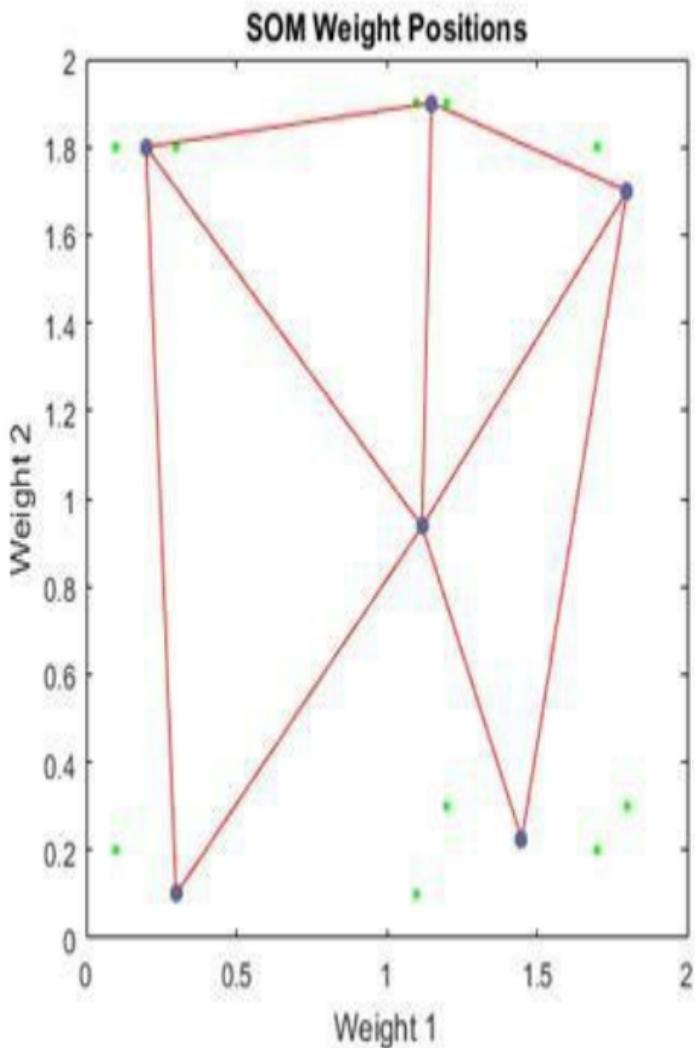
themselves with approximately equal distances between them if input vectors appear with even probability throughout a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

You can train the network for 1000 epochs with

```
net.trainParam.epochs = 1000;  
net = train(net,P);
```

plotsompos(net,P)



You can see that the neurons have started to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

## 3.7 EXAMPLES

Two examples are described briefly below. You also might try the similar examples `demosml` and `demosm2`.

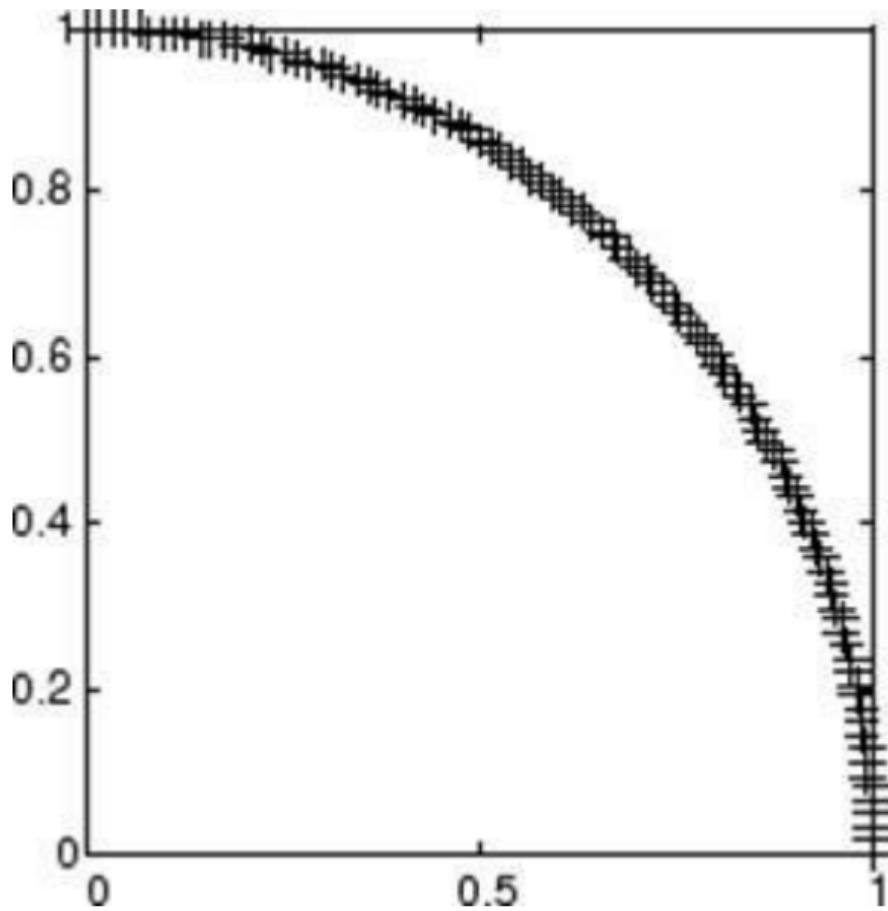
### 3.7.1 One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between  $0^\circ$  and  $90^\circ$ .

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

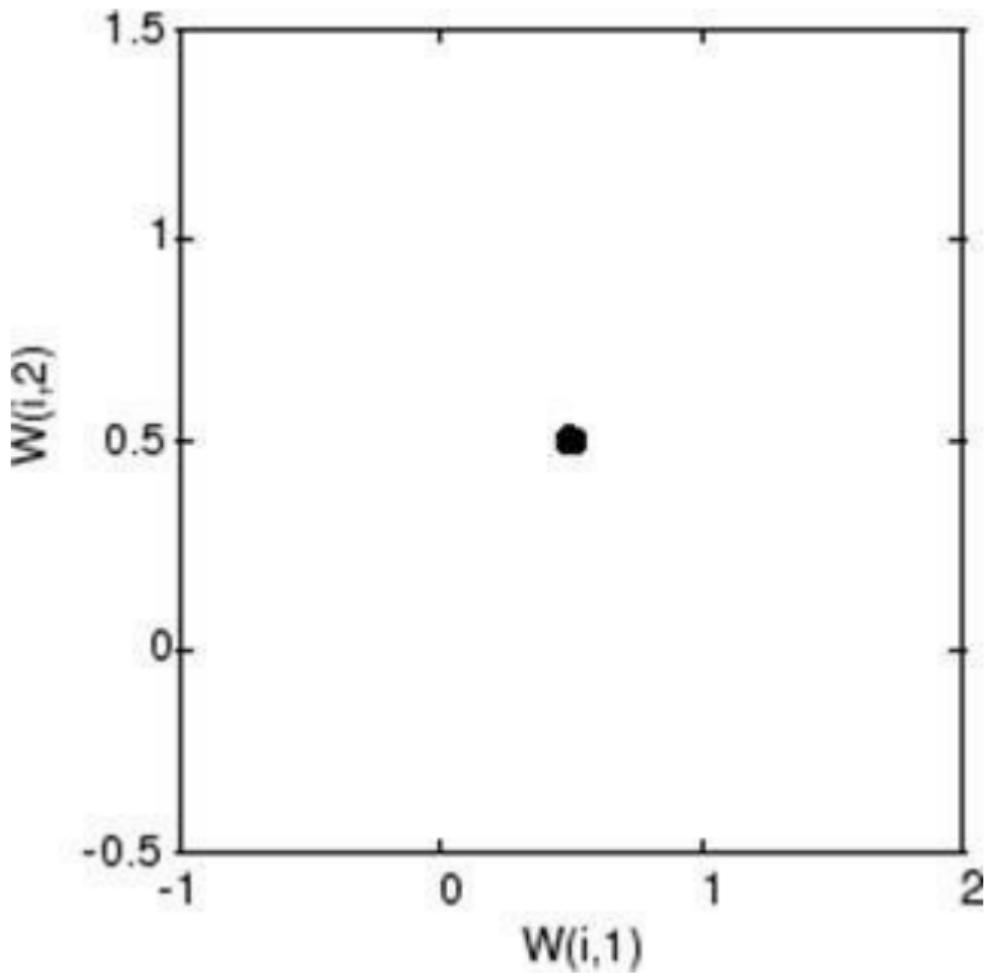
Here is a plot of the data.

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



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

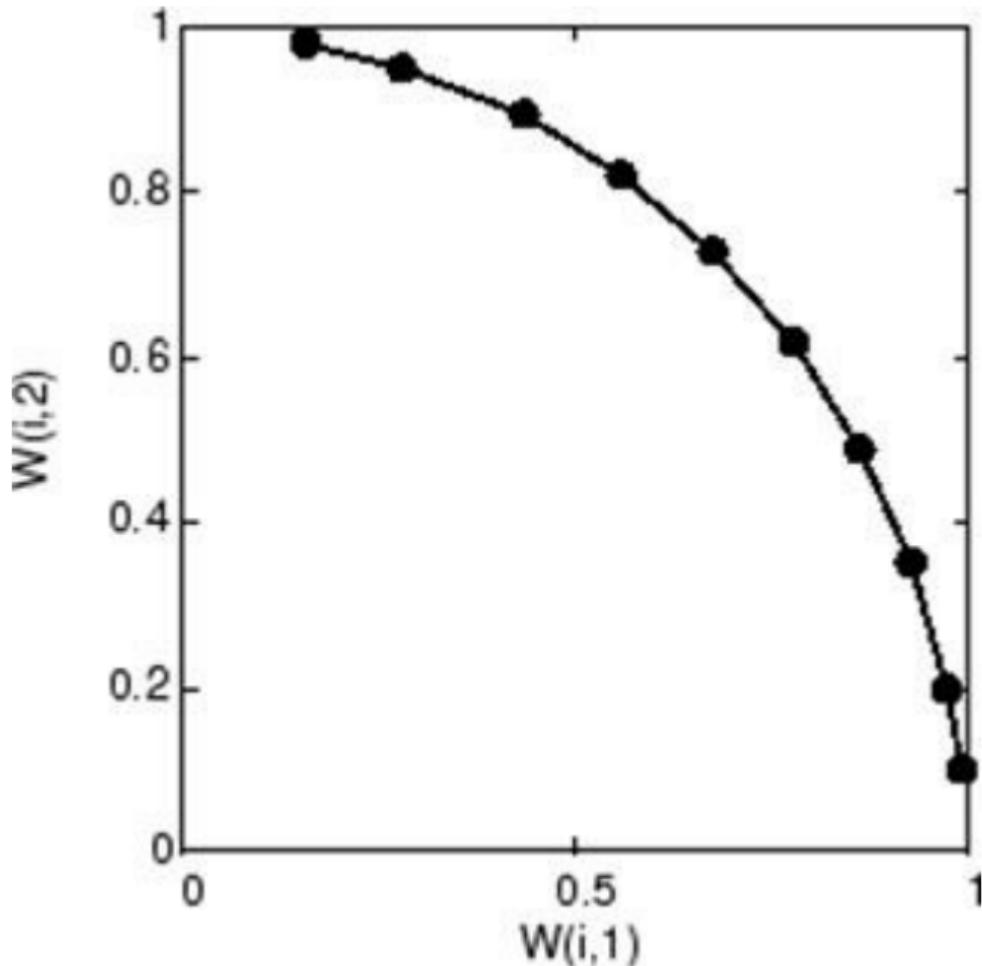
neurons are at the center of the figure.



Of course, because all the weight

vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial

vectors does not guarantee identical training results.

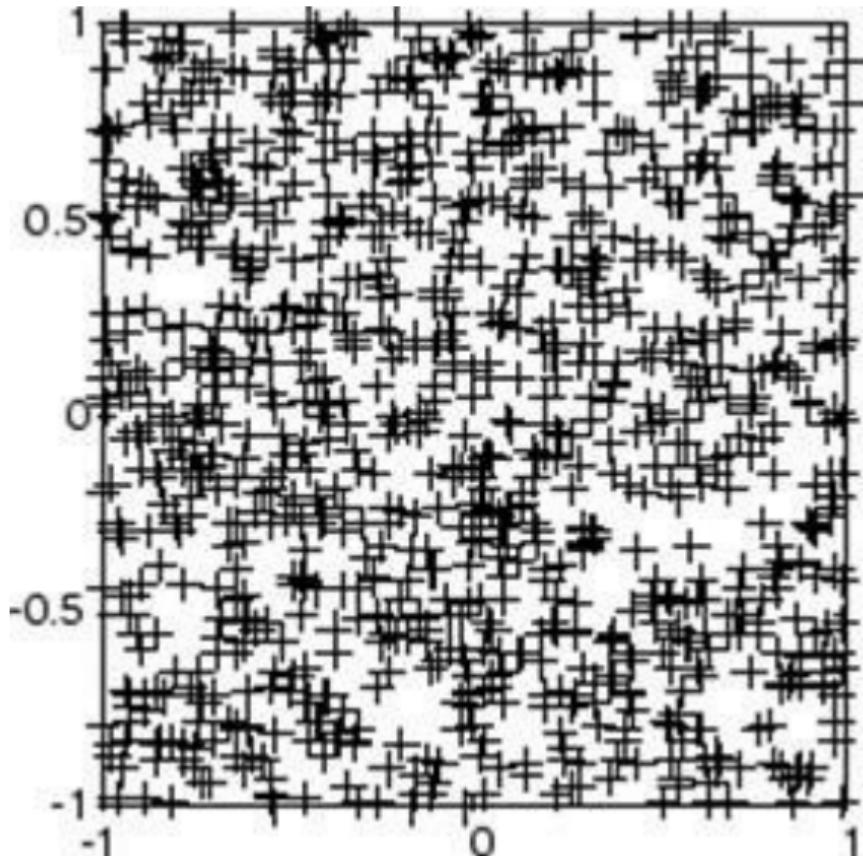
## 3.7.2 Two-Dimensional Self-Organizing Map

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.

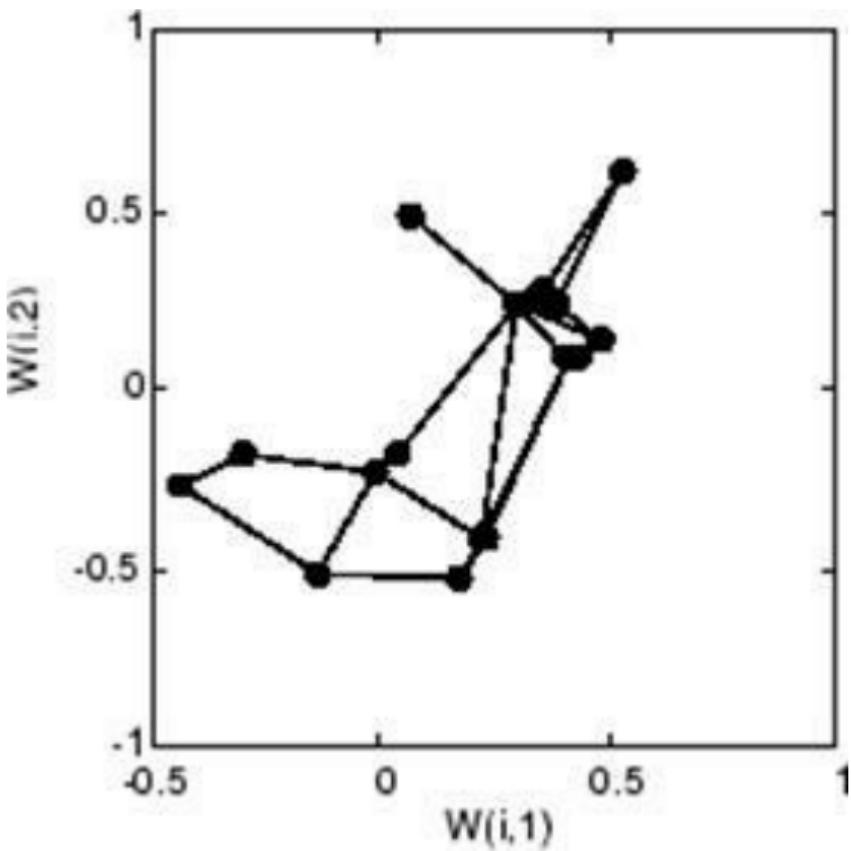


A 5-by-6 two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with

distances calculated according to the Manhattan distance neighborhood function [mandist](#).

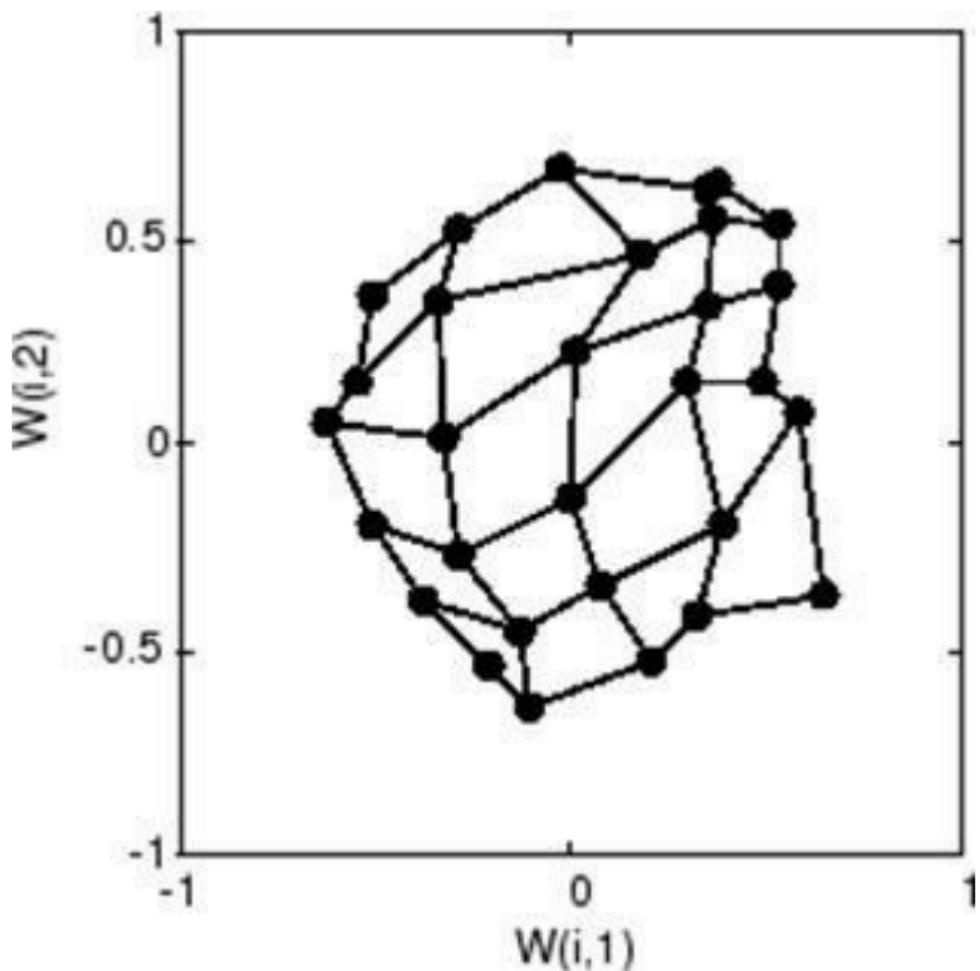
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

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.

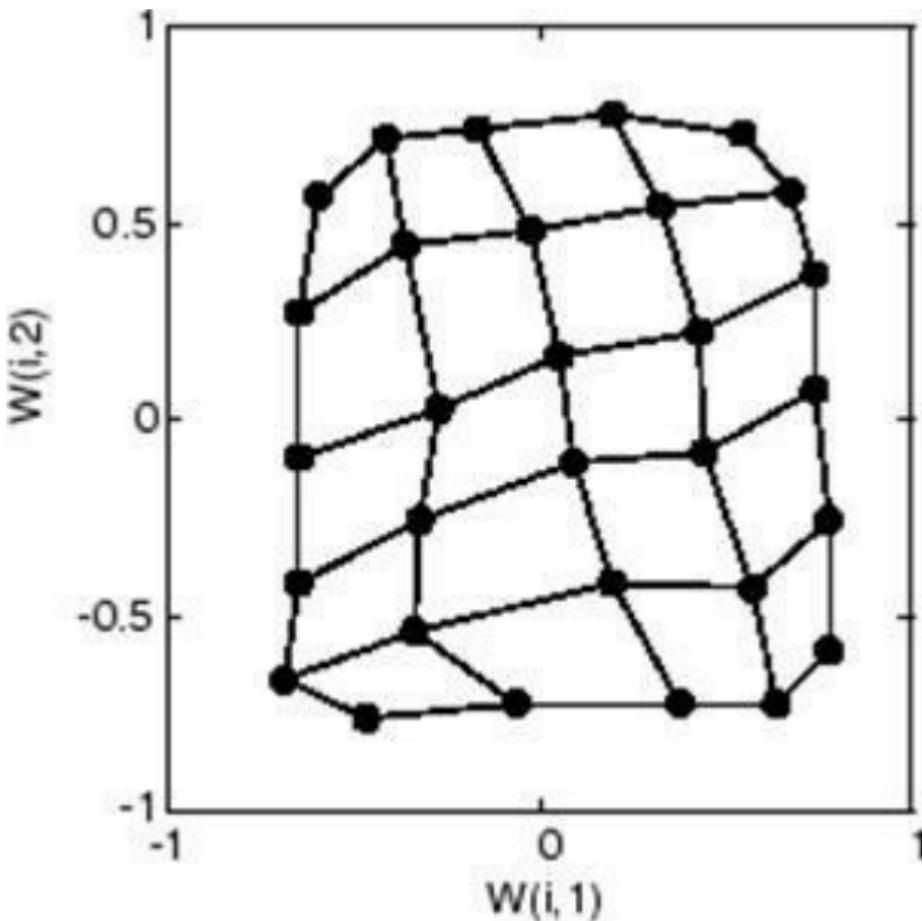
Here is the map after 120 cycles.



After 120 cycles, the map has begun to organize itself according to the topology

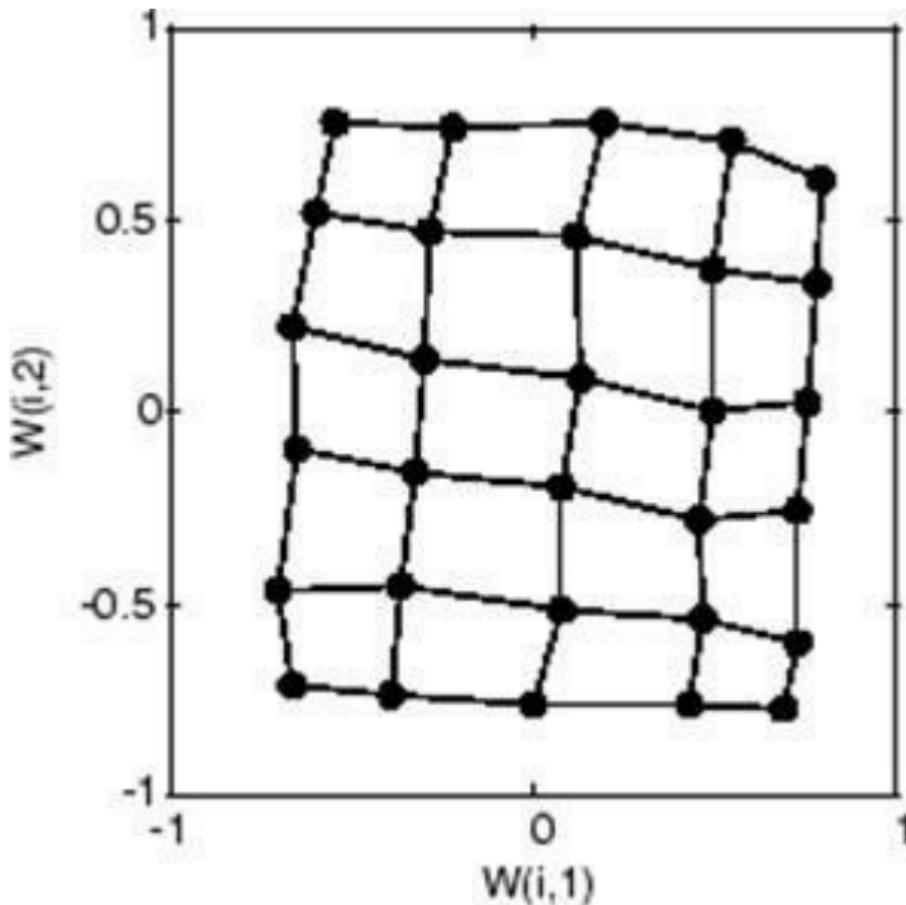
of the input space, which constrains input vectors.

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.

### 3.7.3 Training with the Batch Algorithm

The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

```
x = simplecluster_dataset
```

```
net = selforgmap([6 6]);
```

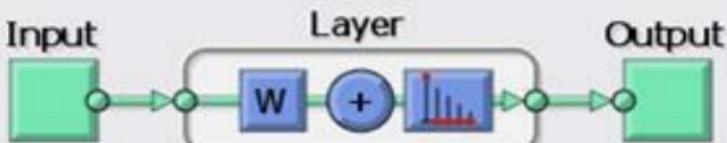
```
net = train(net,x);
```

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

following figure appears.

# Neural Network Training (nntraintool)

## Neural Network



## Algorithms

Training: Batch Unsupervised Weight/Bias Training (trainbuwb)

## Progress

Epoch: 0      200 Iterations      200

Time:      0:00:09

## Plots

SOM Topology (plotsomtop)

SOM Neighbor Connections (plotsomnc)

SOM Neighbor Distances (plotsomnd)

SOM Weight Planes (plotsomplanes)

SOM Sample Hits (plotsomhits)

SOM Weight Positions (plotsompos)

Plot Interval: 1 epochs

✓ Opening SOM Neighbor Distances Plot

Stop Training

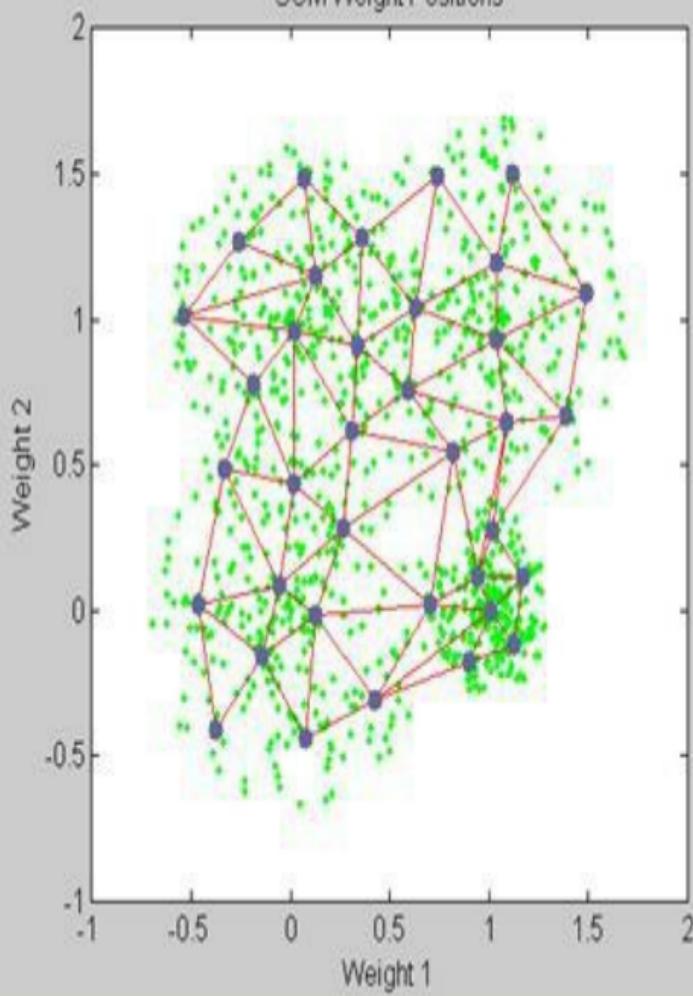
Cancel

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.

# SOM Weight Positions (plotsompos)



SOM Weight Positions



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

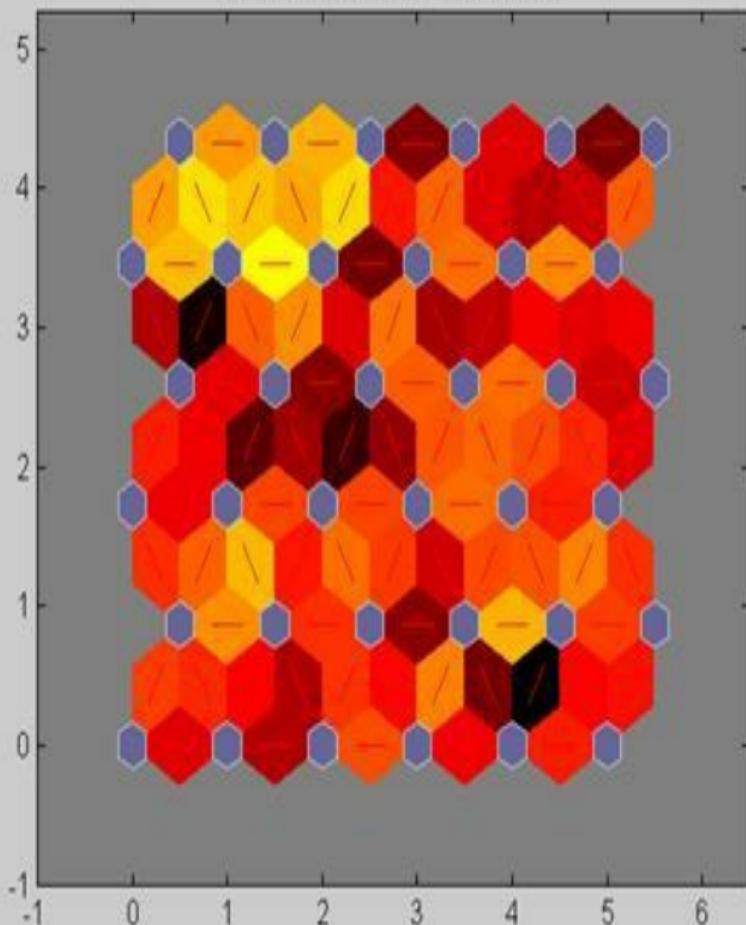
distances between neurons.

- 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

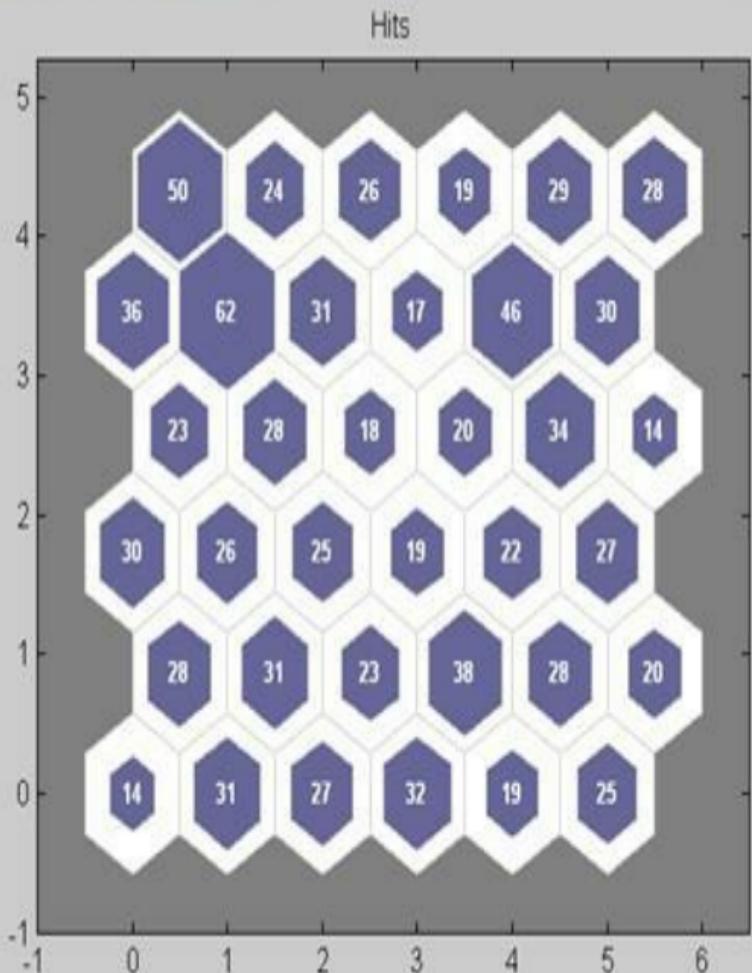
colors in the neighbor distance figure. Where weights in this small region connect to the larger region, the distances are larger, as indicated by the darker band in the neighbor distance figure. The segments in the lower-right region of the neighbor distance figure are darker than those in the upper 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 Weight Distances



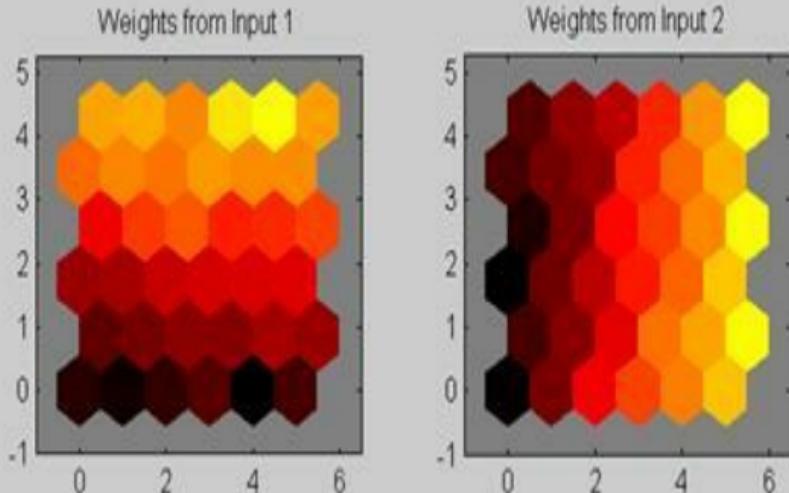
Another useful figure can tell you how

many data points are associated with each neuron. Click **SOM Sample Hits** to see the following figure. It is best if the data are fairly evenly distributed across the neurons. In this example, the data are concentrated a little more in the upper-left neurons, but overall the distribution is fairly even.



You can also visualize the weights

themselves using the weight plane figure. Click **SOM Weight Planes** in the training window to obtain the next figure. There is a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Lighter and darker colors represent larger and smaller weights, respectively.) If the connection patterns of two inputs are very similar, you can assume that the inputs were highly correlated. In this case, input 1 has connections that are very different than those of input 2.



You can also produce all of the previous figures from the command line. Try these plotting commands: [plotsomhits](#), [plotsomnc](#), [plotsompos](#), and [plotsomtop](#).



# **Chapter 4**

## **SELF-ORGANIZING MAPS. FUNCTIONS**

---

# 4.1 FUNCTIONS

<a href="#"><u>nnstart</u></a>	Neural network getting started GUI
<a href="#"><u>view</u></a>	View neural network
<a href="#"><u>selforgmap</u></a>	Self-organizing map
<a href="#"><u>train</u></a>	Train neural network
<a href="#"><u>plotsomhits</u></a>	Plot self-organizing map sample hits
<a href="#"><u>plotsomnc</u></a>	Plot self-organizing map neighbor code
<a href="#"><u>plotsomnd</u></a>	Plot self-organizing map neighbor distance
<a href="#"><u>plotsomplanes</u></a>	Plot self-organizing map weight planes
<a href="#"><u>plotsompos</u></a>	Plot self-organizing map weight positions
<a href="#"><u>plotsomtop</u></a>	Plot self-organizing map topology
<a href="#"><u>genFunction</u></a>	Generate MATLAB function for si

## 4.2 NNSTART

Neural network getting started GUI

### Syntax

Nnstart

### Description

nnstart opens a window with launch buttons for neural network fitting, pattern recognition, clustering and time series tools. It also provides links to lists of data sets, examples, and other useful information for getting started.

# nnstart

 Neural Network Start (nnstart)



## Welcome to Neural Network Start

Learn how to solve problems with neural networks.

[Getting Started Wizards](#) [More Information](#)

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 app (nftool)

Pattern recognition and classification.

 Pattern Recognition app (nprtool)

Clustering.

 Clustering app (nctool)

Dynamic Time series.

 Time Series app (ntstool)

## 4.3 VIEW

View neural network

### Syntax

`view(net)`

### Description

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

### Example. View Neural Network

This example shows how to view the

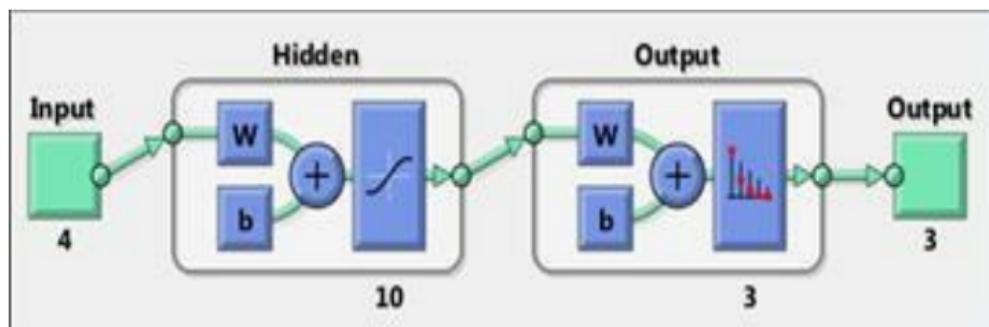
diagram of a pattern recognition network.

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

```
net = patternnet;
```

```
net = configure(net,x,t);
```

```
view(net)
```



## 4.4 SELFORGMAP

Self-organizing map

### Syntax

```
selforgmap(dimensions,coverSteps,ii)
```

### 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,initN`  
these arguments,

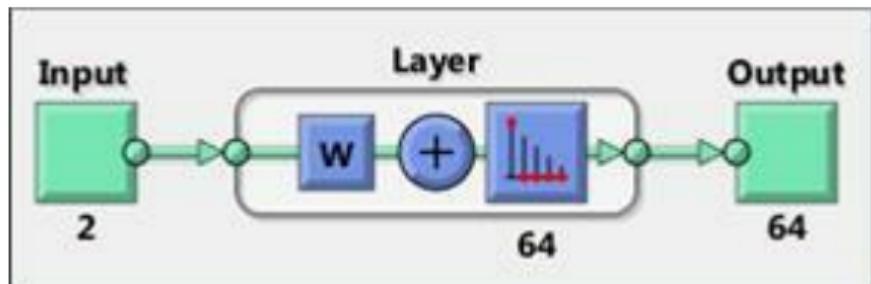
<code>dimensions</code>	Row vector of dimension sizes (default = [8 8])
<code>coverSteps</code>	Number of training steps for initial covering of the input data (default = 100)
<code>initNeighbor</code>	Initial neighborhood size (default = 3)
<code>topologyFcn</code>	Layer topology function (default =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

and returns a self-organizing map.

# Examples. Use Self-Organizing Map to Cluster Data

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

```
x = simplecluster_dataset;  
net = selforgmap([8 8]);  
net = train(net,x);  
view(net)  
y = net(x);  
classes = vec2ind(y);
```



# 4.5 TRAIN

Train neural network

## Syntax

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

[net,\_\_] = train(\_\_,'useParallel',\_\_)

[net,\_\_] = train(\_\_,'useGPU',\_\_)

[net,\_\_] =

train(\_\_,'showResources',\_\_)

[net,\_\_] =

train(Xcomposite,Tcomposite,\_\_)

[net,\_\_] = train(Xgpu,Tgpu,\_\_)

net =

train(\_\_,'CheckpointFile','path/name')

## Description

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

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

`net` Network

`X` Network inputs

`T` Network targets (default = zeros)

`Xi` Initial input delay conditions  
(default = zeros)

`Ai` Initial layer delay conditions  
(default = zeros)

`EW` Error weights

and returns

`net` Newly trained network

`tr` Training record (epoch and perf)

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

train arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

The matrix format is as follows:

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

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

X Ni-by-TS cell array      Each element  $X\{i,ts\}$  is an  $R_i$ -by-Q matrix.

T No-by-TS cell array      Each element  $T\{i,ts\}$  is a  $U_i$ -by-Q matrix.

$X_i$  Ni-by-ID cell array      Each element  $X_i\{i,k\}$  is an  $R_i$ -by-Q matrix.

$A_i$  Nl-by-LD cell array      Each element  $A_i\{i,k\}$  is an  $S_i$ -by-Q matrix.

EW No-by-TS cell array      Each element  $EW\{i,ts\}$  is a  $U_i$ -by-Q matrix

where

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

The columns of  $X_i$  and  $A_i$  are ordered from the oldest delay condition to the most recent:

$$X_i\{i,k\} = \text{Input } i \text{ at time ts} = k$$

- ID

$$A_i\{i,k\} = \text{Layer output } i \text{ at time } ts = k - LD$$

The error weights EW can also have a size of 1 in place of all or any of No, TS, Ui or Q. In that case, EW is automatically dimension extended to match the targets T. This allows for conveniently weighting the importance in any dimension (such as per sample) while having equal importance across another (such as time, with TS=1). If all dimensions are 1, for instance if EW = {1}, then all target values are treated with the same importance. That is the default value of EW.

The matrix format can be used if only

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

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

X	(sum of $R_i$ )-by-Q matrix
T	(sum of $U_i$ )-by-Q matrix
$X_i$	(sum of $R_i$ )-by-( $ID^*Q$ ) matrix
$A_i$	(sum of $S_i$ )-by-( $LD^*Q$ ) matrix
EW	(sum of $U_i$ )-by-Q matrix

As noted above, the error weights EW can be of the same dimensions as the targets T, or have

some dimensions set to 1. For instance if EW is 1-by-Q, then target samples will have different importances, but each element in a sample will have the same importance. If EW is (sum of  $U_i$ )-by-Q, then each output element has a different importance, with all samples treated with the same importance.

The training record TR is a structure whose fields depend on the network training function (net.NET.trainFcn). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for

training, validation and test sets

- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network  
(`best_perf`, `best_vperf`, `best_tperf`)

[net, \_\_] =

train(\_\_, 'useParallel', \_\_), [net, \_\_] =

`train(__,'useGPU',__)`, or `[net,__] = train(__,'showResources',__)` accepts optional name/value pair arguments to control how calculations are performed. Two of these options allow training to happen faster or on larger datasets using parallel workers or GPU devices if Parallel Computing Toolbox is available. These are the optional name/value pairs:

'useParallel','no'	Calculations occur on normal MATLAB thread. This is the default 'useParallel' setting.
'useParallel','yes'	Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.
'useGPU','no'	Calculations occur on the CPU. This is the default 'useGPU' setting.
	Calculations occur on the current <code>gpuDevice</code> if it is a supported GPU (See Parallel

'useGPU','yes'	Computing Toolbox for GPU requirements.) If the current gpuDevice is not supported, calculations remain on the CPU. If 'useParallel' is also 'yes' and a parallel pool is open, then each worker with a unique GPU uses that GPU, other workers run calculations on their respective CPU cores.
'useGPU','only'	If no parallel pool is open, then this setting is the same as 'yes'. If a parallel pool is open then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.
'showResources','no'	Do not display computing resources used at the command line. This is the default setting.
'showResources','yes'	Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker's computation mode is described, including workers in the pool that are not used.
'reduction',N	For most neural networks, the default CPU training computation mode is a compiled MEX algorithm. However, for large networks the calculations might occur with a MATLAB calculation mode. This can be confirmed using 'showResources'. If MATLAB is being used and memory is an issue, setting the reduction option to a value N greater than 1, reduces much of the temporary storage required

to train by a factor of N, in exchange for longer training times.

[net,\_] =  
train(Xcomposite,Tcomposite,\_) takes Composite data and returns Composite results. If Composite data is used, then 'useParallel' is automatically set to 'yes'.

[net,\_] = train(Xgpu,Tgpu,\_) takes gpuArray data and returns gpuArray results. If gpuArray data is used, then 'useGPU' is automatically set to 'yes'.

net =  
train(\_, 'CheckpointFile', 'path/name', 'Ch' saves intermediate values of the neural

network and training record during training to the specified file. This protects training results from power failures, computer lock ups, Ctrl+C, or any other event that halts the training process before train returns normally.

The value for 'CheckpointFile' can be set to a filename to save in the current working folder, to a file path in another folder, or to an empty string to disable checkpoint saves (the default value).

The optional parameter 'CheckpointDelay' limits how often saves happen. Limiting the frequency of checkpoints can improve efficiency by keeping the amount of time saving checkpoints low compared to the

time spent in calculations. It has a default value of 60, which means that checkpoint saves do not happen more than once per minute. Set the value of 'CheckpointDelay' to 0 if you want checkpoint saves to occur only once every epoch.

**Note** Any NaN values in the inputs X or the targets T, are treated as missing data. If a column of X or T contains at least one NaN, that column is not used for training, testing, or validation.

## Examples:

# Train and Plot Networks

Here input x and targets t define a simple function that you can plot:

```
x = [0 1 2 3 4 5 6 7 8];
```

```
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28  
0.66 0.99];
```

```
plot(x,t,'o')
```

Here feedforwardnet creates a two-layer feed-forward network. The network has one hidden layer with ten neurons.

```
net = feedforwardnet(10);
```

```
net = configure(net,x,t);
```

```
y1 = net(x)
```

```
plot(x,t,'o',x,y1,'x')
```

The network is trained and then resimulated.

```
net = train(net,x,t);
```

```
y2 = net(x)
```

```
plot(x,t,'o',x,y1,'x',x,y2,'*')
```

## Train NARX Time Series Network

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current  $x$  and the magnet's vertical position response  $t$ , then simulates the network. The function [prepares](#) prepares the data

before training and simulation. It creates the open-loop network's combined inputs  $x_o$ , which contains both the external input  $x$  and previous values of position  $t$ . It also prepares the delay states  $x_i$ .

```
[x,t] = maglev_dataset;  
net = narxnet(10);  
[xo,xi,~,to] = preparets(net,x,{},t);  
net = train(net,xo,to,xi);  
y = net(xo,xi)
```

This same system can also be simulated in closed-loop form.

```
netc = closeloop(net);  
view(netc)  
[xc,xi,ai,tc] = preparets(netc,x,{},t);
```

```
yc = netc(xc,xi,ai);
```

## Train a Network in Parallel on a Parallel Pool

Parallel Computing Toolbox™ allows Neural Network Toolbox™ to simulate and train networks faster and on larger datasets than can fit on one PC. Parallel training is currently supported for backpropagation training only, not for self-organizing maps.

Here training and simulation happens across parallel MATLAB workers.

```
parpool
```

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

```
net = feedforwardnet(10);
```

```
net =  
train(net,X,T,'useParallel','yes','showRes'  
Y = net(X);
```

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded as it is distributed then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

```
[X,T] = vinyl_dataset;  
Q = size(X,2);  
Xc = Composite;  
Tc = Composite;  
numWorkers = numel(Xc);  
ind = [0 ceil((1:4)*(Q/4))];  
for i=1:numWorkers
```

```
indi = (ind(i)+1):ind(i+1);
Xc{i} = X(:,indi);
Tc{i} = T(:,indi);
end
net = feedforwardnet;
net = configure(net,X,T);
net = train(net,Xc,Tc);
Yc = net(Xc);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when `train` is called, but when providing composite data this step must be done manually with non-Composite data.

# Train a Network on GPUs

Networks can be trained using the current GPU device, if it is supported by Parallel Computing Toolbox. GPU training is currently supported for backpropagation training only, not for self-organizing maps.

```
[X,T] = vinyl_dataset;  
net = feedforwardnet(10);  
net = train(net,X,T,'useGPU','yes');  
y = net(X);
```

To put the data on a GPU manually:

```
[X,T] = vinyl_dataset;  
Xgpu = gpuArray(X);  
Tgpu = gpuArray(T);
```

```
net = configure(net,X,T);  
net = train(net,Xgpu,Tgpu);  
Ygpu = net(Xgpu);  
Y = gather(Ygpu);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when `train` is called, but when providing `gpuArray` data this step must be done manually with non-`gpuArray` data.

To run in parallel, with workers each assigned to a different unique GPU, with extra workers running on CPU:

```
net =
```

```
train(net,X,T,'useParallel','yes','useGPU',  
y = net(X);
```

Using only workers with unique GPUs might result in higher speed, as CPU workers might not keep up.

```
net =  
train(net,X,T,'useParallel','yes','useGPU',  
Y = net(X);
```

## Train Network Using Checkpoint Saves

Here a network is trained with checkpoints saved at a rate no greater than once every two minutes.

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

```
net = fitnet([60 30]);  
net =  
train(net,x,t,'CheckpointFile','MyCheckpc
```

After a computer failure, the latest network can be recovered and used to continue training from the point of failure. The checkpoint file includes a structure variable `checkpoint`, which includes the network, training record, filename, time, and number.

```
[x,t] = vinyl_dataset;  
load MyCheckpoint  
net = checkpoint.net;  
net =  
train(net,x,t,'CheckpointFile','MyCheckpc
```

Another use for the checkpoint feature is

when you stop a parallel training session (started with the 'UseParallel' parameter) even though the Neural Network Training Tool is not available during parallel training. In this case, set a 'CheckpointFile', use Ctrl+C to stop training any time, then load your checkpoint file to get the network and training record.

## Algorithms

train calls the function indicated by net.trainFcn, using the training parameter values indicated by net.trainParam.

Typically one epoch of training is

defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of the function `net.trainFcn` occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly for each epoch from concurrent input vectors (or sequences). [competlayer](#) returns networks that use [trainru](#), a training function that does this.

## 4.6 PLOTSOMHITS

Plot self-organizing map sample hits

### Syntax

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

### Description

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

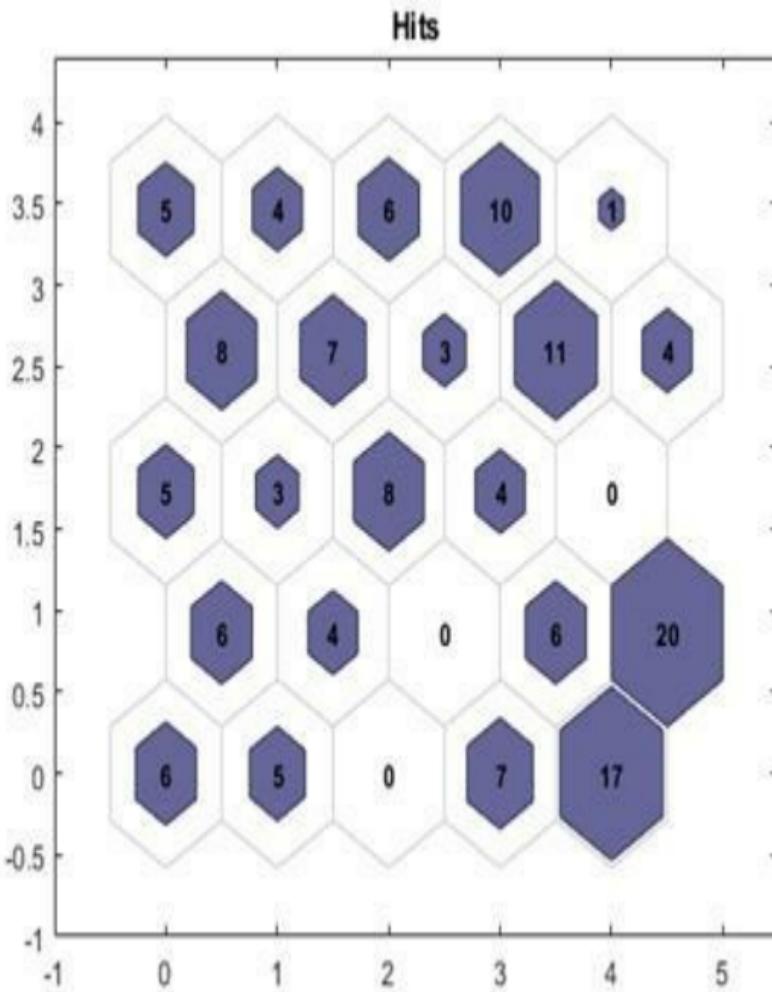
This plot supports SOM networks

with `hextop` and `gridtop` topologies,  
but not `tritop` or `randtop`.

## Examples. Plot SOM Sample Hits

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

plotsomhits (net, x)



## 4.7 PLOTSOMNC

Plot self-organizing map neighbor connections

### Syntax

```
plotsomnc(net)
```

### Description

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

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

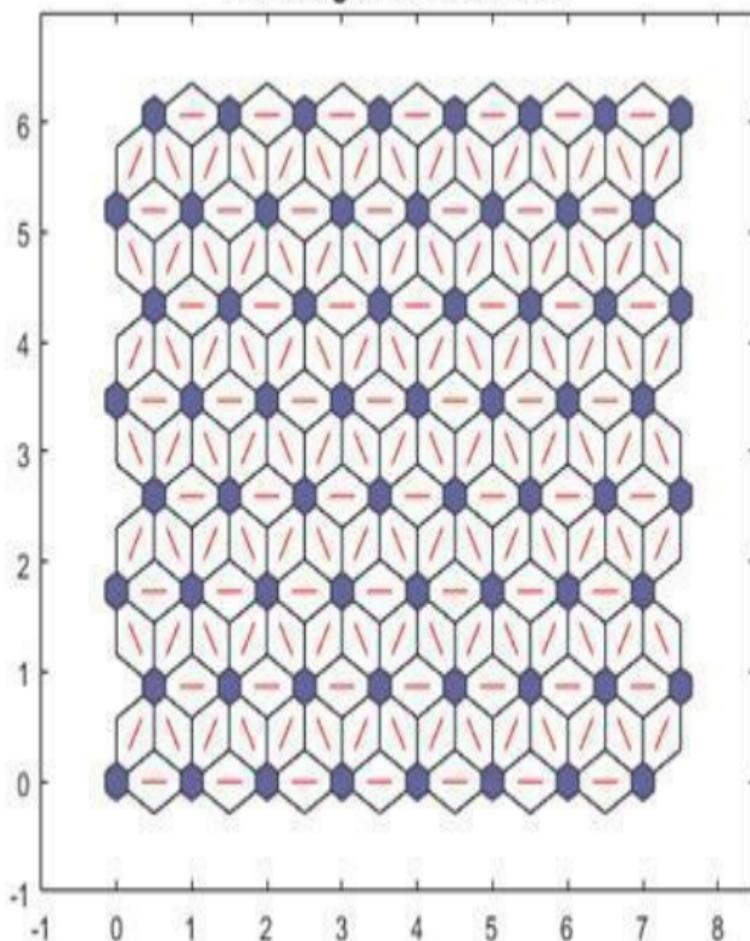
## Examples. Plot SOM Neighbor Connections

```
x = iris_dataset;  
  
net = selforgmap([8  
8]);
```

```
net = train(net,x);
```

```
plotsomnc(net)
```

### SOM Neighbor Connections



## 4.8 PLOTSOMND

Plot self-organizing map neighbor distances

### Syntax

```
plotsomnd(net)
```

### Description

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

colored from black to yellow to show how close each neuron's weight vector is to its neighbors.

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

## Examples. Plot SOM Neighbor Distances

```
x = iris_dataset;
```

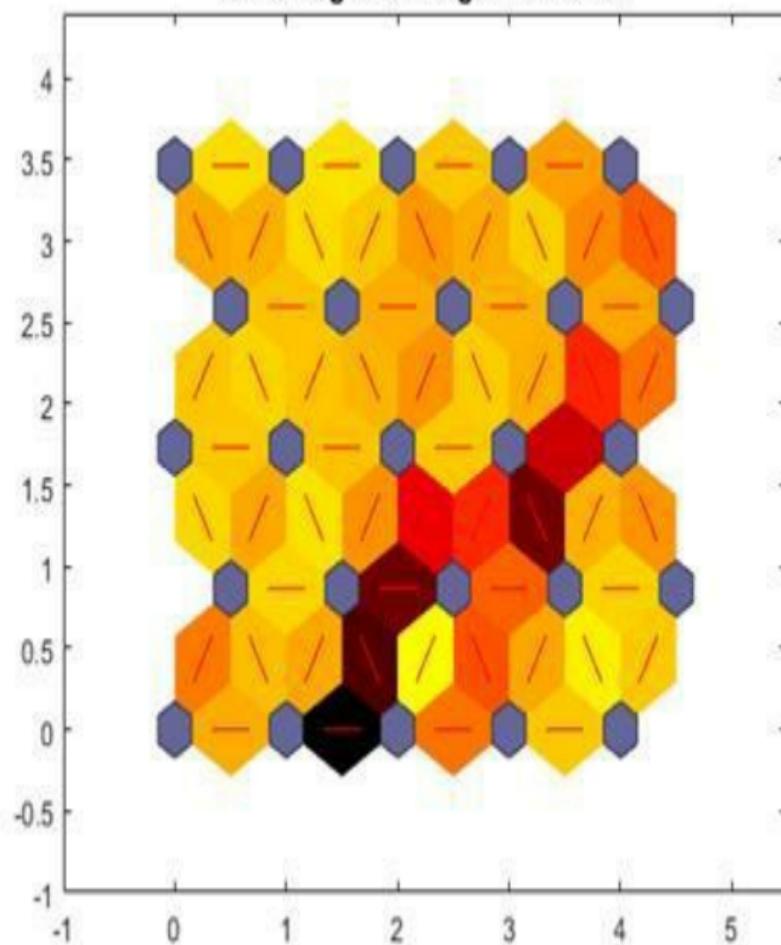
```
net = selforgmap([5
```

5] );

net = train(net,x);

plotsomnd(net)

### SOM Neighbor Weight Distances



## 4.9 PLOTSOMPLANES

Plot self-organizing map weight planes

### Syntax

```
plotsomplanes (net)
```

### Description

`plotsomplanes (net)` generates a set of subplots. Each  $i$ th subplot shows the weights from the  $i$ th input to the layer's neurons, with the most negative

connections shown as blue, zero connections as black, and the strongest positive connections as red.

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

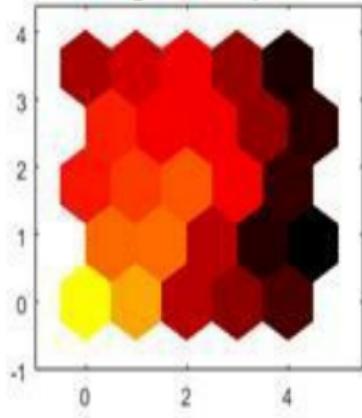
This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

This function can also be called with standardized plotting function arguments used by the function [train](#).

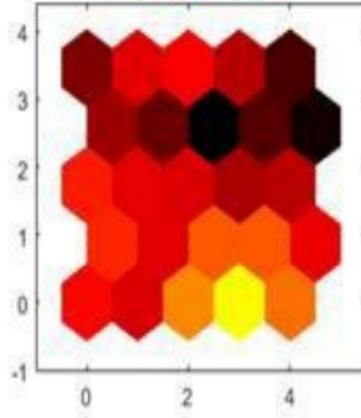
## Examples. Plot SOM Weight Planes

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

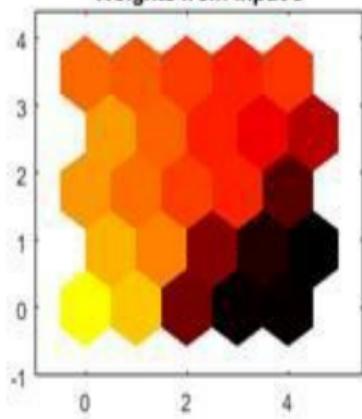
Weights from Input 1



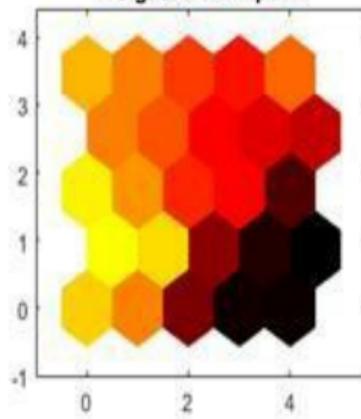
Weights from Input 2



Weights from Input 3



Weights from Input 4





## 4.10 PLOTSOMPOS

Plot self-organizing map weight positions

### Syntax

```
plotsompos(net)
```

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

### Description

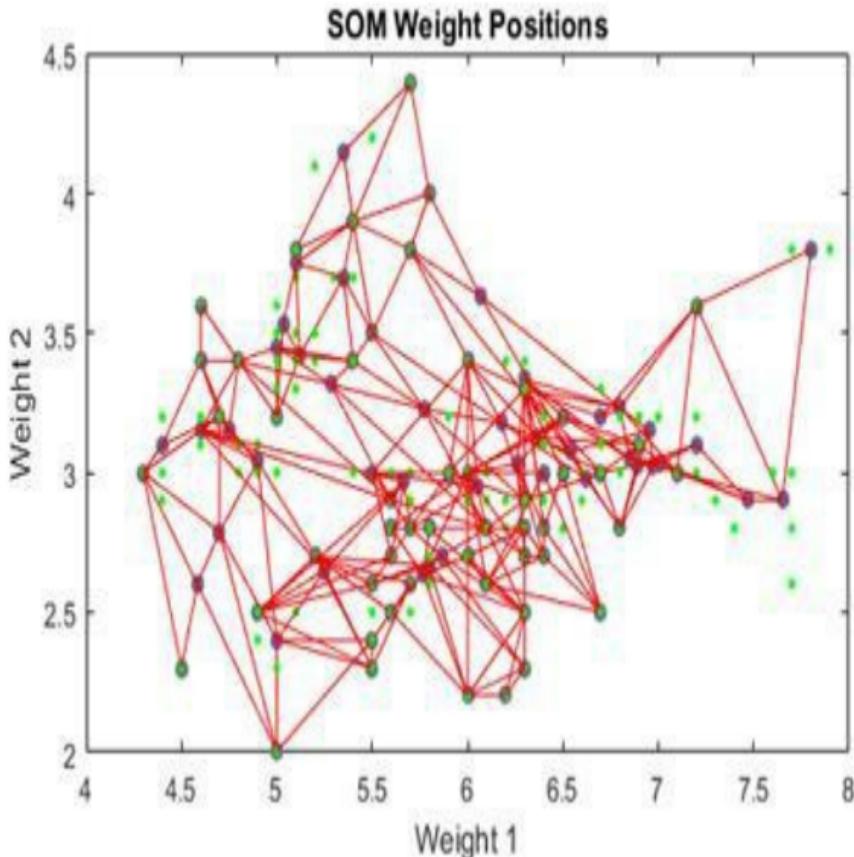
`plotsompos(net)` plots the input vectors as green dots and shows how the SOM classifies the input space by showing blue-gray dots for each neuron's weight vector and connecting neighboring neurons with red lines.

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

## Examples. Plot SOM Weight Positions

```
x = iris_dataset;  
net = selforgmap([10  
10]);  
net = train(net,x);
```

`plotsompos(net, x)`



## 4.11 PLOTSOMTOP

Plot self-organizing map topology

### Syntax

```
plotsomtop(net)
```

### Description

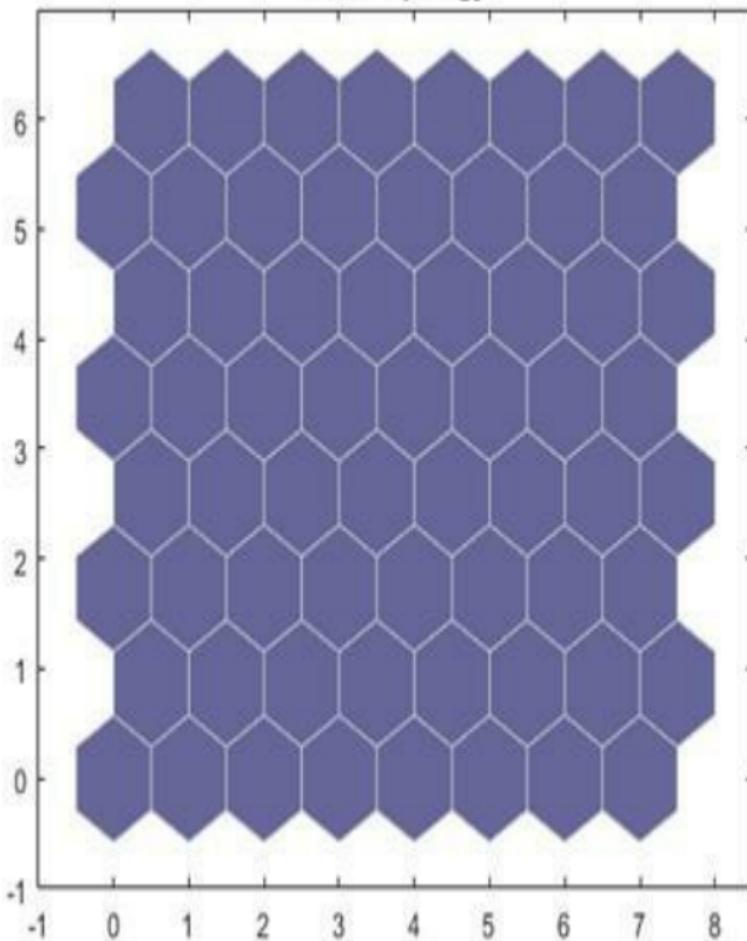
`plotsomtop(net)` plots the topology of a SOM layer.

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

### Examples. Plot SOM Topology

```
x = iris_dataset;  
  
net = selforgmap([8  
8]);  
  
plotsomtop(net)
```

SOM Topology



## 4.12 GENFUNCTION

Generate MATLAB function for simulating neural network

### Syntax

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

### Description

`genFunction(net,pathname)` generates a complete stand-alone MATLAB® function for simulating a neural network including all settings,

weight and bias values, module functions, and calculations in one file. The result is a standalone MATLAB function file. You can also use this function with MATLAB Compiler™ and MATLAB Coder™ tools.

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

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

## Examples:

### Create Functions from Static Neural Network

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

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

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

```
houseNet = train(houseNet,x,t);  
y = houseNet(x);
```

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

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

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

function, which is also tested.

```
genFunction(houseNet,'houseFcn','Matrix  
y3 = houseFcn(x);  
accuracy3 = max(abs(y-y3))
```

```
x1Type = coder.typeof(double(0),[13  
Inf]); % Coder type of input 1  
codegen houseFcn.m -config:mex -o  
houseCodeGen -args {x1Type}  
y4 = houseCodeGen(x);  
accuracy4 = max(abs(y-y4))
```

## Create Functions from Dynamic Neural Network

This example shows how to create a MATLAB function and a MEX-function

from a dynamic neural network.

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

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

```
maglevNet = narxnet(1:2,1:2,10);
```

```
[X,Xi,Ai,T] = prepares(maglevNet,x,  
{});
```

```
maglevNet =
```

```
train(maglevNet,X,T,Xi,Ai);
```

```
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next, generate and test a MATLAB function. Use the function to create a shared/dynamically linked library with mcc.

```
genFunction(maglevNet,'maglevFcn');
```

```
[y2,xf,af] = maglevFcn(X,Xi,Ai);  
accuracy2 = max(abs(cell2mat(y)-  
cell2mat(y2)))  
mcc -W lib:libMaglev -T link:lib  
maglevFcn
```

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

```
genFunction(maglevNet,'maglevFcn','Ma  
x1 = cell2mat(X(1,:)); % Convert each  
input to matrix  
x2 = cell2mat(X(2,:));  
xi1 = cell2mat(Xi(1,:)); % Convert each
```

input state to matrix

```
xi2 = cell2mat(Xi(2,:));
```

```
[y3,xfl,xf2] = maglevFcn(x1,x2,xi1,xi2);
```

```
accuracy3 = max(abs(cell2mat(y)-y3))
```

x1Type = coder.typeof(double(0),[1

Inf]); % Coder type of input 1

x2Type = coder.typeof(double(0),[1

Inf]); % Coder type of input 2

xi1Type = coder.typeof(double(0),[1 2]);

% Coder type of input 1 states

xi2Type = coder.typeof(double(0),[1 2]);

% Coder type of input 2 states

```
codegen maglevFcn.m -config:mex -o
```

```
maglevNetCodeGen -args {x1Type}
```

```
x2Type xi1Type xi2Type}
```

```
[y4,xfl,xf2] =
```

```
maglevNetCodeGen(x1,x2,xi1,xi2);
```

```
dynamic_codegen_accuracy =  
max(abs(cell2mat(y)-y4))
```

# **Chapter 5**

# **COMPETITIVE NEURAL NETWORKS**

---

## 5.1 CREATE A COMPETITIVE NEURAL NETWORK

You can create a competitive neural network with the function [competlayer](#). A simple example shows how this works.

Suppose you want to divide the following four two-element vectors into two classes.

```
p = [.1 .8 .1 .9; .2  
.9 .1 .8]
```

$p =$

0.1000

0.8000 0.1000

0.9000

0.2000

0.9000 0.1000

0.8000

There are two vectors near the origin

and two vectors near (1,1).

First, create a two-neuron competitive layer.:

```
net =  
competlayer(2);
```

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

The first time the network is trained, its weights will initialized to the centers of the input ranges with the function midpoint. You can check see these initial values using the number of neurons and the input data:

```
wts = midpoint(2,p)
```

```
wts =
```

```
0.5000 0.5000
```

```
0.5000 0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs.

The initial biases are computed

by [initcon](#), which gives

```
biases = initcon(2)
```

```
biases =
```

```
5.4366
```

```
5.4366
```

Recall that each neuron competes to respond to an input vector  $p$ . If the biases are all 0, the neuron whose

weight vector is closest to  $\mathbf{p}$  gets the highest net input and, therefore, wins the competition, and outputs 1. All other neurons output 0. You want to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

## 5.1.1 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.

## 5.1.2 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.

## 5.1.3 Training

Now train the network for 500 epochs.  
You can use either [train](#) or [adapt](#).

```
net.trainParam.epoch  
= 500;
```

```
net = train(net,p);
```

Note that [train](#) for competitive networks uses the training function [trainru](#). You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

```
ans =
```

```
trainru
```

For each epoch, all training vectors (or sequences) are each presented once in a different random order with the network and weight and bias values updated after each individual presentation.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

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

```
ac = vec2ind(a)
```

```
ac =
```

```
1 2
```

```
1 2
```

You see that the network is trained to classify the input vectors into two groups, those near the origin, class 1,

and those near (1,1), class 2.

It might be interesting to look at the final weights and biases.

net.IW{1,1}

ans =

0.1000 0.1500

0.8500 0.8500

net.b{1}

ans =

5.4367

5.4365

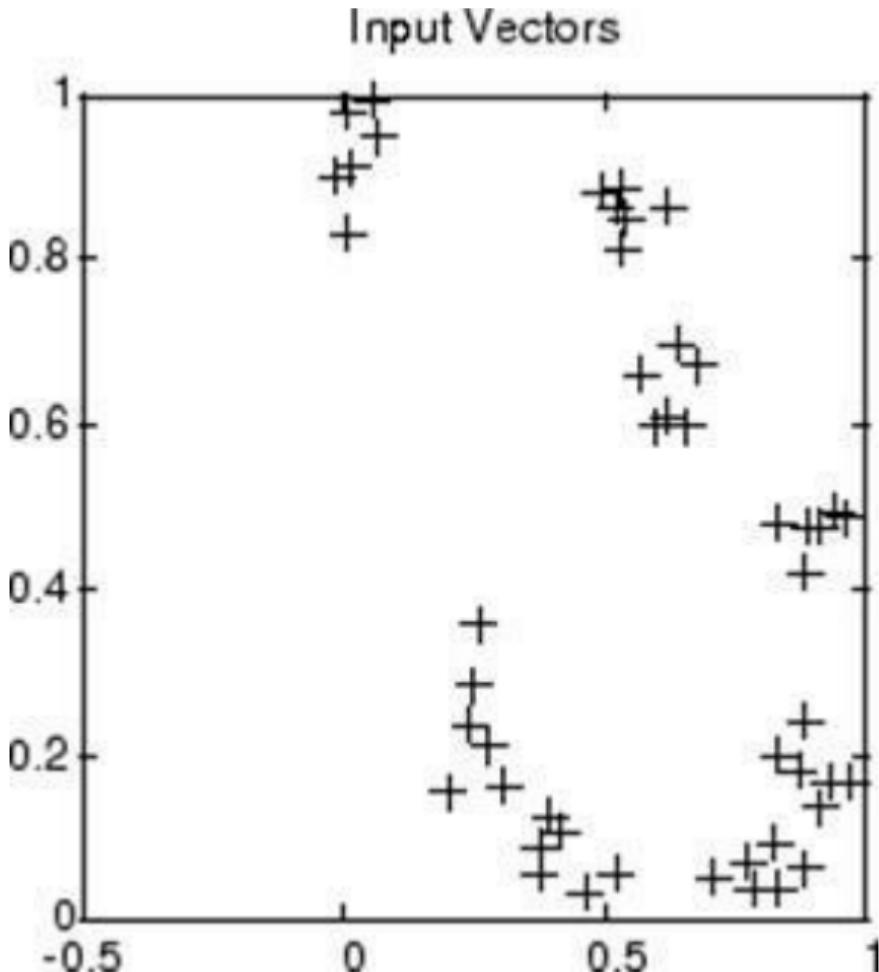
(You might get different answers when you run this problem, because a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to

the origin, while the vector formed from the second row of the weight matrix is close to the input vectors near (1,1). Thus, the network has been trained—just by exposing it to the inputs—to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

## 5.1.4 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.

## 5.2 COMPETITIVE LAYERS

Identify prototype vectors for clusters of examples using a simple neural network

## 5.2.1 Functions

<a href="#"><u>competlayer</u></a>	Competitive layer
<a href="#"><u>view</u></a>	View neural network
<a href="#"><u>train</u></a>	Train neural network
<a href="#"><u>trainru</u></a>	Unsupervised random order weight/bi:
<a href="#"><u>learnk</u></a>	Kohonen weight learning function
<a href="#"><u>learncon</u></a>	Conscience bias learning function
<a href="#"><u>genFunction</u></a>	Generate MATLAB function for simul

## 5.3 NNSTART

Neural network getting started GUI

### Syntax

`Nnstart`

### Description

`nnstart` opens a window with launch buttons for neural network fitting, pattern recognition, clustering and time series tools. It also provides links to lists of data sets, examples, and other useful information for getting started.

`nnstart`



## Welcome to Neural Network Start

Learn how to solve problems with neural networks.

Getting Started Wizards

More Information

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 app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(nprtool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

## 5.4 VIEW

View neural network

### Syntax

`view(net)`

### Description

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

### Example. View Neural Network

This example shows how to view the

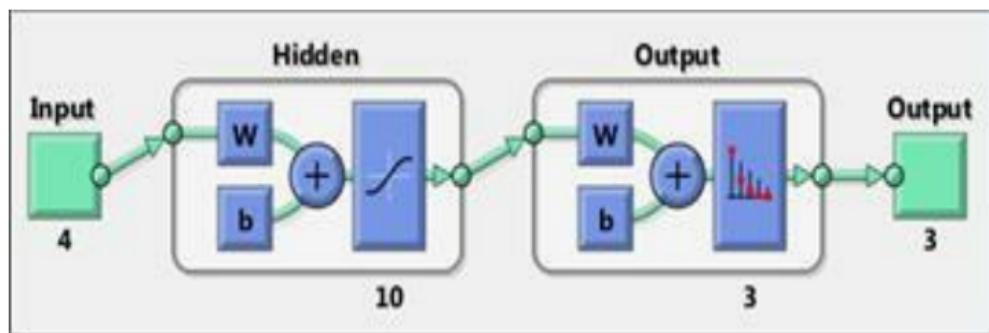
diagram of a pattern recognition network.

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

```
net = patternnet;
```

```
net = configure(net,x,t);
```

```
view(net)
```



## 5.5 SELFORGMAP

Self-organizing map

### Syntax

```
selforgmap(dimensions,coverSteps,ii)
```

### 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,initN`  
these arguments,

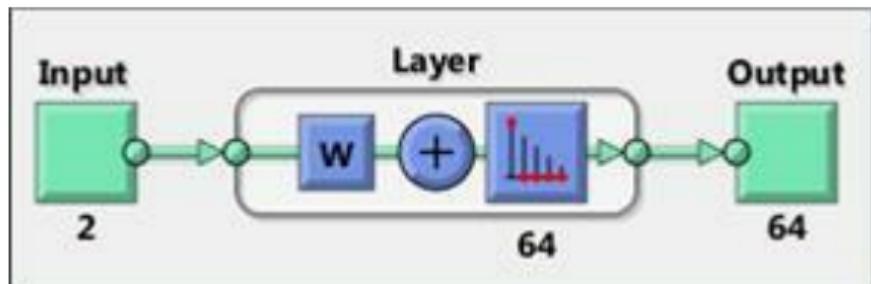
<code>dimensions</code>	Row vector of dimension sizes (default = [8 8])
<code>coverSteps</code>	Number of training steps for initial covering of the input data (default = 100)
<code>initNeighbor</code>	Initial neighborhood size (default = 3)
<code>topologyFcn</code>	Layer topology function (default =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

and returns a self-organizing map.

# Examples. Use Self-Organizing Map to Cluster Data

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

```
x = simplecluster_dataset;  
net = selforgmap([8 8]);  
net = train(net,x);  
view(net)  
y = net(x);  
classes = vec2ind(y);
```



# 5.6 TRAIN

Train neural network

## Syntax

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

[net,\_\_] = train(\_\_,'useParallel',\_\_)

[net,\_\_] = train(\_\_,'useGPU',\_\_)

[net,\_\_] =

train(\_\_,'showResources',\_\_)

[net,\_\_] =

train(Xcomposite,Tcomposite,\_\_)

[net,\_\_] = train(Xgpu,Tgpu,\_\_)

net =

train(\_\_,'CheckpointFile','path/name')

## Description

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

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

`net` Network

`X` Network inputs

`T` Network targets (default = zeros)

`Xi` Initial input delay conditions  
(default = zeros)

`Ai` Initial layer delay conditions  
(default = zeros)

`EW` Error weights

and returns

`net` Newly trained network

`tr` Training record (epoch and perf)

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

train arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

The matrix format is as follows:

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

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

X Ni-by-TS cell array      Each element  $X\{i,ts\}$  is an  $R_i$ -by-Q matrix.

T No-by-TS cell array      Each element  $T\{i,ts\}$  is a  $U_i$ -by-Q matrix.

$X_i$  Ni-by-ID cell array      Each element  $X_i\{i,k\}$  is an  $R_i$ -by-Q matrix.

$A_i$  Nl-by-LD cell array      Each element  $A_i\{i,k\}$  is an  $S_i$ -by-Q matrix.

EW No-by-TS cell array      Each element  $EW\{i,ts\}$  is a  $U_i$ -by-Q matrix

where

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

The columns of  $X_i$  and  $A_i$  are ordered from the oldest delay condition to the most recent:

$$X_i\{i,k\} = \text{Input } i \text{ at time ts} = k$$

- ID

$$A_i\{i,k\} = \text{Layer output } i \text{ at time } ts = k - LD$$

The error weights EW can also have a size of 1 in place of all or any of No, TS, Ui or Q. In that case, EW is automatically dimension extended to match the targets T. This allows for conveniently weighting the importance in any dimension (such as per sample) while having equal importance across another (such as time, with TS=1). If all dimensions are 1, for instance if EW = {1}, then all target values are treated with the same importance. That is the default value of EW.

The matrix format can be used if only

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

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

X	(sum of $R_i$ )-by-Q matrix
T	(sum of $U_i$ )-by-Q matrix
$X_i$	(sum of $R_i$ )-by-( $ID^*Q$ ) matrix
$A_i$	(sum of $S_i$ )-by-( $LD^*Q$ ) matrix
EW	(sum of $U_i$ )-by-Q matrix

As noted above, the error weights EW can be of the same dimensions as the targets T, or have

some dimensions set to 1. For instance if EW is 1-by-Q, then target samples will have different importances, but each element in a sample will have the same importance. If EW is (sum of  $U_i$ )-by-Q, then each output element has a different importance, with all samples treated with the same importance.

The training record TR is a structure whose fields depend on the network training function (net.NET.trainFcn). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for

training, validation and test sets

- Data division masks for training validation and test sets
- Number of epochs (`num_epochs`) and the best epoch (`best_epoch`).
- A list of training state names (`states`).
- Fields for each state name recording its value throughout training
- Performances of the best network  
(`best_perf`, `best_vperf`, `best_tperf`)

[net, \_\_] =

train(\_\_, 'useParallel', \_\_), [net, \_\_] =

`train(__,'useGPU',__)`, or `[net,__] = train(__,'showResources',__)` accepts optional name/value pair arguments to control how calculations are performed. Two of these options allow training to happen faster or on larger datasets using parallel workers or GPU devices if Parallel Computing Toolbox is available. These are the optional name/value pairs:

'useParallel','no'	Calculations occur on normal MATLAB thread. This is the default 'useParallel' setting.
'useParallel','yes'	Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.
'useGPU','no'	Calculations occur on the CPU. This is the default 'useGPU' setting.
	Calculations occur on the current <code>gpuDevice</code> if it is a supported GPU (See Parallel

'useGPU','yes'	Computing Toolbox for GPU requirements.) If the current gpuDevice is not supported, calculations remain on the CPU. If 'useParallel' is also 'yes' and a parallel pool is open, then each worker with a unique GPU uses that GPU, other workers run calculations on their respective CPU cores.
'useGPU','only'	If no parallel pool is open, then this setting is the same as 'yes'. If a parallel pool is open then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.
'showResources','no'	Do not display computing resources used at the command line. This is the default setting.
'showResources','yes'	Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker's computation mode is described, including workers in the pool that are not used.
'reduction',N	For most neural networks, the default CPU training computation mode is a compiled MEX algorithm. However, for large networks the calculations might occur with a MATLAB calculation mode. This can be confirmed using 'showResources'. If MATLAB is being used and memory is an issue, setting the reduction option to a value N greater than 1, reduces much of the temporary storage required

to train by a factor of N, in exchange for longer training times.

[net,\_] =  
train(Xcomposite,Tcomposite,\_) takes Composite data and returns Composite results. If Composite data is used, then 'useParallel' is automatically set to 'yes'.

[net,\_] = train(Xgpu,Tgpu,\_) takes gpuArray data and returns gpuArray results. If gpuArray data is used, then 'useGPU' is automatically set to 'yes'.

net =  
train(\_, 'CheckpointFile', 'path/name', 'Ch' saves intermediate values of the neural

network and training record during training to the specified file. This protects training results from power failures, computer lock ups, Ctrl+C, or any other event that halts the training process before train returns normally.

The value for 'CheckpointFile' can be set to a filename to save in the current working folder, to a file path in another folder, or to an empty string to disable checkpoint saves (the default value).

The optional parameter 'CheckpointDelay' limits how often saves happen. Limiting the frequency of checkpoints can improve efficiency by keeping the amount of time saving checkpoints low compared to the

time spent in calculations. It has a default value of 60, which means that checkpoint saves do not happen more than once per minute. Set the value of 'CheckpointDelay' to 0 if you want checkpoint saves to occur only once every epoch.

**Note** Any NaN values in the inputs X or the targets T, are treated as missing data. If a column of X or T contains at least one NaN, that column is not used for training, testing, or validation.

## Examples:

# Train and Plot Networks

Here input x and targets t define a simple function that you can plot:

```
x = [0 1 2 3 4 5 6 7 8];
```

```
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28  
0.66 0.99];
```

```
plot(x,t,'o')
```

Here feedforwardnet creates a two-layer feed-forward network. The network has one hidden layer with ten neurons.

```
net = feedforwardnet(10);
```

```
net = configure(net,x,t);
```

```
y1 = net(x)
```

```
plot(x,t,'o',x,y1,'x')
```

The network is trained and then resimulated.

```
net = train(net,x,t);
```

```
y2 = net(x)
```

```
plot(x,t,'o',x,y1,'x',x,y2,'*')
```

## Train NARX Time Series Network

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current  $x$  and the magnet's vertical position response  $t$ , then simulates the network. The function [prepares](#) prepares the data

before training and simulation. It creates the open-loop network's combined inputs  $x_o$ , which contains both the external input  $x$  and previous values of position  $t$ . It also prepares the delay states  $x_i$ .

```
[x,t] = maglev_dataset;  
net = narxnet(10);  
[xo,xi,~,to] = preparets(net,x,{},t);  
net = train(net,xo,to,xi);  
y = net(xo,xi)
```

This same system can also be simulated in closed-loop form.

```
netc = closeloop(net);  
view(netc)  
[xc,xi,ai,tc] = preparets(netc,x,{},t);
```

```
yc = netc(xc,xi,ai);
```

## Train a Network in Parallel on a Parallel Pool

Parallel Computing Toolbox™ allows Neural Network Toolbox™ to simulate and train networks faster and on larger datasets than can fit on one PC. Parallel training is currently supported for backpropagation training only, not for self-organizing maps.

Here training and simulation happens across parallel MATLAB workers.

```
parpool
```

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

```
net = feedforwardnet(10);
```

```
net =  
train(net,X,T,'useParallel','yes','showRes'  
Y = net(X);
```

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded as it is distributed then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

```
[X,T] = vinyl_dataset;  
Q = size(X,2);  
Xc = Composite;  
Tc = Composite;  
numWorkers = numel(Xc);  
ind = [0 ceil((1:4)*(Q/4))];  
for i=1:numWorkers
```

```
indi = (ind(i)+1):ind(i+1);
Xc{i} = X(:,indi);
Tc{i} = T(:,indi);
end
net = feedforwardnet;
net = configure(net,X,T);
net = train(net,Xc,Tc);
Yc = net(Xc);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when `train` is called, but when providing composite data this step must be done manually with non-Composite data.

# Train a Network on GPUs

Networks can be trained using the current GPU device, if it is supported by Parallel Computing Toolbox. GPU training is currently supported for backpropagation training only, not for self-organizing maps.

```
[X,T] = vinyl_dataset;  
net = feedforwardnet(10);  
net = train(net,X,T,'useGPU','yes');  
y = net(X);
```

To put the data on a GPU manually:

```
[X,T] = vinyl_dataset;  
Xgpu = gpuArray(X);  
Tgpu = gpuArray(T);
```

```
net = configure(net,X,T);  
net = train(net,Xgpu,Tgpu);  
Ygpu = net(Xgpu);  
Y = gather(Ygpu);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when `train` is called, but when providing `gpuArray` data this step must be done manually with non-`gpuArray` data.

To run in parallel, with workers each assigned to a different unique GPU, with extra workers running on CPU:

```
net =
```

```
train(net,X,T,'useParallel','yes','useGPU',  
y = net(X);
```

Using only workers with unique GPUs might result in higher speed, as CPU workers might not keep up.

```
net =
```

```
train(net,X,T,'useParallel','yes','useGPU',  
Y = net(X);
```

## Train Network Using Checkpoint Saves

Here a network is trained with checkpoints saved at a rate no greater than once every two minutes.

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

```
net = fitnet([60 30]);  
net =  
train(net,x,t,'CheckpointFile','MyCheckpc
```

After a computer failure, the latest network can be recovered and used to continue training from the point of failure. The checkpoint file includes a structure variable `checkpoint`, which includes the network, training record, filename, time, and number.

```
[x,t] = vinyl_dataset;  
load MyCheckpoint  
net = checkpoint.net;  
net =  
train(net,x,t,'CheckpointFile','MyCheckpc
```

Another use for the checkpoint feature is

when you stop a parallel training session (started with the 'UseParallel' parameter) even though the Neural Network Training Tool is not available during parallel training. In this case, set a 'CheckpointFile', use Ctrl+C to stop training any time, then load your checkpoint file to get the network and training record.

## Algorithms

train calls the function indicated by net.trainFcn, using the training parameter values indicated by net.trainParam.

Typically one epoch of training is

defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of the function `net.trainFcn` occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly for each epoch from concurrent input vectors (or sequences). [competlayer](#) returns networks that use [trainru](#), a training function that does this.

## 5.7 PLOTSOMHITS

Plot self-organizing map sample hits

### Syntax

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

### Description

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

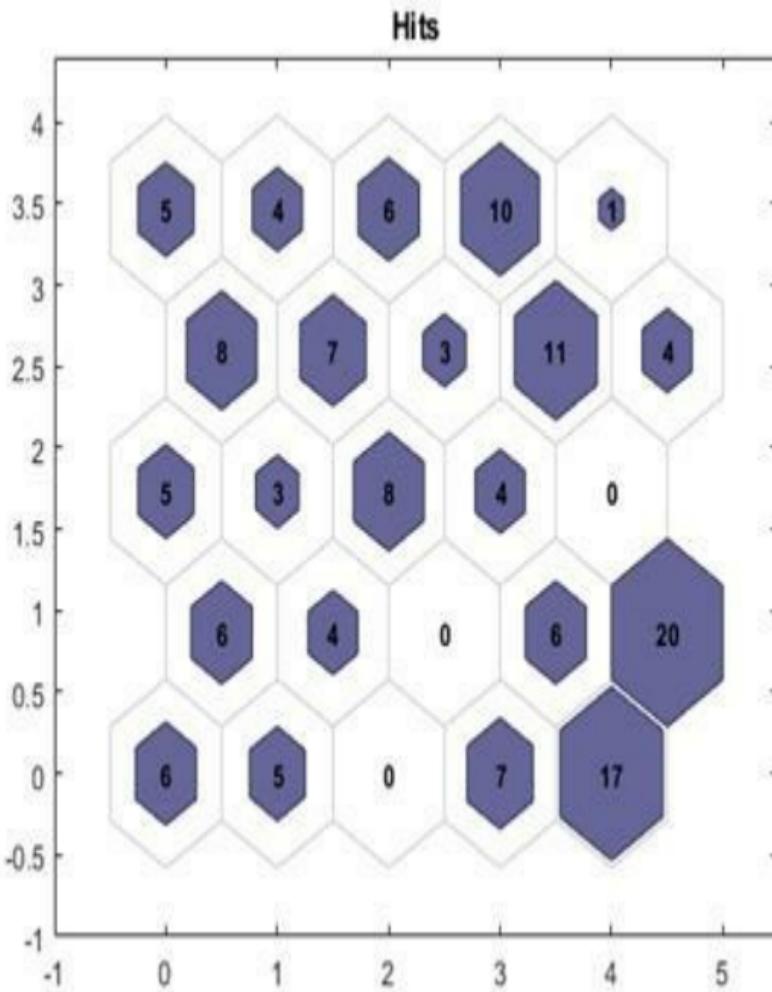
This plot supports SOM networks

with `hextop` and `gridtop` topologies,  
but not `tritop` or `randtop`.

## Examples. Plot SOM Sample Hits

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

plotsomhits (net, x)



## **5.8 PLOTSOMNC**

Plot self-organizing map neighbor connections

### **Syntax**

```
plotsomnc(net)
```

### **Description**

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

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

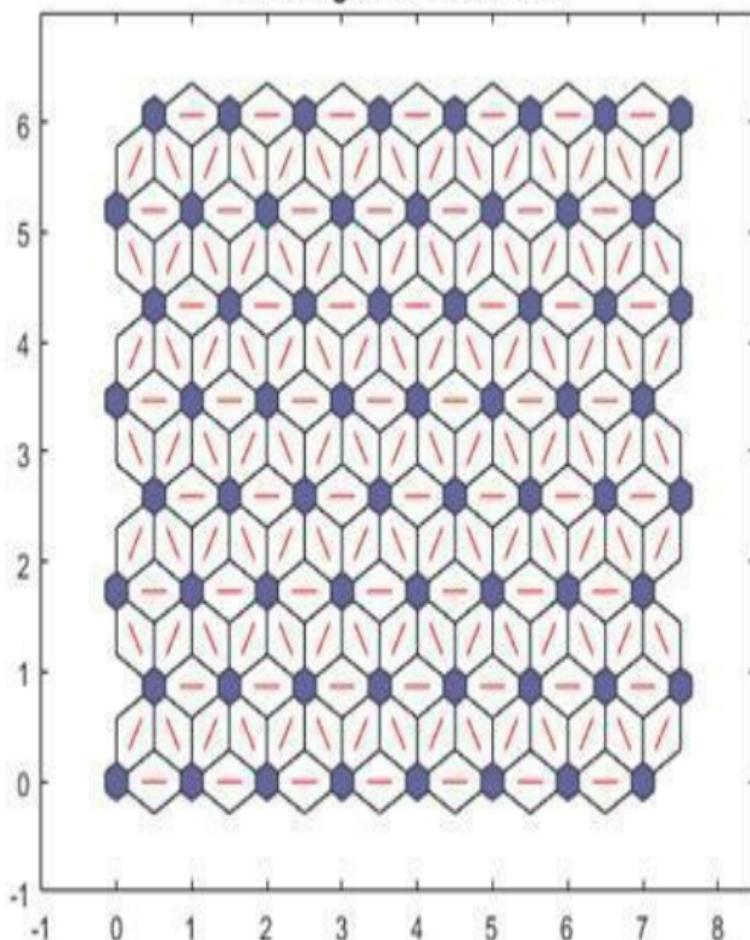
## Examples. Plot SOM Neighbor Connections

```
x = iris_dataset;  
  
net = selforgmap([8  
8]);
```

```
net = train(net,x);
```

```
plotsomnc(net)
```

### SOM Neighbor Connections



## 5.9 PLOTSOMND

Plot self-organizing map neighbor distances

### Syntax

```
plotsomnd(net)
```

### Description

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

colored from black to yellow to show how close each neuron's weight vector is to its neighbors.

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

## Examples. Plot SOM Neighbor Distances

```
x = iris_dataset;
```

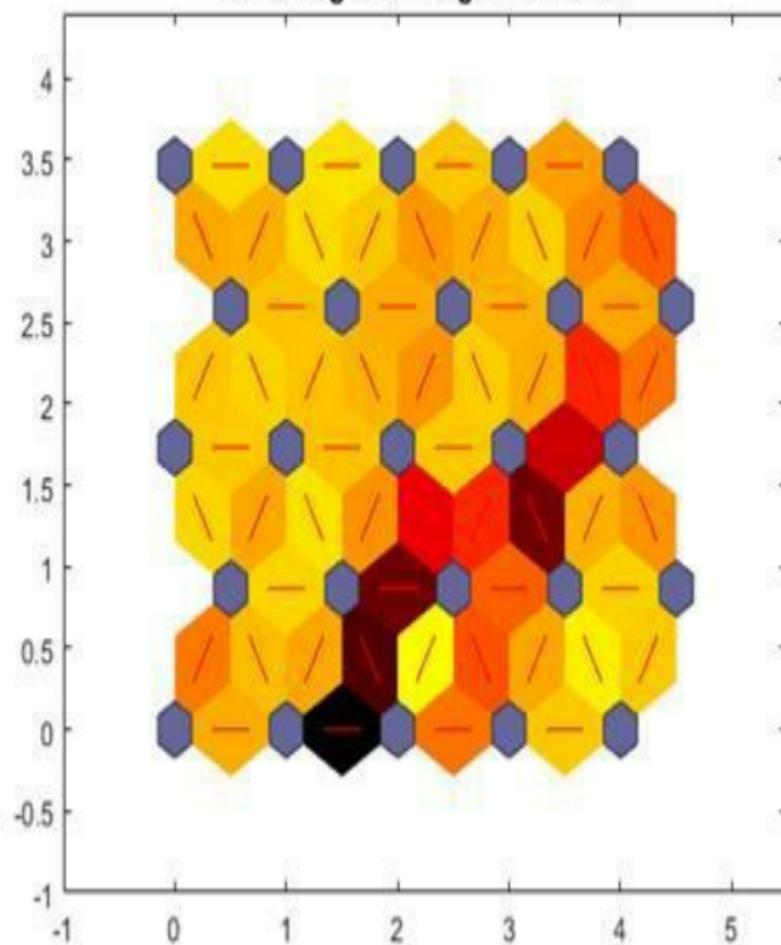
```
net = selforgmap([5
```

5] );

net = train(net,x);

plotsomnd(net)

### SOM Neighbor Weight Distances



# 5.10 PLOTSOMPLANES

Plot self-organizing map weight planes

## Syntax

```
plotsomplanes (net)
```

## Description

`plotsomplanes (net)` generates a set of subplots. Each  $i$ th subplot shows the weights from the  $i$ th input to the layer's neurons, with the most negative

connections shown as blue, zero connections as black, and the strongest positive connections as red.

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

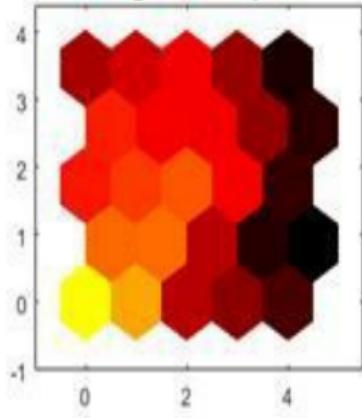
This plot supports SOM networks with `hextop` and `gridtop` topologies, but not `tritop` or `randtop`.

This function can also be called with standardized plotting function arguments used by the function [train](#).

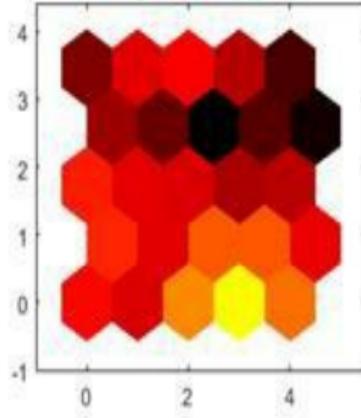
## Examples. Plot SOM Weight Planes

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

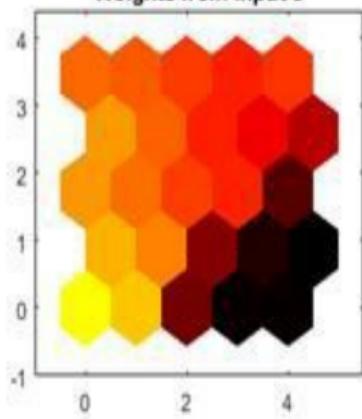
Weights from Input 1



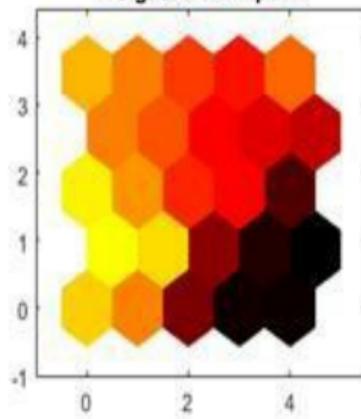
Weights from Input 2



Weights from Input 3



Weights from Input 4





## 5.11 PLOTSOMPOS

Plot self-organizing map weight positions

### Syntax

```
plotsompos(net)
```

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

### Description

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

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

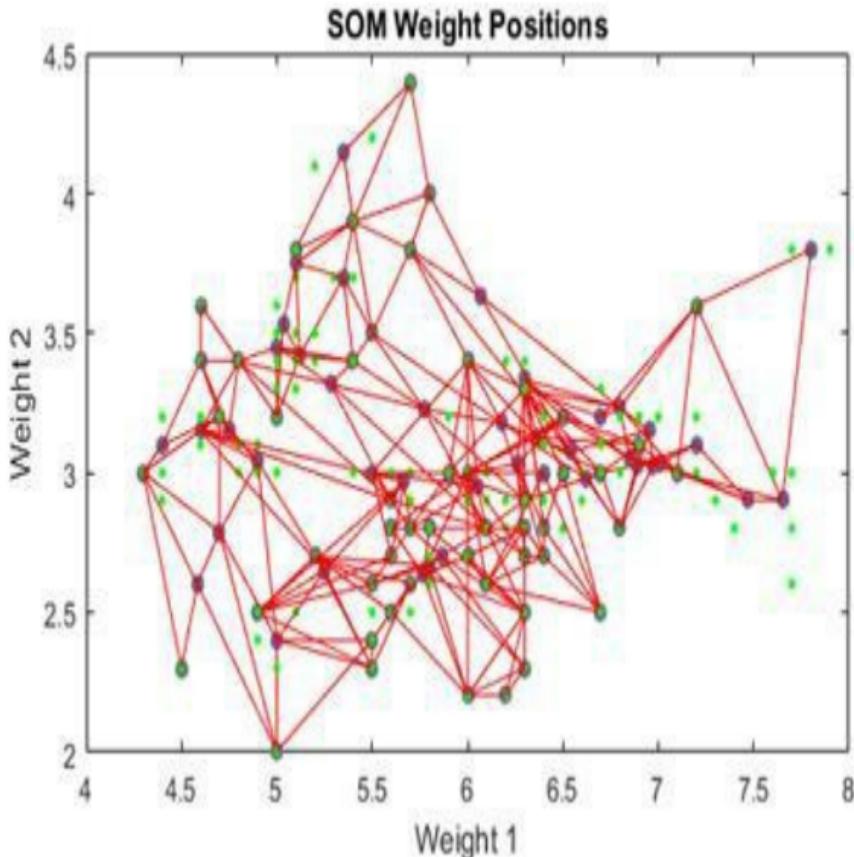
## Examples. Plot SOM Weight Positions

```
x = iris_dataset;
```

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

```
net = train(net, x);
```

`plotsompos(net, x)`



# 5.12 PLOTSOMTOP

Plot self-organizing map topology

## Syntax

```
plotsomtop(net)
```

## Description

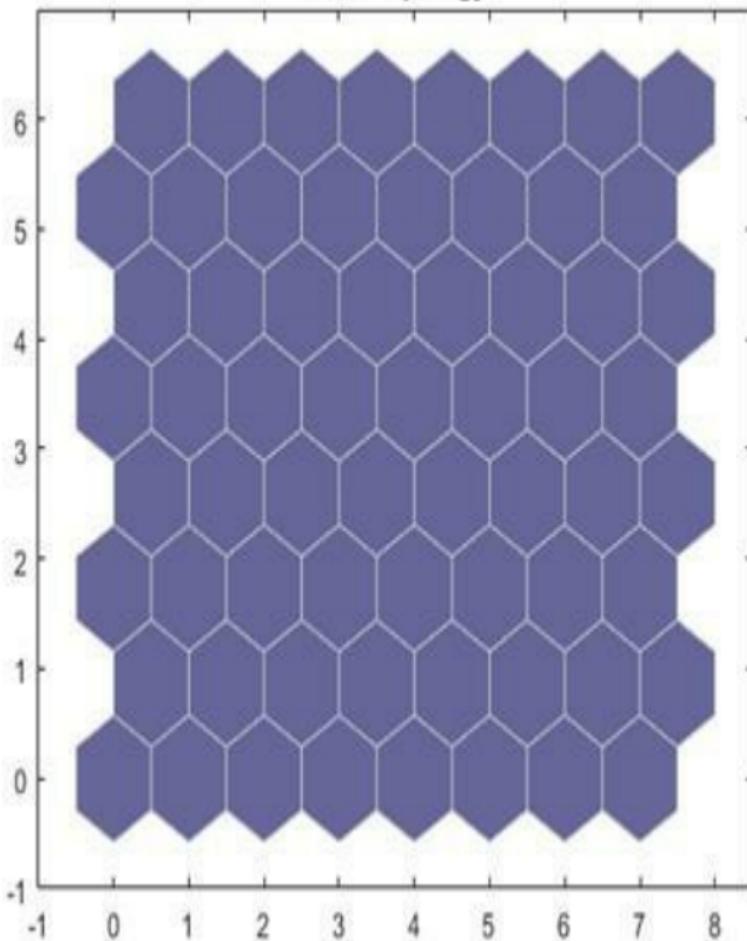
`plotsomtop(net)` plots the topology of a SOM layer.

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

## Examples. Plot SOM Topology

```
x = iris_dataset;  
  
net = selforgmap([8  
8]);  
  
plotsomtop(net)
```

SOM Topology



# 5.13 GENFUNCTION

Generate MATLAB function for simulating neural network

## Syntax

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

## Description

`genFunction(net,pathname)` generates a complete stand-alone MATLAB® function for simulating a neural network including all settings,

weight and bias values, module functions, and calculations in one file. The result is a standalone MATLAB function file. You can also use this function with MATLAB Compiler™ and MATLAB Coder™ tools.

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

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

## Examples

### Create Functions from Static Neural Network

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

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

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

```
houseNet = train(houseNet,x,t);  
y = houseNet(x);
```

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

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

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

function, which is also tested.

```
genFunction(houseNet,'houseFcn','Matrix  
y3 = houseFcn(x);  
accuracy3 = max(abs(y-y3))
```

```
x1Type = coder.typeof(double(0),[13  
Inf]); % Coder type of input 1  
codegen houseFcn.m -config:mex -o  
houseCodeGen -args {x1Type}  
y4 = houseCodeGen(x);  
accuracy4 = max(abs(y-y4))
```

## Create Functions from Dynamic Neural Network

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

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

```
[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(maglevNet,x,
{},t);
maglevNet =
train(maglevNet,X,T,Xi,Ai);
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next, generate and test a MATLAB function. Use the function to create a shared/dynamically linked library with mcc.

```
genFunction(maglevNet,'maglevFcn');
[y2,xf,af] = maglevFcn(X,Xi,Ai);
accuracy2 = max(abs(cell2mat(y)-
```

```
cell2mat(y2)))
```

```
mcc -W lib:libMaglev -T link:lib  
maglevFcn
```

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

```
genFunction(maglevNet,'maglevFcn','Ma1  
x1 = cell2mat(X(1,:)); % Convert each  
input to matrix  
x2 = cell2mat(X(2,:));  
xi1 = cell2mat(Xi(1,:)); % Convert each  
input state to matrix  
xi2 = cell2mat(Xi(2,:));
```

```
[y3,xfl,xf2] = maglevFcn(x1,x2,xi1,xi2);  
accuracy3 = max(abs(cell2mat(y)-y3))
```

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

$$\max(\text{abs}(\text{cell2mat}(y)-y4))$$

# **Chapter 6**

## **COMPETITIVE LAYERS**

---

# 6.1 FUNCTIONS

Competitive layers identify prototype vectors for clusters of examples using a simple neural network. The more important functions are the following.

<u><a href="#">competlayer</a></u>	Competitive layer
<u><a href="#">view</a></u>	View neural network
<u><a href="#">train</a></u>	Train neural network
<u><a href="#">trainru</a></u>	Unsupervised random order weight/t
<u><a href="#">learnk</a></u>	Kohonen weight learning function
<u><a href="#">learncon</a></u>	Conscience bias learning function

## 6.2 COMPETLAYER

Competitive layer

### Syntax

```
competlayer(numClasses,kohonenLR)
```

### 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,con)
```

these arguments,

numClasses	Number of classes to classify inputs (default = 1)
kohonenLR	Learning rate for Kohonen weights (default = 0.01)
conscienceLR	Learning rate for conscience bias (default = 0.01)

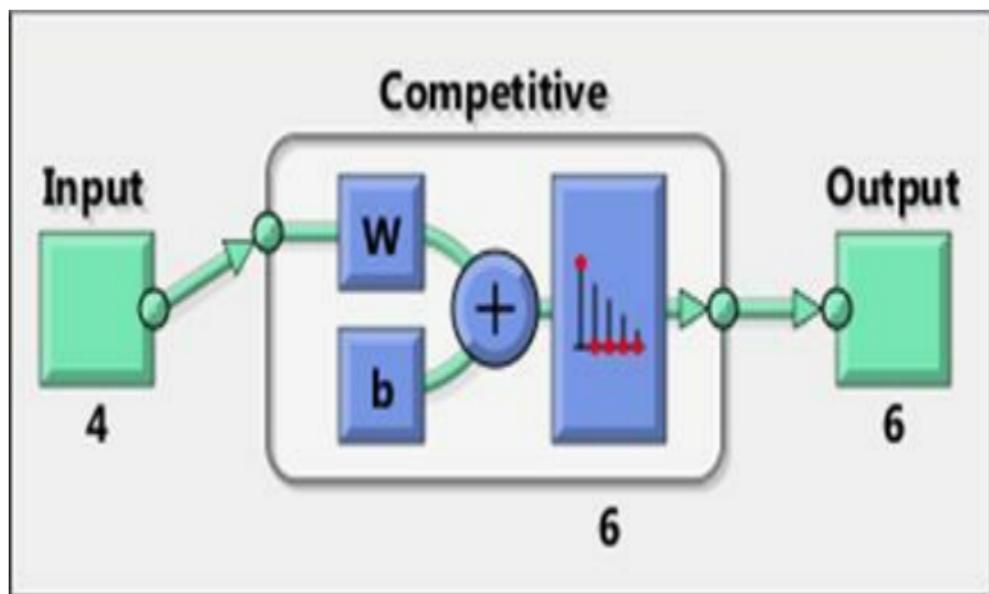
and returns a competitive layer with numClasses neurons.

## Examples. Create and Train a Competitive Layer

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

```
inputs = iris_dataset;
net = competlayer(6);
net = train(net,inputs);
view(net)
```

```
outputs = net(inputs);  
classes = vec2ind(outputs);
```



## 6.3 VIEW

View neural network

### Syntax

`view(net)`

### Description

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

### Example. View Neural Network

This example shows how to view the

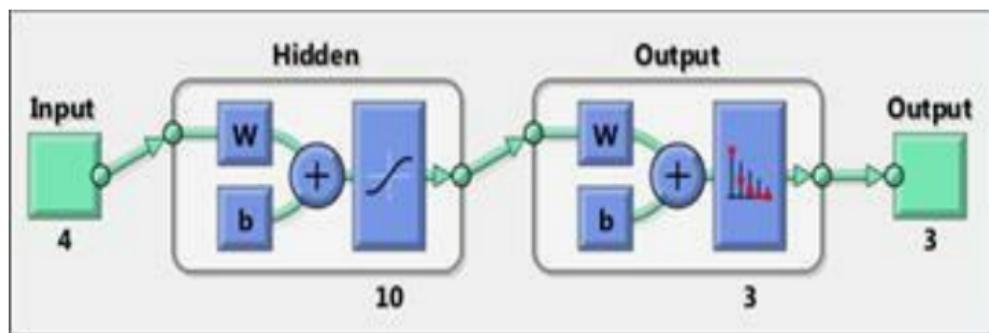
diagram of a pattern recognition network.

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

```
net = patternnet;
```

```
net = configure(net,x,t);
```

```
view(net)
```



## 6.4 TRAINRU

Unsupervised random order weight/bias training

### Syntax

```
net.trainFcn = 'trainru'  
[net,tr] = train(net,...)
```

### Description

trainru is not called directly. Instead it is called by train for networks whose net.trainFcn property is set to 'trainru', thus:

`net.trainFcn = 'trainru'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `trainru`.

`trainru` trains a network with weight and bias learning rules with incremental updates after each presentation of an input. Inputs are presented in random order.

Training occurs according to `trainru` training parameters, shown here with their default values:

<code>net.trainParam.epochs</code>	1000	Maximum num
<code>net.trainParam.show</code>	25	Epochs between
<code>net.trainParam.showCommandLine</code>	false	Generate comm
<code>net.trainParam.showWindow</code>	true	Show training C
<code>net.trainParam.time</code>	Inf	Maximum time

# Network Use

To prepare a custom network to be trained with `trainru`,

1. Set `net.trainFcn` to '`trainru`'. This sets `net.trainParam` to `trainru`'s default parameters.
2. Set each `net.inputWeights{i,j}.learnFcn` a learning function.
3. Set each `net.layerWeights{i,j}.learnFcn` a learning function.
4. Set each `net.biases{i}.learnFcn` to a learning function. (Weight and bias

learning parameters are automatically set to default values for the given learning function.)

To train the network,

1. Set `net.trainParam` properties to desired values.
2. Set weight and bias learning parameters to desired values.
3. Call `train`.

## Algorithms

For each epoch, all training vectors (or sequences) are each presented once in a

different random order, with the network and weight and bias values updated accordingly after each individual presentation.

Training stops when any of these conditions is met:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.

# 6.5 LEARNK

Kohonen weight learning function

## Syntax

[dW,LS] =

learnk(W,P,Z,N,A,T,E,gW,gA,D,LP,L)

info = learnk('code')

## Description

learnk is the Kohonen weight learning

function.

[dW,LS] =

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

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

and returns

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

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

LP.lr - 0.01	Learning rate
--------------	---------------

info = learnk('code') returns useful information for each *code* string:

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

## Examples

Here you define a random input P, output A, and weight matrix W for a layer with a two-element input and three

neurons. Also define the learning rate LR.

$p = \text{rand}(2,1);$

$a = \text{rand}(3,1);$

$w = \text{rand}(3,2);$

$lp.lr = 0.5;$

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

$dW = \text{learnk}(w, p, [], [], a, [], [], [], [], lp, [])$

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

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

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

## Algorithms

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

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

## 6.6 LEARNCON

Conscience bias learning function

### Syntax

[dB,LS] =

learncon(B,P,Z,N,A,T,E,gW,gA,D,LP)

info = learncon('code')

### Description

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

time.

[dB,LS] =

learncon(B,P,Z,N,A,T,E,gW,gA,D,LP,LS)  
several inputs,

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

and returns

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

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

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

info = learncon('code') returns useful information for each supported *code* string:

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

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

## Examples

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

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

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

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

[],lp,[])

## Network Use

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

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

.(Each weight learning parameter property is automatically set to learncon's default parameters.)

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

Set net.trainParam (or net.adaptParam) properties as desired.

Call train (or adapt).

## Algorithms

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

$$c = (1-lr)*c + lr*a$$

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

$$b = \exp(1 - \log(c)) - b$$

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

# **Chapter 7**

## **CLASSIFY PATTERNS WITH A NEURAL NETWORK**

---

## 7.1 INTRODUCTION

In addition to clustering for classification, neural networks are also good at recognizing patterns with the purpose of classifying.. For example, suppose you want to classify a tumor as benign or malignant, based on uniformity of cell size, clump thickness, mitosis, etc. You have 699 example cases for which you have 9 items of data and the correct classification as benign or malignant.

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

- Use the nprtool GUI, as described in [Using the Neural Network Pattern Recognition Tool](#).
- Use a command-line solution, as described in [Using Command-Line Functions](#).

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.

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. There are two approaches to creating the target vectors.

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

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

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

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

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

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

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

The next section shows how to train a network to recognize patterns, using the neural network pattern recognition tool GUI, [nprtool](#). This example uses the cancer data set provided with the toolbox. This data set

consists of 699 nine-element input vectors and two-element target vectors. There are two elements in each target vector, because there are two categories (benign or malignant) associated with each input vector.

## **7.2 USING THE NEURAL NETWORK PATTERN RECOGNITION TOOL**

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.

Getting Started Wizards

More Information

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 app

(nftool)

Pattern recognition and classification.



Pattern Recognition app

(npctool)

Clustering.



Clustering app

(nctool)

Dynamic Time series.



Time Series app

(ntstool)

- Click **Pattern Recognition Tool** to open the Neural Network Pattern Recognition Tool. (You can also use the command [nprtool](#).)



## Welcome to the Neural Pattern Recognition app.

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

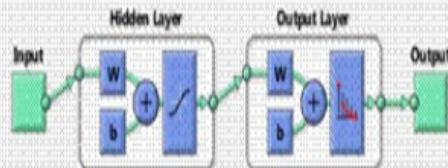
### Introduction

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

For example, recognize 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 Pattern Recognition app will help you select data, create and train a network, and evaluate its performance using cross-entropy and confusion matrices.

### Neural Network



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

The network will be trained with scaled conjugate gradient backpropagation ([theory](#)).

To continue, click [Next].

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

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

## 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 desired network output.

Targets:

(none)

Samples are

Matrix columns  Matrix rows

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

Load Example Data Set

Summary

No inputs selected.

No targets selected.

Select inputs and targets, then click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

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



Select a data set:

Simple Classes

Iris Flowers

Breast Cancer

Types of Glass

Thyroid

Wine Vintage

Description

Filename: [cancer dataset](#)

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.

This dataset can be used to design a neural network that classifies cancers as either benign or malignant depending on the characteristics of sample biopsies.

LOAD [cancer dataset](#).MAT loads these two variables:

cancerInputs - a 9x699 matrix defining nine attributes of 699 biopsies.

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

Import

Cancel

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



## Validation and Test Data

Set aside some samples for validation and testing.

### Select Percentages

Randomly divide up the 699 samples:

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

### Explanation

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.

**Restore Defaults**

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

Neural Network Start

Welcome

Back

Next

Cancel

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:

1. 70% are used for training.
2. 15% are used to validate that the network is generalizing and to stop training before overfitting.
3. The last 15% are used as a completely independent

test of network generalization.

• Click Next.

The standard network that is used for pattern recognition is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer, and a softmax transfer function in the output layer. The default number of hidden neurons is set to 10. You might want to come back and increase this number if the network does not perform as well as you expect. The number of output neurons is set to 2, which is equal to the number of elements in the target vector (the number of categories).

## Network Architecture

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

### Hidden Layer

Define a pattern recognition neural network... (patternnet)

Number of Hidden Neurons:

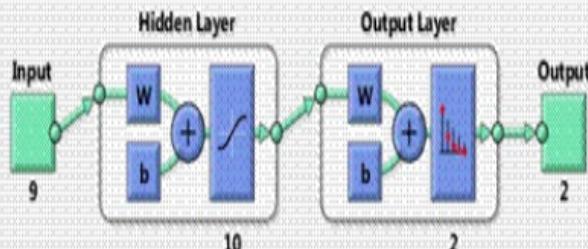
10

### Recommendation

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

[Restore Defaults](#)

### Neural Network



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

[Neural Network Start](#)

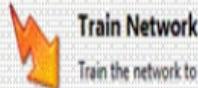
[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

- Click **Next**.



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 cross-entropy error of the validation samples.

### Notes

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

### Results

	Samples	CE	%
--	---------	----	---

	Training:	489	
	Validation:	105	
	Testing:	105	

[Plot Confusion](#)

[Plot ROC](#)

Minimizing Cross-Entropy results in good classification. 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].

[Neural Network Start](#)

[Welcome](#)

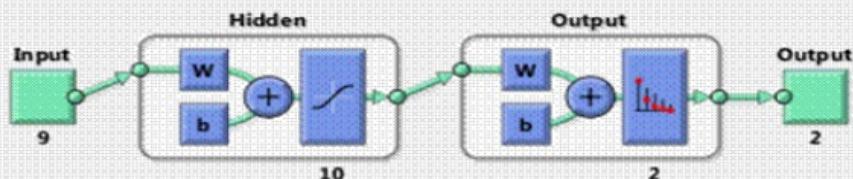
[Back](#)

[Next](#)

[Cancel](#)

· Click Train.

## Neural Network



## Algorithms

Data Division: Random (dividerand)  
 Training: Scaled Conjugate Gradient (trainscg)  
 Performance: Cross-Entropy (crossentropy)  
 Calculations: MEX

## Progress

Epoch:	0	20 iterations	1000
Time:		0:00:00	
Performance:	0.424	0.0236	0.00
Gradient:	0.442	0.0188	1.00e-06
Validation Checks:	0	6	6

## Plots

**Performance** (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Confusion (plotconfusion)

Receiver Operating Characteristic (plotroc)

Plot Interval:  1 epochs



Validation stop.



Stop Training



Cancel

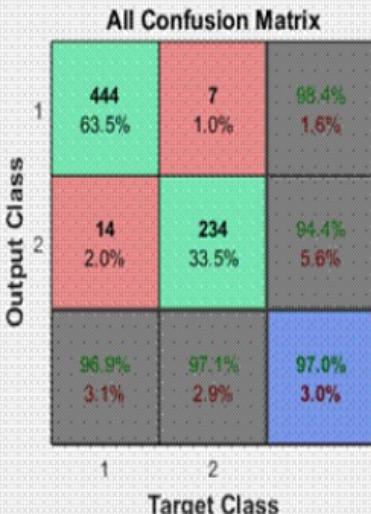
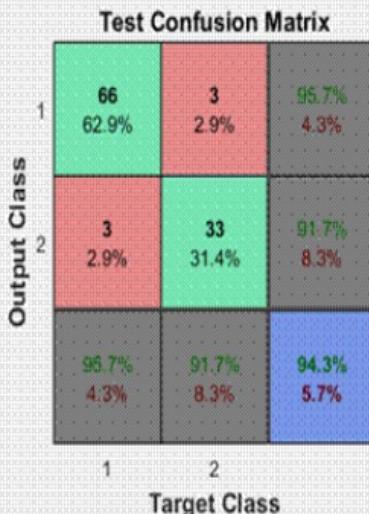
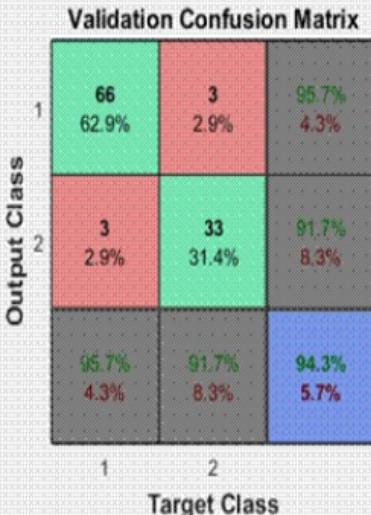
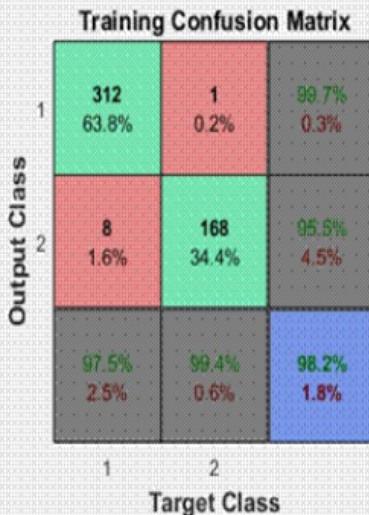
The training continues for 55 iterations.

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.

### Confusion (plotconfusion)

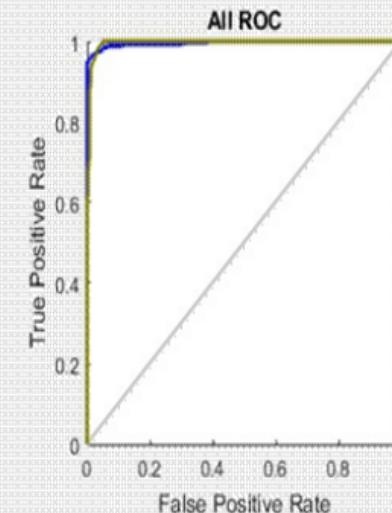
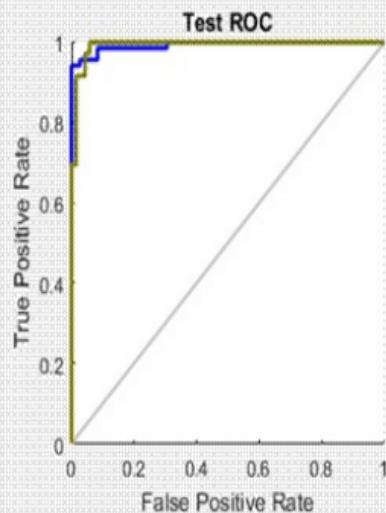
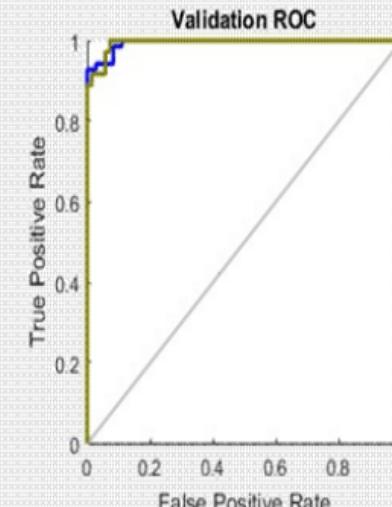
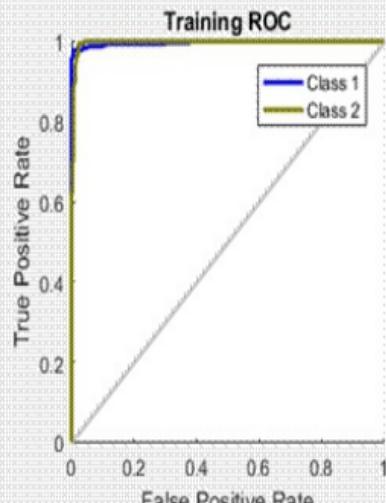
File Edit View Insert Tools Desktop Window Help



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

# Receiver Operating Characteristic (plotroc)

File Edit View Insert Tools Desktop Window Help



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



## 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.

Train Again

Increase network size if retraining did not help.

Adjust Network Size

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

Import Larger Data Set

Test Network

CE

PE

Plot Confusion

Plot ROC

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

Neural Network Start

Welcome

Back

Next

Cancel

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.

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

Use this panel to generate a MATLAB

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



## Deploy Solution

Generate deployable versions of your trained neural network.

### Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

Generate a MATLAB function with matrix and cell array argument support:

(genFunction)



### Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



### Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

Generate a Simulink diagram:

(gensim)



### Graphics

Generate a graphical diagram of the neural network:

(network/view)



Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

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

## Save Results



Generate MATLAB scripts, save results and generate diagrams.

Generate Scripts

**Recommended >>** Use these scripts to reproduce results and solve similar problems.

Generate a script to train and test a neural network as you just did with this tool



Generate a script with additional options and example code:



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

Restore Defaults



Save results and click [Finish].

Neural Network Start

Welcome

Back

Next

Finish

- You can click **Simple Script** or **Advanced Script** to create MATLAB<sup>®</sup> 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.
- 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.
- When you have saved your results, click **Finish**.



## 7.3 USING COMMAND-LINE FUNCTIONS

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

```
% Solve a Pattern Recognition Problem  
with a Neural Network
```

```
% Script generated by NPRTOOL
```

%

% This script assumes these variables  
are defined:

%

% cancerInputs - input data.

% cancerTargets - target data.

inputs = cancerInputs;

targets = cancerTargets;

% Create a Pattern Recognition Network

hiddenLayerSize = 10;

net = patternnet(hiddenLayerSize);

% Set up Division of Data for Training,  
Validation, Testing

net.divideParam.trainRatio = 70/100;

net.divideParam.valRatio = 15/100;

net.divideParam.testRatio = 15/100;

% Train the Network

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

% Test the Network

outputs = net(inputs);

errors = gsubtract(targets,outputs);

```
performance =  
perform(net,targets,outputs)
```

```
% View the Network
```

```
view(net)
```

```
% Plots
```

```
% Uncomment these lines to enable  
various plots.
```

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

```
% figure, plottrainstate(tr)
```

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

```
% figure, ploterrhist(errors)
```

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

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

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

Create the network. The default network for function fitting (or regression) problems, [patternnet](#), is a feedforward network with the default tan-sigmoid transfer function in the hidden layer, and

a softmax transfer function in the output layer. You assigned ten neurons (somewhat arbitrary) to the one hidden layer in the previous section.

The network has two output neurons, because there are two target values (categories) associated with each input vector.

Each output neuron represents a category.

When an input vector of the appropriate category is applied to the network, the corresponding neuron should produce a 1, and the other neurons should output a 0.

To create the network, enter these

commands:

```
hiddenLayerSize = 10;
```

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

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

as elements of an array in the [patternnet](#) command.

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.

Train the network. The pattern recognition network uses the default Scaled Conjugate Gradient ([trainscg](#)) algorithm for training. To train the

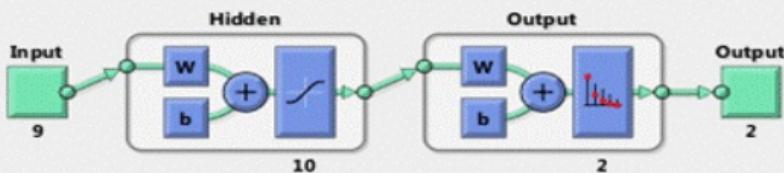
network, enter this command:

```
[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: Random (dividerand)  
Training: Scaled Conjugate Gradient (trainscg)  
Performance: Cross-Entropy (crossentropy)  
Derivative: Default (defaultderiv)

## Progress

Epoch:	0	9 iterations	1000
Time:		0:00:00	
Performance:	0.909	0.0427	0.00
Gradient:	1.40	0.0243	1.00e-06
Validation Checks:	0	6	6

## Plots

- Performance** (plotperform)
- Training State (plottrainstate)
- Error Histogram (ploterrhist)
- Confusion (plotconfusion)
- Receiver Operating Characteristic (plotroc)

Plot Interval:



Validation stop.



Stop Training



Cancel

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

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

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

```
errors = gsubtract(targets,outputs);
```

```
performance =
```

```
perform(net,targets,outputs)
```

```
performance =
```

0.0307

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

```
tInd = tr.testInd;
```

```
tstOutputs = net(inputs(:,tInd));
```

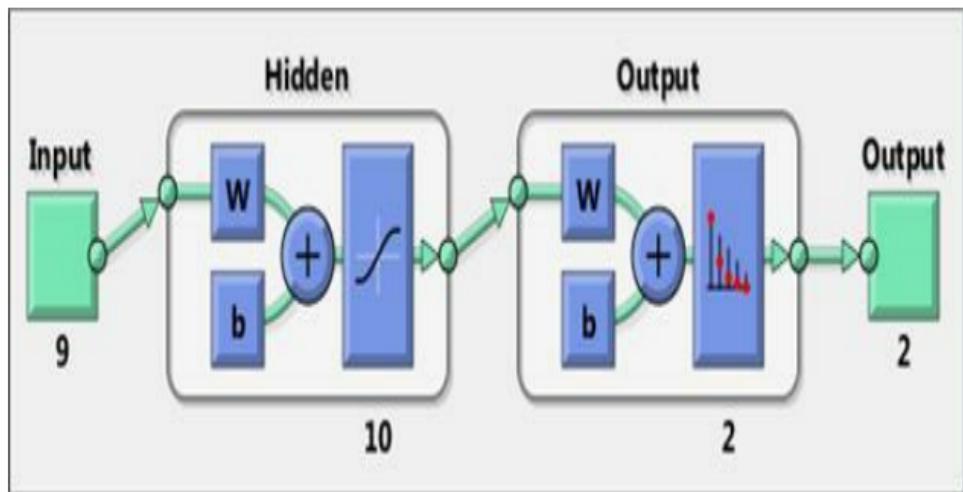
```
tstPerform =  
perform(net,targets(:,tInd),tstOutputs)
```

```
tstPerform =
```

0.0257

View the network diagram.

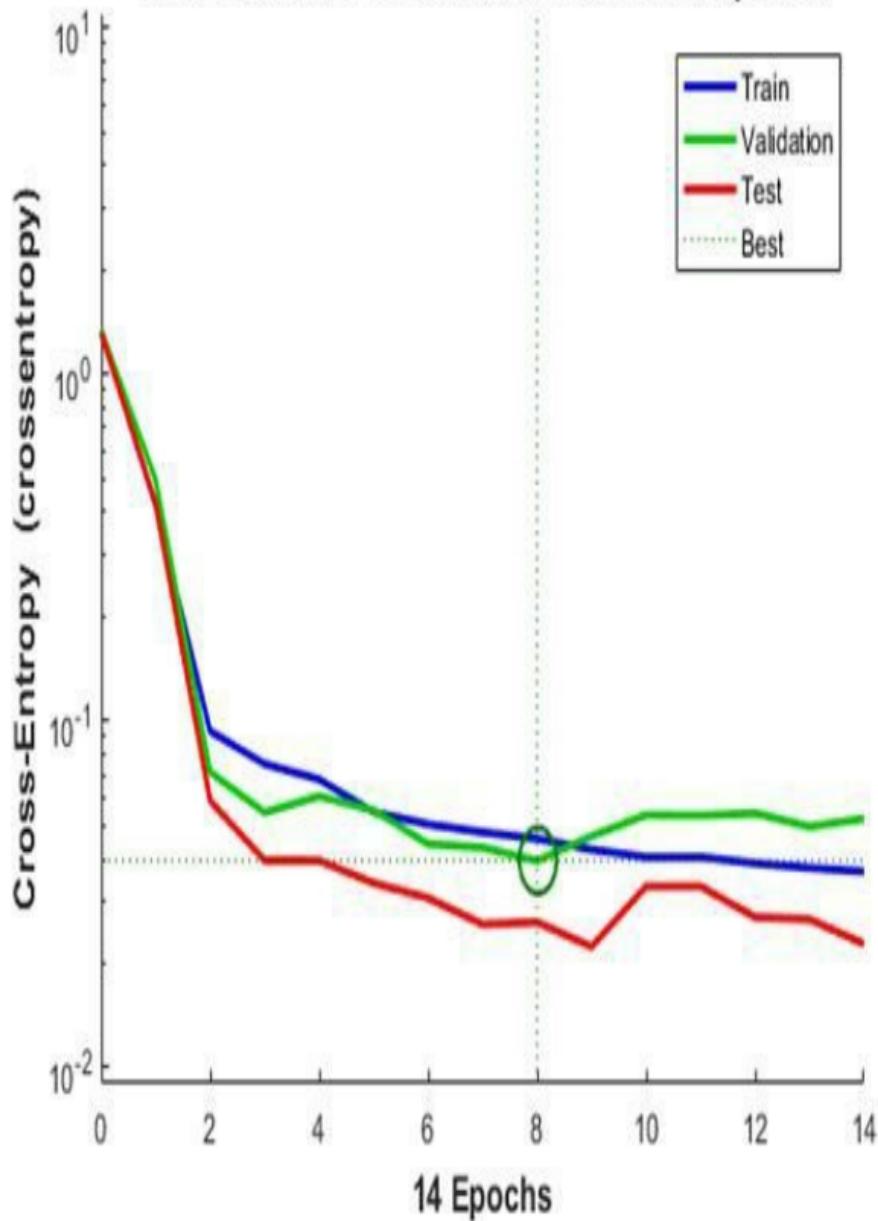
`view(net)`



Plot the training, validation,  
and test performance.

`figure, plotperform(tr)`

Best Validation Performance is 0.039514 at epoch 8



Use the [plotconfusion](#) function to plot the confusion matrix. It shows the various types of errors that occurred for the final trained network.

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

### Confusion Matrix

		Target Class
Output Class	1	2
	1	2
1	446 63.8%	5 0.7%
2	12 1.7%	236 33.8%
	97.4% 2.6%	97.9% 2.1%
		97.6% 2.4%

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

- Reset the initial network weights and biases to new values with [init](#) and train again.
- Increase the number of hidden neurons.

- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm (see "[Training Algorithms](#)").

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

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

- During training, open a plot window (such as the confusion

plot), and watch it animate.

- Plot from the command line with functions such as [plotroc](#) and [plottrainstate](#).

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

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

been found, retrain several times.

# **Chapter 8**

## **WORK FLOW FOR NEURAL NETWORK DESIGN**

---

# 8.1 INTRODUCTION

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

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

Data collection in step 1 generally

occurs outside the framework of Neural Network Toolbox software. Details of the other steps and discussions of steps 4, 6, and 7, are discussed in topics specific to the type of network.

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

After a neural network has been created, it needs to be configured and then trained. Configuration involves arranging

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

## **8.2 FOUR LEVELS OF NEURAL NETWORK DESIGN**

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

The second level of toolbox use is through basic command-line operations. The command-line functions use simple argument lists with intelligent default

settings for function parameters. (You can override all of the default settings, for increased functionality.) This topic, and the ones that follow, concentrate on command-line operations.

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

A third level of toolbox use is customization of the toolbox. This advanced capability allows you to create

your own custom neural networks, while still having access to the full functionality of the toolbox.

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

# **8.3 MULTILAYER NEURAL NETWORKS AND BACKPROPAGATION TRAINING**

The multilayer feedforward neural network is the workhorse of the Neural Network Toolbox™ software. It can be used for both function fitting and pattern recognition problems. With the addition of a tapped delay line, it can also be used for prediction problems. This topic shows how you can use a multilayer network. It also illustrates the basic procedures for designing any neural network.

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

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

1. Collect data
2. Create the network
3. Configure the network
4. Initialize the weights and biases
5. Train the network

6. Validate the network (post-training analysis)
7. Use the network

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

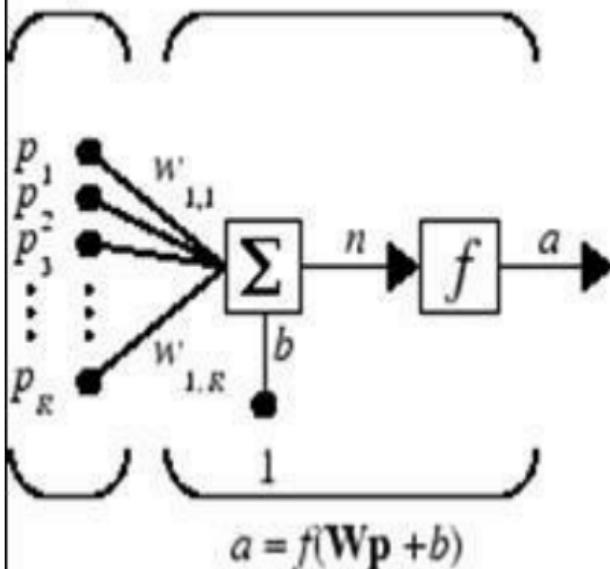
# **8.4 MULTILAYER NEURAL NETWORK ARCHITECTURE**

This topic presents part of a typical multilayer network workflow.

## 8.4.1 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.

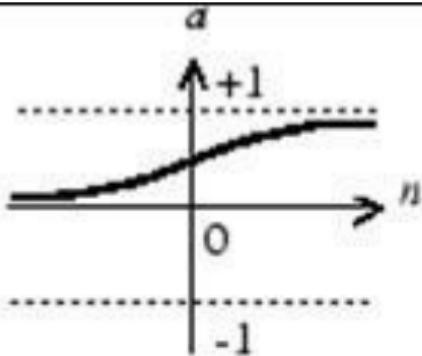
## Input General Neuron



Where

$R$  = number of elements in input vector

Multilayer networks often use the log-sigmoid transfer function [logsig](#).

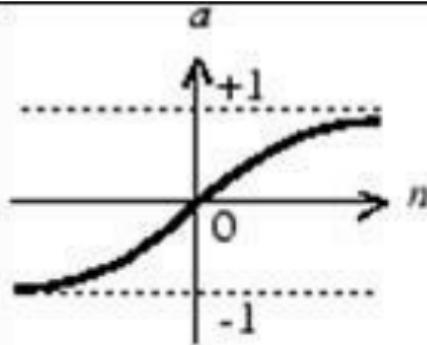


$$a = \text{logsig}(n)$$

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.

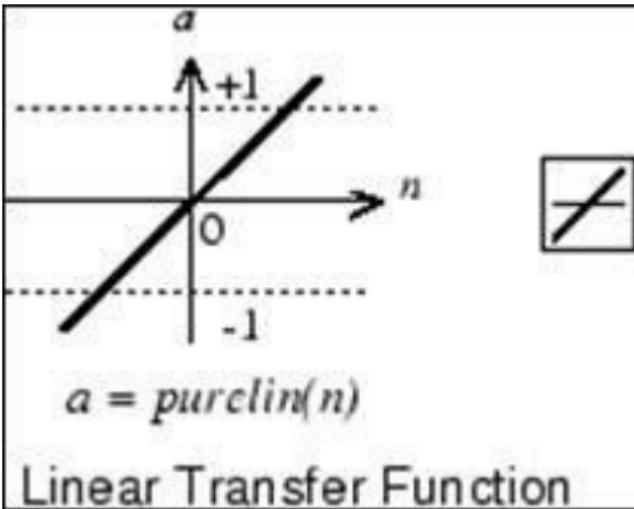
Alternatively, multilayer networks can use the tan-sigmoid transfer function tansig.



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

### Tan-Sigmoid Transfer Function

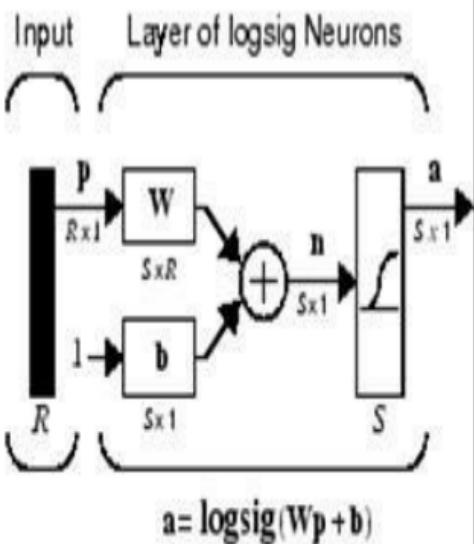
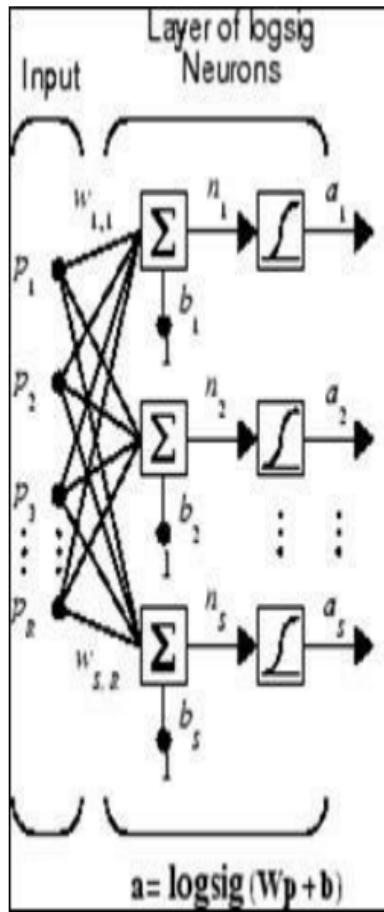
Sigmoid output neurons are often used for pattern recognition problems, while linear output neurons are used for function fitting problems. The linear transfer function [purelin](#) is shown below.



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.

## 8.4.2 Feedforward Neural Network

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



Where...

$R$  = number of elements in input vector

$S$  = number of neurons in layer

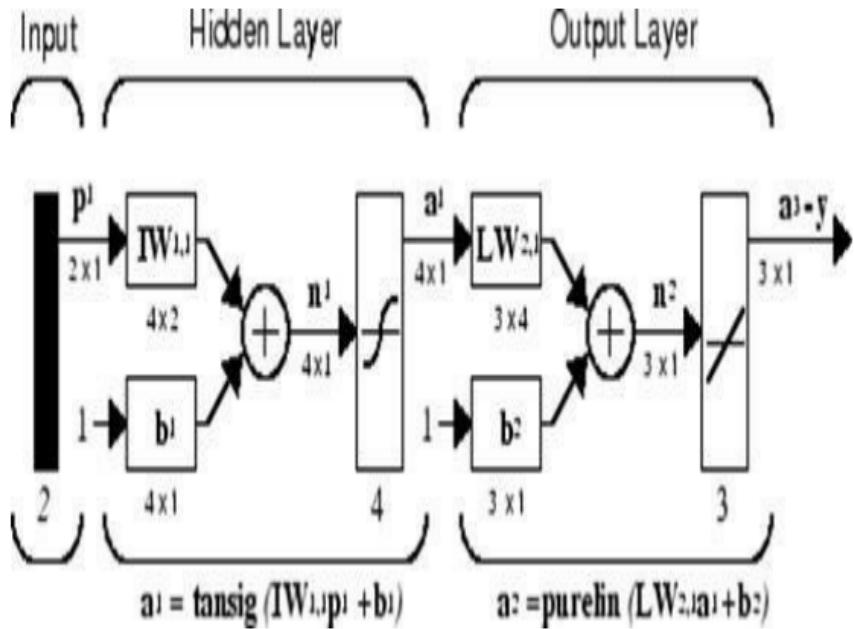
Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of

neurons with nonlinear transfer functions allow the network to learn nonlinear relationships between input and output vectors. The linear output layer is most often used for function fitting (or nonlinear regression) problems.

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

For multiple-layer networks the

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



This network can be used as a general function approximator. It can approximate any function with a finite

number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer.

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

# **8.5 UNDERSTANDING NEURAL NETWORK TOOLBOX DATA STRUCTURES**

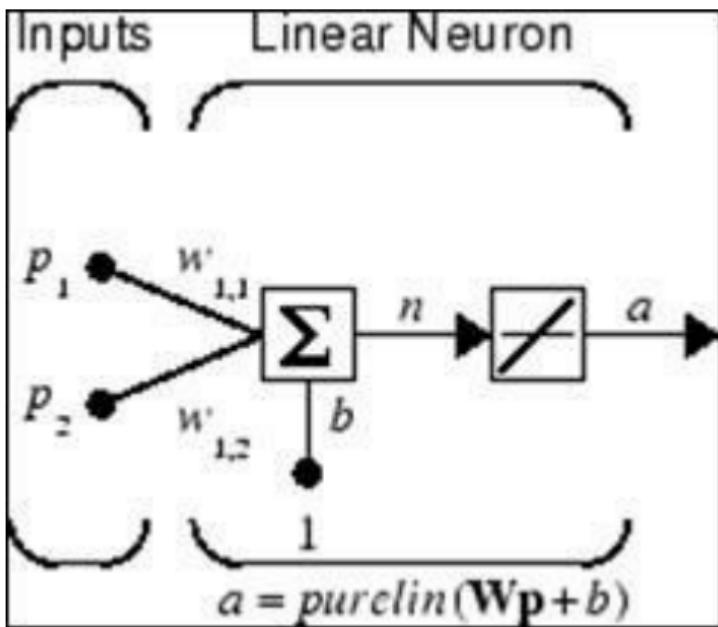
This topic discusses how the format of input data structures affects the simulation of networks. It starts with static networks, and then continues with dynamic networks. The following section describes how the format of the data structures affects network training.

There are two basic types of input vectors: those that occur concurrently (at the same time, or in no particular time

sequence), and those that occur sequentially in time. For concurrent vectors, the order is not important, and if there were a number of networks running in parallel, you could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

## 8.5.1 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  $W = [1 \ 2]$  and  $b = [0]$ .

The commands for these assignments are

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

```
net.b{1} = 0;
```

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

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

Concurrent vectors are presented to

the network as a single matrix:

$$P = [1 \ 2 \ 2 \ 3; 2 \ 1 \ 3 \ 1];$$

You can now simulate the network:

$$A = \text{net}(P)$$

$$A =$$

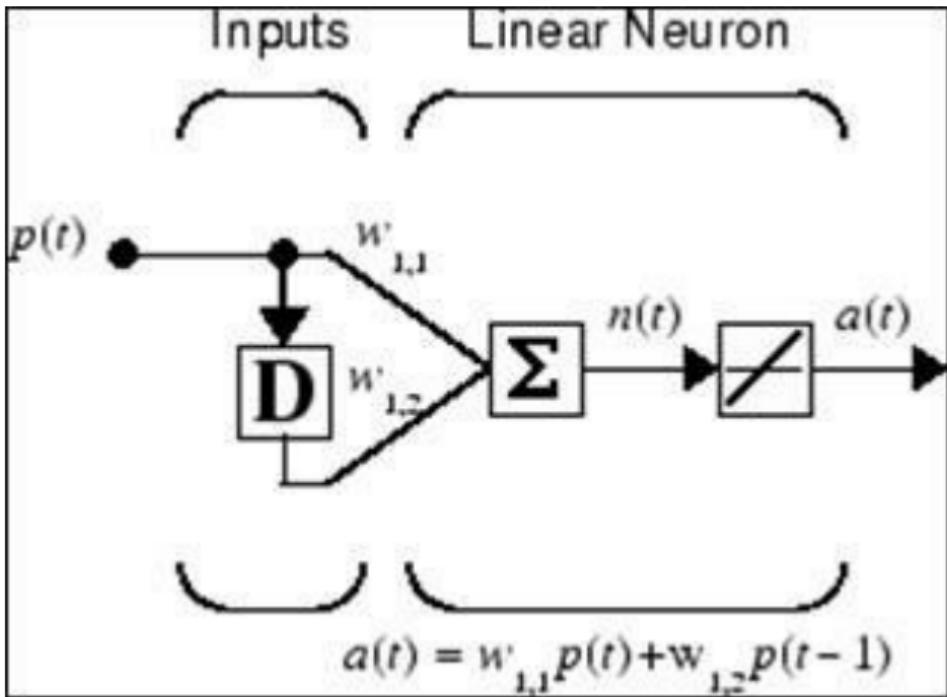
$$\begin{matrix} 5 & 4 & 8 & 5 \end{matrix}$$

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.

## 8.5.2 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  $W = [1 \ 2]$ .

The command is:

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

Suppose that the input sequence is:

$p_1 = [1], p_2 = [2], p_3 = [3], p_4 = [4]$

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

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

You can now simulate the network:

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

A =

[1] [4] [7] [10]

You input a cell array containing a sequence of inputs, and the network produces a cell array containing a sequence of outputs. The order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result. If you were to change the order of the inputs, the numbers obtained in the output would change.

### 8.5.3 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, if you use a concurrent set of inputs you have

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

which can be created with the following code:

$$P = [1 \ 2 \ 3 \ 4];$$

When you simulate with concurrent inputs, you obtain

$$A = \text{net}(P)$$

$$A =$$

$$\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}$$

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:

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

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:

$$A = \text{net}(P);$$

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

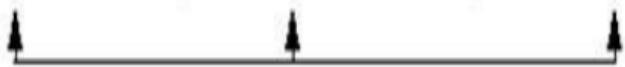
between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

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.

*Q*th Sequence



$\{[p_1(1), p_2(1), \dots, p_Q(1)], [p_1(2), p_2(2), \dots, p_Q(2)], \dots, [p_1(TS), p_2(TS), \dots, p_Q(TS)]\}$



First Sequence

# 8.6 NEURAL NETWORK OBJECT PROPERTIES

## 8.6.1 General

Here are the general properties of neural networks.

### **net.name**

This property consists of a string defining the network name. Network creation functions, such as [feedforwardnet](#), define this appropriately. But it can be set to any string as desired.

## **net userdata**

This property provides a place for users to add custom information to a network object. Only one field is predefined. It contains a *secret* message to all Neural Network Toolbox users:

### **net userdata.note**

## 8.6.2 Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

### **net.numInputs**

This property defines the number of inputs a network receives. It can be set to 0 or a positive integer.

**Clarification.** The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors

the network receives as input. The size of each input (i.e., the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

**Side Effects.** Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

## **net.numLayers**

This property defines the number of layers a network has. It can be set to 0 or a positive integer.

**Side Effects.** Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers:

`net.biasConnect`

`net.inputConnect`

`net.layerConnect`

`net.outputConnect`

and changes the size of each cell array of subobject structures whose size depends on the number of layers:

`net.biases`

net.inputWeights  
net.layerWeights  
net.outputs

and also changes the size of each of the network's adjustable parameter's properties:

net.IW  
net.LW  
net.b

## **net.biasConnect**

This property defines which layers have biases. It can be set to any  $N$ -by-1 matrix of Boolean values, where  $N_l$  is the number of network layers

(net.numLayers). The presence (or absence) of a bias to the  $i$ th layer is indicated by a 1 (or 0) at

`net.biasConnect(i)`

**Side Effects.** Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

## **net.inputConnect**

This property defines which layers have weights coming from inputs.

It can be set to any  $N_l \times N_i$  matrix of

Boolean values, where  $N_l$  is the number of network layers (`net.numLayers`), and  $N_i$  is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the  $i$ th layer from the  $j$ th input is indicated by a 1 (or 0) at `net.inputConnect(i,j)`.

**Side Effects.** Any change to this property alters the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

## **net.layerConnect**

This property defines which layers have weights coming from other layers. It can be set to any  $N_l \times N_l$  matrix of Boolean values, where  $N_l$  is the number of network layers (net.numLayers). The presence (or absence) of a weight going to the  $i$ th layer from the  $j$ th layer is indicated by a 1 (or 0) at

```
net.layerConnect(i,j)
```

**Side Effects.** Any change to this property alters the presence or absence of structures in the cell array of layer weight subobjects (net.layerWeights) and the presence or absence of matrices in the cell array of layer weight matrices (net.LW).

## **net.outputConnect**

This property defines which layers generate network outputs. It can be set to any  $1 \times N_l$  matrix of Boolean values, where  $N_l$  is the number of network layers (net.numLayers). The presence (or absence) of a network output from the  $i$ th layer is indicated by a 1 (or 0) at net.outputConnect( $i$ ).

**Side Effects.** Any change to this property alters the number of network outputs (net.numOutputs) and the presence or absence of structures in the cell array of output subobjects (net.outputs).

## **net.numOutputs (read only)**

This property indicates how many outputs the network has. It is always equal to the number of 1s in net.outputConnect.

## **net.numInputDelays (read only)**

This property indicates the number of time steps of past inputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's input weights:

```
numInputDelays = 0;  
for i=1:net.numLayers
```

```
for j=1:net.numInputs
    if net.inputConnect(i,j)
        numInputDelays = max( ...
            [numInputDelays
            net.inputWeights{i,j}.delays]);
    end
end
end
```

## **net.numLayerDelays (read only)**

This property indicates the number of time steps of past layer outputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's layer weights:

```
numLayerDelays = 0;
for i=1:net.numLayers
    for j=1:net.numLayers
        if net.layerConnect(i,j)
            numLayerDelays = max( ...
                [numLayerDelays
                net.layerWeights {i,j} .delays]);
        end
    end
end
```

## **net.numWeightElements (read only)**

This property indicates the number of weight and bias values in the network. It is the sum of the number of elements in

the matrices stored in the two cell arrays:

net.IW

new.b

## 8.6.3 Subobject Structures

These properties consist of cell arrays of structures that define each of the network's inputs, layers, outputs, targets, biases, and weights.

### **net.inputs**

This property holds structures of properties for each of the network's inputs. It is always an  $N_i \times 1$  cell array of input structures, where  $N_i$  is the number of network inputs (net.numInputs).

The structure defining the properties of

the  $i$ th network input is located at  
`net.inputs{i}`

If a neural network has only one input, then you can access `net.inputs{1}` without the cell array notation as follows:

`net.input`

## **net.layers**

This property holds structures of properties for each of the network's layers. It is always an  $N_l \times 1$  cell array of layer structures, where  $N_l$  is the number of network layers (`net.numLayers`).

The structure defining the properties of the  $i$ th layer is located at `net.layers{i}`.

## **net.outputs**

This property holds structures of properties for each of the network's outputs. It is always a  $1 \times N_l$  cell array, where  $N_l$  is the number of network outputs (`net.numOutputs`).

The structure defining the properties of the output from the  $i$ th layer (or a null matrix `[]`) is located at `net.outputs{i}` if `net.outputConnect(i)` is 1 (or 0).

If a neural network has only one output

at layer  $i$ , then you can access `net.outputs{i}` without the cell array notation as follows:

```
net.output
```

## **net.biases**

This property holds structures of properties for each of the network's biases. It is always an  $N_l \times 1$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`).

The structure defining the properties of the bias associated with the  $i$ th layer (or a null matrix `[]`) is located at `net.biases{i}` if `net.biasConnect(i)` is

1 (or 0).

## **net.inputWeights**

This property holds structures of properties for each of the network's input weights. It is always an  $N_l \times N_i$  cell array, where  $N_l$  is the number of network layers (`net.numLayers`), and  $N_i$  is the number of network inputs (`net.numInputs`).

The structure defining the properties of the weight going to the  $i$ th layer from the  $j$ th input (or a null matrix []) is located at `net.inputWeights{i,j}` if `net.inputConnect(i,j)` is 1 (or 0).

## **net.layerWeights**

This property holds structures of properties for each of the network's layer weights. It is always an  $N_l \times N_l$  cell array, where  $N_l$  is the number of network layers (net.numLayers).

The structure defining the properties of the weight going to the  $i$ th layer from the  $j$ th layer (or a null matrix []) is located at net.layerWeights{i,j} if net.layerConnect(i,j) is 1 (or 0).

## 8.6.4 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 nnnderivative`.

### **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.divideFcn)
```

## **net.divideMode**

This property defines the target data dimensions which to divide up when the data division function is called. Its default value is 'sample' for static networks and 'time' for dynamic networks. It may also be set to 'sampletime' to divide targets by both sample and timestep, 'all' to divide up targets by every scalar value, or 'none' to not divide up data at all (in which case all data is used for training, none for validation or testing).

## **net.initFcn**

This property defines the function used to initialize the network's weight matrices and bias vectors. . The initialization function is used to initialize the network whenever [init](#) is called:

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

**Side Effects.** Whenever this property is altered, the network's initialization parameters (`net.initParam`) are set to contain the parameters and default values of the new function.

## **net.initParam**

This property defines the parameters and values of the current initialization function. Call `help` on the current

initialization function to get a description of what each field means:

```
help(net.initFcn)
```

## **net.performFcn**

This property defines the function used to measure the network's performance. The performance function is used to calculate network performance during training whenever [train](#) is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nnperformance`.

**Side Effects.** Whenever this property is

altered, the network's performance parameters (`net.performParam`) are set to contain the parameters and default values of the new function.

## **net.performParam**

This property defines the parameters and values of the current performance function. Call `help` on the current performance function to get a description of what each field means:

`help(net.performFcn)`

## **net.plotFcns**

This property consists of a row cell array of strings, defining the plot

functions associated with a network. The neural network training window, which is opened by the [train](#) function, shows a button for each plotting function. Click the button during or after training to open the desired plot.

## **net.plotParams**

This property consists of a row cell array of structures, defining the parameters and values of each plot function in `net.plotFcns`. Call `help` on the each plot function to get a description of what each field means:

```
help(net.plotFcns{i})
```

## **net.trainFcn**

This property defines the function used to train the network. It can be set to the name of any of the training functions, which is used to train the network whenever [train](#) is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nntrain`.

**Side Effects.** Whenever this property is altered, the network's training parameters (`net.trainParam`) are set to contain the parameters and default values of the new function.

## **net.trainParam**

This property defines the parameters and

values of the current training function.  
Call help on the current training function  
to get a description of what each field  
means:

```
help(net.trainFcn)
```

## 8.6.5 Weight and Bias Values

These properties define the network's adjustable parameters: its weight matrices and bias vectors.

### **net.IW**

This property defines the weight matrices of weights going to layers from network inputs. It is always an  $N_l \times N_i$  cell array, where  $N_l$  is the number of network layers (net.numLayers), and  $N_i$  is the number of network inputs (net.numInputs).

The weight matrix for the weight going to the  $i$ th layer from the  $j$ th input (or a

null matrix  $[]$ ) is located at  $\text{net.IW}\{i,j\}$  if  $\text{net.inputConnect}(i,j)$  is 1

The weight matrix has as many rows as the size of the layer it goes to ( $\text{net.layers}\{i\}.\text{size}$ ). It has as many columns as the product of the input size with the number of delays associated with the weight:

$\text{net.inputs}\{j\}.\text{size}$  \*  
 $\text{length}(\text{net.inputWeights}\{i,j\}.\text{delays})$

These dimensions can also be obtained from the input weight properties:

$\text{net.inputWeights}\{i,j\}.\text{size}$

**net.LW**

This property defines the weight matrices of weights going to layers from other layers. It is always an  $N_l \times N_l$  cell array, where  $N_l$  is the number of network layers (net.numLayers).

The weight matrix for the weight going to the  $i$ th layer from the  $j$ th layer (or a null matrix []) is located at net.LW{i,j} if net.layerConnect(i,j) is 1 (or 0).

The weight matrix has as many rows as the size of the layer it goes to (net.layers{i}.size). It has as many columns as the product of the size of the layer it comes from with the number of delays associated with the weight:

```
net.layers{j}.size
```

```
length(net.layerWeights{i,j}.delays)
```

These dimensions can also be obtained from the layer weight properties:

```
net.layerWeights{i,j}.size
```

## **net.b**

This property defines the bias vectors for each layer with a bias. It is always an  $N_l \times 1$  cell array, where  $N_l$  is the number of network layers (net.numLayers).

The bias vector for the  $i$ th layer (or a null matrix []) is located at net.b{i} if net.biasConnect(i) is 1 (or 0).

The number of elements in the bias vector is always equal to the size of the layer it is associated with (`net.layers{i}.size`).

This dimension can also be obtained from the bias properties:

`net.biases{i}.size`

## 8.7 NEURAL NETWORK SUBOBJECT PROPERTIES

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

- Inputs
- Layers
- Outputs
- Biases
- Input Weights
- Layer Weights

## 8.7.1 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 exampleInput and then used to provide consistent results to new input values before the network uses them.

**net.inputs{i}.processedRange (read only)**

This property defines the range of exampleInput values after they have been processed with processingFcns and processingParamFcns.

**net.inputs{i}.processedSize (read only)**

This property defines the number of rows in the exampleInput values after they have been processed with processingFcns and processingParamFcns.

**net.inputs{i}.range**

This property defines the range of each element of the  $i$ th network input.

It can be set to any  $R_i \times 2$  matrix, where  $R_i$  is the number of elements in the input (`net.inputs{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each  $j$ th row defines the minimum and maximum values of the  $j$ th input element, in that order:

`net.inputs{i}(j,:)`

**Uses.** Some initialization functions use input ranges to find appropriate initial

values for input weight matrices.

**Side Effects.** Whenever the number of rows in this property is altered, the input size, processedSize, and processedRange change to remain consistent. The sizes of any weights coming from this input and the dimensions of the weight matrices also change.

### **net.inputs{i}.size**

This property defines the number of elements in the  $i$ th network input. It can be set to 0 or a positive integer.

**Side Effects.** Whenever this property is

altered, the input range, processedRange, and processedSize are updated. Any associated input weights change size accordingly.

### **net.inputs{i}.userdata**

This property provides a place for users to add custom information to the *i*th network input.

## 8.7.2 Layers

These properties define the details of each  $i$ th network layer.

### **net.layers{i}.name**

This property consists of a string defining the layer name. Network creation functions, such as [feedforwardnet](#), define this appropriately. But it can be set to any string as desired.

### **net.layers{i}.dimensions**

This property defines

the *physical* dimensions of the *i*th layer's neurons. Being able to arrange a layer's neurons in a multidimensional manner is important for self-organizing maps.

It can be set to any row vector of 0 or positive integer elements, where the product of all the elements becomes the number of neurons in the layer (`net.layers{i}.size`).

**Uses.** Layer dimensions are used to calculate the neuron positions within the layer (`net.layers{i}.positions`) using the layer's topology function (`net.layers{i}.topologyFcn`).

**Side Effects.** Whenever this property is altered, the layer's size (`net.layers{i}.size`) changes to remain consistent. The layer's neuron positions (`net.layers{i}.positions`) and the distances between the neurons (`net.layers{i}.distances`) are also updated.

### **`net.layers{i}.distanceFcn`**

This property defines which of the distance functions is used to calculate distances between neurons in the *i*th layer from the neuron positions. Neuron distances are used by self-organizing maps. It can be set to the name of any distance function.

For a list of functions, type help 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 layer initialization functions are used to initialize the *i*th layer, if the network initialization function (`net.initFcn`) is [initlay](#). If the network initialization is set to [initlay](#), then the function indicated by this property is used to initialize the layer's weights and biases.

### **`net.layers{i}.netInputFcn`**

This property defines which of the net input functions is used to calculate the  $i$ th layer's net input, given the layer's weighted inputs and bias during simulating and training.

For a list of functions, type `help nnnetinput`.

## **net.layers{i}.netInputParam**

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field:

`help(net.layers{i}.netInputFcn)`

## **net.layers{i}.positions (read only)**

This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

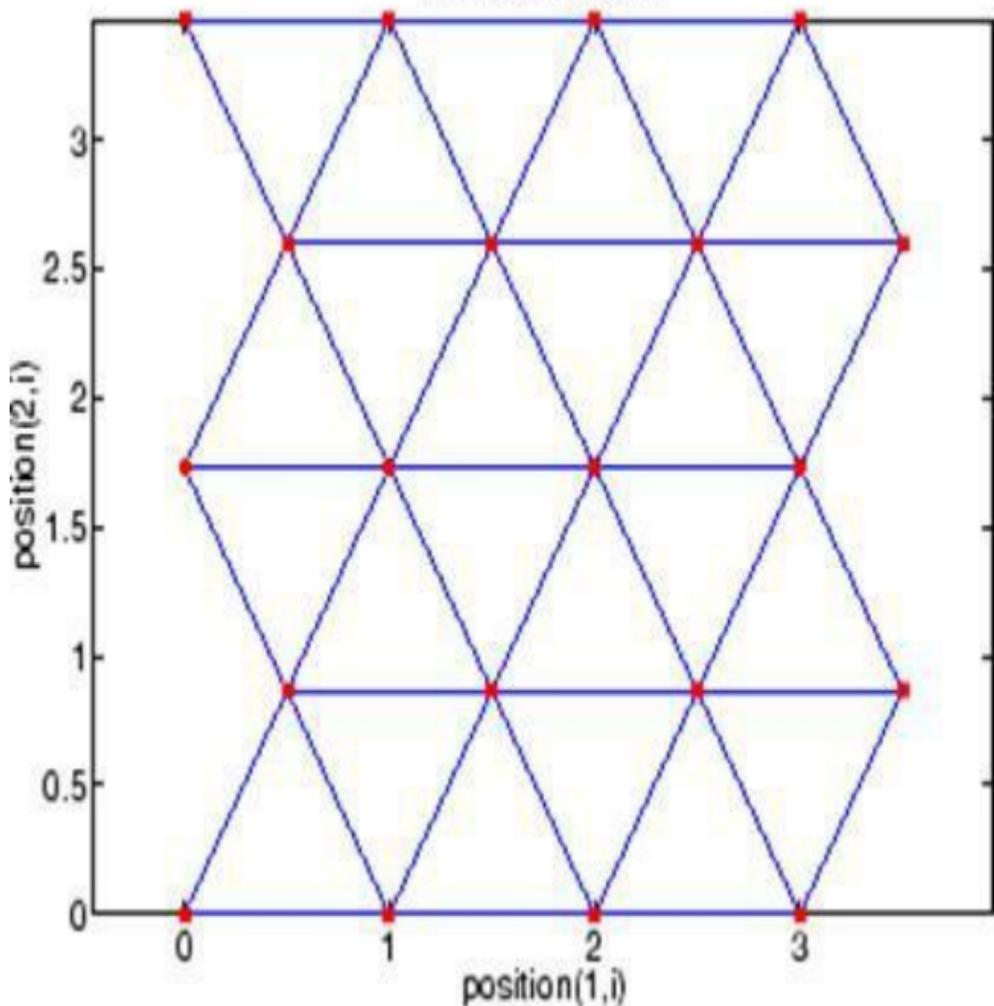
**Plotting.** Use [plotsom](#) to plot the positions of a layer's neurons.

For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [4 5], and

the topology function  
(net.layers{1}.topologyFcn) is [hextop](#),  
the neurons' positions can be plotted as follows:

```
plotsom(net.layers{1}.positions)
```

Neuron Positions



**net.layers{i}.range (read only)**

This property defines the output range of

each neuron of the  $i$ th layer.

It is set to an  $S_i \times 2$  matrix, where  $S_i$  is the number of neurons in the layer (net.layers{i}.size), and each element in column 1 is less than the element next to it in column 2.

Each  $j$ th row defines the minimum and maximum output values of the layer's transfer

function net.layers{i}.transferFcn.

### **net.layers{i}.size**

This property defines the number of neurons in the  $i$ th layer. It can be set to 0

or a positive integer.

**Side Effects.** Whenever this property is altered, the sizes of any input weights going to the layer (`net.inputWeights{i,:}.size`), any layer weights going to the layer (`net.layerWeights{i,:}.size`) or coming from the layer (`net.inputWeights{i,:}.size`), and the layer's bias (`net.biases{i}.size`), change.

The dimensions of the corresponding weight matrices (`net.IW{i,:}`, `net.LW{i,:}`, `net.LW{:,i}`), and biases (`net.b{i}`) also change.

Changing this property also changes the

size of the layer's output (`net.outputs{i}.size`) and target (`net.targets{i}.size`) if they exist.

Finally, when this property is altered, the dimensions of the layer's neurons (`net.layers{i}.dimension`) are set to the same value. (This results in a one-dimensional arrangement of neurons. If another arrangement is required, set the dimensions property directly instead of using size.)

### **`net.layers{i}.topologyFcn`**

This property defines which of the topology functions are used to calculate the *i*th layer's neuron positions

(`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

For a list of functions, type `help nntopology`.

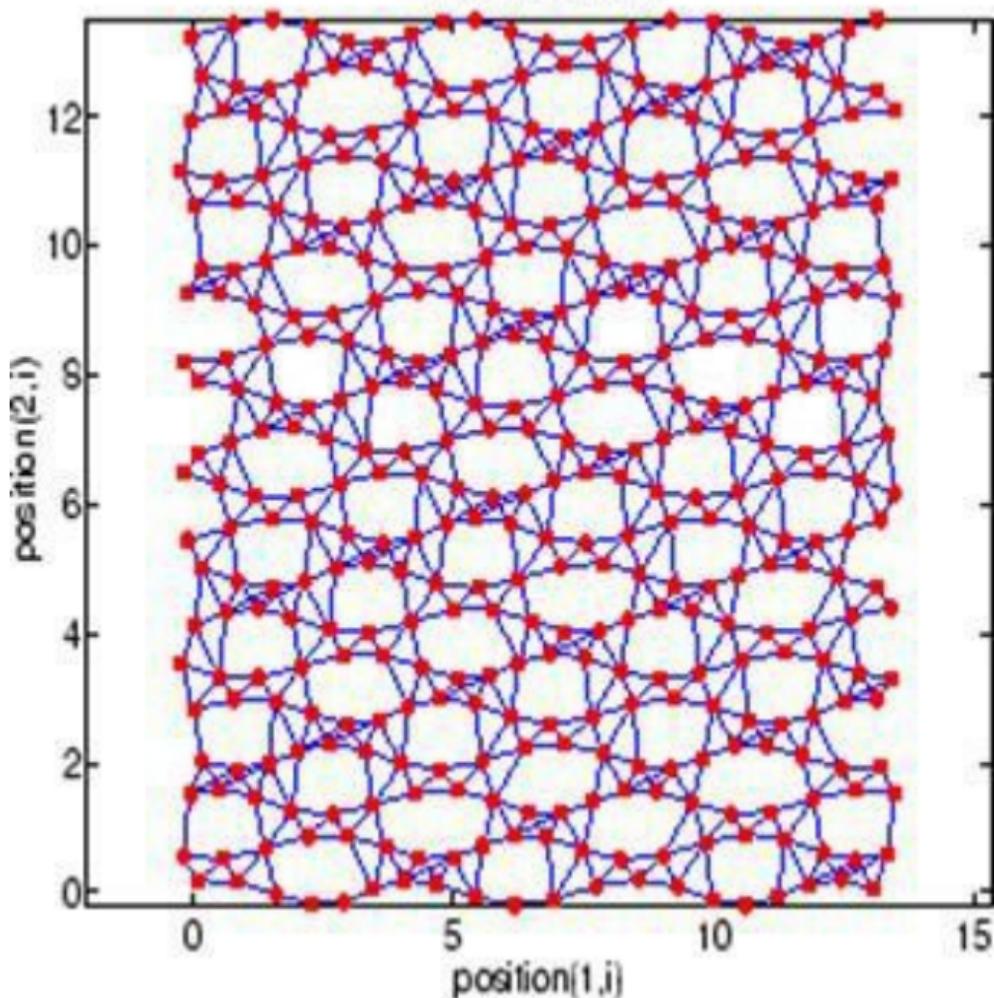
**Side Effects.** Whenever this property is altered, the positions of the layer's neurons (`net.layers{i}.positions`) are updated.

Use [plotsom](#) to plot the positions of the layer neurons. For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [8 10] and the topology function (`net.layers{1}.topologyFcn`) is [randtop](#),

the neuron positions are arranged to resemble the following plot:

```
plotsom(net.layers{1}.positions)
```

Neuron Positions



**net.layers{i}.transferFcn**

This function defines which of the

transfer functions is used to calculate the  $i$ th layer's output, given the layer's net input, during simulation and training.

For a list of functions, type `help nntransfer`.

### **net.layers{i}.transferParam**

This property defines the parameters of the layer's transfer function.

Call `help` on the current transfer function to get a description of what each field means:

`help(net.layers{i}.transferFcn)`

### **net.layers{i}.userdata**

This property provides a place for users to add custom information to the  $i$ th network layer.

## 8.7.3 Outputs

### **net.outputs{i}.name**

This property consists of a string defining the output name. Network creation functions, such as [feedforwardnet](#), define this appropriately. But it can be set to any string as desired.

### **net.outputs{i}.feedbackInput**

If the output implements open-loop feedback (`net.outputs{i}.feedbackMode = 'open'`), then this property indicates the index of the associated feedback input, otherwise it will be an empty matrix.

## `net.outputs{i}.feedbackDelay`

This property defines the timestep difference between this output and network inputs. Input-to-output network delays can be removed and added with [removedelay](#) and [adddelay](#) functions resulting in this property being incremented or decremented respectively. The difference in timing between inputs and outputs is used by [prepares](#) to properly format simulation and training data, and used by [closeloop](#) to add the correct number of delays when closing an open-loop output, and [openloop](#) to remove delays.

when opening a closed loop.

### **net.outputs{i}.feedbackMode**

This property is set to the string 'none' for non-feedback outputs. For feedback outputs it can either be set to 'open' or 'closed'. If it is set to 'open', then the output will be associated with a feedback input, with the property feedbackInput indicating the input's index.

### **net.outputs{i}.processFcns**

This property defines a row cell array of processing function names to be used by the *i*th network output. The processing

functions are applied to target values before the network uses them, and applied in reverse to layer output values before being returned as network output values.

**Side Effects.** When you change this property, you also affect the following settings: the output parameters processParams are modified to the default values of the specified processing functions; processSettings, processedSize and processedRange are defined using the results of applying the process functions and parameters to exampleOutput; the *i*th layer size is

updated to match the processedSize.

For a list of functions, type help nnprocess.

### **net.outputs{i}.processParams**

This property holds a row cell array of processing function parameters to be used by *i*th network output on target values. The processing parameters are applied by the processing functions to input values before the network uses them.

**Side Effects.** Whenever this property is altered, the output processSettings,

`processedSize` and `processedRange` are defined by applying the process functions and parameters to `exampleOutput`. The *i*th layer's size is also updated to match `processedSize`.

### **`net.outputs{i}.processSettings` (read only)**

This property holds a row cell array of processing function settings to be used by *i*th network output. The processing settings are found by applying the processing functions and parameters to `exampleOutput` and then used to provide consistent results to new target values before the network uses them.

The processing settings are also applied in reverse to layer output values before being returned by the network.

### **net.outputs{i}.processedRange (read only)**

This property defines the range of exampleOutput values after they have been processed with processingFcns and processingParamFcns.

### **net.outputs{i}.processedSize (read only)**

This property defines the number of rows in the exampleOutput values after they have been processed.

with processingFcns and processingParams

## **net.outputs{i}.size (read only)**

This property defines the number of elements in the *i*th layer's output. It is always set to the size of the *i*th layer (net.layers{i}.size).

## **net.outputs{i}.userdata**

This property provides a place for users to add custom information to the *i*th layer's output.

## 8.7.4 Biases

### **net.biases{i}.initFcn**

This property defines the weight and bias initialization functions used to set the  $i$ th layer's bias vector ( $\text{net.b}\{i\}$ ) if the network initialization function is [initlay](#) and the  $i$ th layer's initialization function is [initwb](#).

### **net.biases{i}.learn**

This property defines whether the  $i$ th bias vector is to be altered during training and adaption. It can be set to 0 or 1.

It enables or disables the bias's learning during calls to [adapt](#) and train.

## **net.biases{i}.learnFcn**

This property defines which of the learning functions is used to update the *i*th layer's bias vector (`net.b{i}`) during training, if the network training function is [trainb](#), [trainc](#), or [trainr](#), or during adaption, if the network adapt function is [trains](#).

For a list of functions, type `help nnlearn`.

**Side Effects.** Whenever this property is altered, the biases learning parameters

`(net.biases{i}.learnParam)` are set to contain the fields and default values of the new function.

## **net.biases{i}.learnParam**

This property defines the learning parameters and values for the current learning function of the *i*th layer's bias. The fields of this property depend on the current learning function. Call help on the current learning function to get a description of what each field means.

## **net.biases{i}.size (read only)**

This property defines the size of the *i*th layer's bias vector. It is always set to the

size of the  $i$ th layer (`net.layers{i}.size`).

## **net.biases{i}.userdata**

This property provides a place for users to add custom information to the  $i$ th layer's bias.

## 8.7.5 Input Weights

### `net.inputWeights{i,j}.delays`

This property defines a tapped delay line between the  $j$ th input and its weight to the  $i$ th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

**Side Effects.** Whenever this property is altered, the weight's size (`net.inputWeights{i,j}.size`) and the dimensions of its weight matrix (`net.IW{i,j}`) are updated.

## **net.inputWeights{i,j}.initFcn**

This property defines which of the Weight and Bias Initialization Functions is used to initialize the weight matrix ( $\text{net.IW}\{i,j\}$ ) going to the  $i$ th layer from the  $j$ th input, if the network initialization function is [initlay](#), and the  $i$ th layer's initialization function is [initwb](#). This function can be set to the name of any weight initialization function.

## **net.inputWeights{i,j}.initSettings (read only)**

This property is set to values useful for initializing the weight as part of the

configuration process that occurs automatically the first time a network is trained, or when the function [configure](#) is called on a network directly.

### **net.inputWeights{i,j}.learn**

This property defines whether the weight matrix to the  $i$ th layer from the  $j$ th input is to be altered during training and adaption. It can be set to 0 or 1.

### **net.inputWeights{i,j}.learnFcn**

This property defines which of the learning functions is used to update the weight matrix ( $\text{net.IW}\{i,j\}$ ) 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:

$$\text{length}(\text{net.inputWeights}\{i,j\}.\text{delays}) \\ * \text{net.inputs}\{j\}.\text{size}$$

### **net.inputWeights{i,j}.userdata**

This property provides a place for users to add custom information to the  $(i,j)$ th input weight.

### **net.inputWeights{i,j}.weightFcn**

This property defines which of the weight functions is used to apply the  $i$ th

layer's weight from the  $j$ th input to that input. It can be set to the name of any weight function. The weight function is used to transform layer inputs during simulation and training.

For a list of functions, type `help nnweight`.

## **net.inputWeights{i,j}.weightParam**

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

## 8.7.6 Layer Weights

### **net.layerWeights{i,j}.delays**

This property defines a tapped delay line between the  $j$ th layer and its weight to the  $i$ th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

### **net.layerWeights{i,j}.initFcn**

This property defines which of the weight and bias initialization functions is used to initialize the weight matrix (net.LW{i,j}) going to the  $i$ th layer from

the  $j$ th layer, if the network initialization function is [initlay](#), and the  $i$ th layer's initialization function is [initwb](#). This function can be set to the name of any weight initialization function.

## **net.layerWeights{i,j}.initSettings (read only)**

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function [configure](#) is called on a network directly.

## **net.layerWeights{i,j}.learn**

This property defines whether the weight matrix to the  $i$ th layer from the  $j$ th layer is to be altered during training and adaption. It can be set to 0 or 1.

## **net.layerWeights{i,j}.learnFcn**

This property defines which of the learning functions is used to update the weight matrix ( $\text{net.LW}\{i,j\}$ ) going to the  $i$ th layer from the  $j$ th layer during training, if the network training function is [trainb](#), [trainc](#), or [trainr](#), or during adaption, if the network adapt function is [trains](#). It can be set to the name of any

weight learning function.

For a list of functions, type `help nnlearn`.

### **net.layerWeights{i,j}.learnParam**

This property defines the learning parameters fields and values for the current learning function of the  $i$ th layer's weight coming from the  $j$ th layer. The fields of this property depend on the current learning function. Call `help` on the current net input function to get a description of each field.

### **net.layerWeights{i,j}.size (read only)**

This property defines the dimensions of

the  $i$ th layer's weight matrix from the  $j$ th layer. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.LW{i,j}`). The first element is equal to the size of the  $i$ th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the  $j$ th layer.

### **net.layerWeights{i,j}.userdata**

This property provides a place for users to add custom information to the  $(i,j)$ th layer weight.

### **net.layerWeights{i,j}.weightFcn**

This property defines which of the weight functions is used to apply the  $i$ th layer's weight from the  $j$ th layer to that layer's output. It can be set to the name of any weight function. The weight function is used to transform layer inputs when the network is simulated.

For a list of functions, type `help nnweight`.

### **`net.layerWeights{i,j}.weightParam`**

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each

field.

# **Chapter 9**

## **FUNCTIONS FOR PATTERN RECOGNITION AND CLASSIFICATION**

---

# 9.1 INTRODUCTION

The more important functions for pattern recognition and classification are de following:

<a href="#"><u>Autoencoder</u></a>	Autoencoder class
<a href="#"><u>nnstart</u></a>	Neural network getting started GL
<a href="#"><u>view</u></a>	View neural network
<a href="#"><u>trainAutoencoder</u></a>	Train an autoencoder
<a href="#"><u>trainSoftmaxLayer</u></a>	Train a softmax layer for classifica
<a href="#"><u>decode</u></a>	Decode encoded data
<a href="#"><u>encode</u></a>	Encode input data
<a href="#"><u>predict</u></a>	Reconstruct the inputs using train
<a href="#"><u>stack</u></a>	Stack encoders from several autoe
<a href="#"><u>network</u></a>	Convert Autoencoder object into a
<a href="#"><u>patternnet</u></a>	Pattern recognition network
<a href="#"><u>lvqnet</u></a>	Learning vector quantization neur
<a href="#"><u>train</u></a>	Train neural network
<a href="#"><u>trainlm</u></a>	Levenberg-Marquardt backpropaga
<a href="#"><u>trainbr</u></a>	Bayesian regularization backpropo
<a href="#"><u>trainscg</u></a>	Scaled conjugate gradient backpro
<a href="#"><u>trainrp</u></a>	Resilient backpropagation
<a href="#"><u>mse</u></a>	Mean squared normalized error pe
<a href="#"><u>regression</u></a>	Linear regression

<a href="#"><u>roc</u></a>	Receiver operating characteristic
<a href="#"><u>plotconfusion</u></a>	Plot classification confusion matrix
<a href="#"><u>ploterrhist</u></a>	Plot error histogram
<a href="#"><u>plotperform</u></a>	Plot network performance
<a href="#"><u>plotregression</u></a>	Plot linear regression
<a href="#"><u>plotroc</u></a>	Plot receiver operating characteristic
<a href="#"><u>plottrainstate</u></a>	Plot training state values
<a href="#"><u>crossentropy</u></a>	Neural network performance
<a href="#"><u>genFunction</u></a>	Generate MATLAB function for si

## 9.2 VIEW NEURAL NETWORK

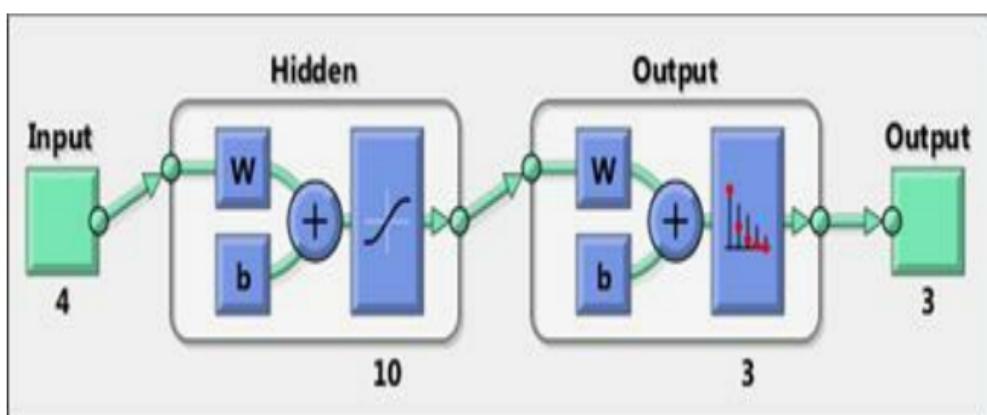
`view(net)` opens a window that shows your neural network (specified in `net`) as a graphical diagram.

This example shows how to view the diagram of a pattern recognition network.

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

```
net = patternnet;
```

```
net =  
configure(net,x,t);  
  
view(net)
```



# **9.3 PATTERN RECOGNITION AND LEARNING VECTOR QUANTIZATION**

## **9.3.1 Pattern recognition network: patternnet**

### **Syntax**

```
patternnet(hiddenSizes, trainF
```

### **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)`  
these arguments,

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

and returns a pattern recognition neural network.

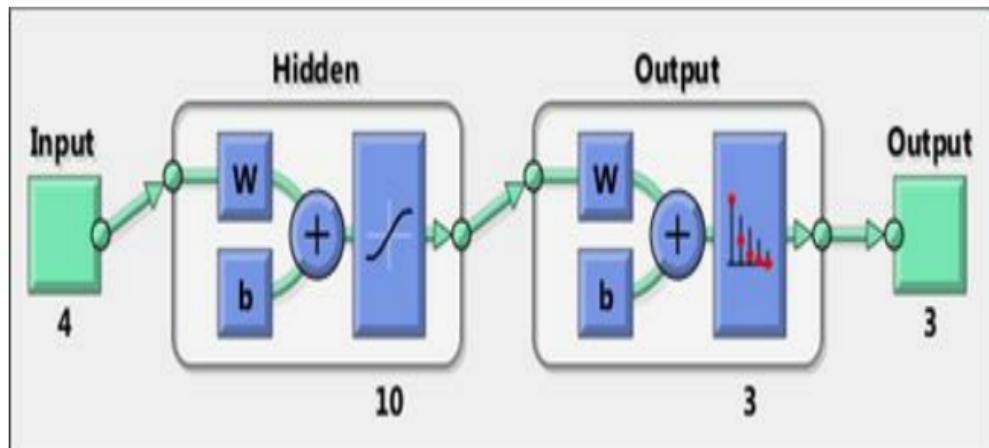
## Example of Pattern Recognition

This example shows how to design a pattern recognition network to classify iris flowers.

```
[x,t] =  
iris_dataset;  
  
net =  
patternnet(10);  
  
net =  
train(net,x,t);  
  
view(net)  
  
y = net(x);
```

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

```
classes =  
vec2ind(y);
```



```
net = fitnet(hiddenSizes)  
net =  
fitnet(hiddenSizes,trainFcn)
```

net = fitnet(hiddenSizes) returns a function fitting neural network with a hidden layer size of hiddenSizes (default=10).

The argument `hiddenSizes` represents the size of the hidden layers in the network, specified as a row vector. The length of the vector determines the number of hidden layers in the network. For example, you can specify a network with 3 hidden layers, where the first hidden layer size is 10, the second is 8, and the third is 5 as follows: [10, 8, 5]

net

=

fitnet(hiddenSizes,trainFcn) returns a function fitting neural network with a hidden layer size of `hiddenSizes` and training function, specified by `trainFcn` (default='trainlm'). The training functions are the following:

Training Function	Algorithm
'trainlm'	Levenberg-Marquardt
'trainbr'	Bayesian Regularization
'trainbfg'	BFGS Quasi-Newton
'trainrp'	Resilient Backpropagation
'trainscg'	Scaled Conjugate Gradient
'traincgb'	Conjugate Gradient with Powell/Beale I
'traincfgf'	Fletcher-Powell Conjugate Gradient
'traincgp'	Polak-Ribière Conjugate Gradient
'trainoss'	One Step Secant
'traingdx'	Variable Learning Rate Gradient Descent
'traingdm'	Gradient Descent with Momentum
'traingd'	Gradient Descent

## 9.3.2 Learning vector quantization neural network: lvqnet

### Syntax

```
lvqnet(hiddenSize, lvqLR, lvqLF)
```

### Description

LVQ (learning vector quantization) neural networks consist of two layers. The first layer maps input vectors into clusters that are found by the network during training. The second layer merges groups of first layer clusters into the classes defined by the target data.

The total number of first layer clusters is determined by the number of hidden neurons. The larger the hidden layer the more clusters the first layer can learn, and the more complex mapping of input to target classes can be made. The relative number of first layer clusters assigned to each target class are determined according to the distribution of target classes at the time of network initialization. This occurs when the network is automatically configured the first time [train](#) is called, or manually configured with the function [configure](#), or manually initialized with the function [init](#) is called.

```
lvqnet(hiddenSize, lvqLR, lvqLF) tal
```

these arguments,

hiddenSize	Size of hidden layer (default = 10)
lvqLR	LVQ learning rate (default = 0.01)
lvqLF	LVQ learning function (default = ' <a href="#">learnlv1</a> )

and returns an LVQ neural network.

The other option for the `lvq` learning function is [learnlv2](#).

## Example: Train a Learning Vector Quantization Network

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

[ x, t ] =

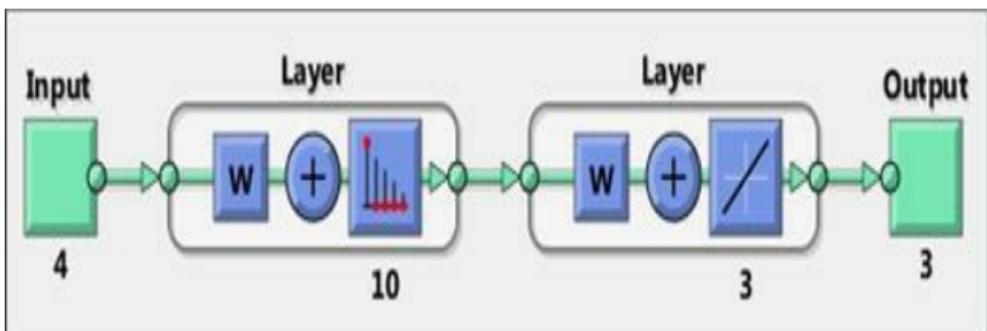
```
iris_dataset;  
  
net = lvqnet(10);  
  
net.trainParam.epoch  
= 50;  
  
net =  
train(net,x,t);  
  
view(net)  
  
y = net(x);
```

```
perf =  
perform(net,y,t)
```

```
classes =  
vec2ind(y);
```

```
perf =
```

0.0489



# 9.4 TRAINING OPTIONS AND NETWORK PERFORMANCE

The following functions are used to training and network performance.

<a href="#"><u>train</u></a>	Train neural network
<a href="#"><u>trainlm</u></a>	Levenberg-Marquardt backpropagation
<a href="#"><u>trainbr</u></a>	Bayesian regularization backpropagation
<a href="#"><u>trainscg</u></a>	Scaled conjugate gradient backpropagation
<a href="#"><u>trainrp</u></a>	Resilient backpropagation
<a href="#"><u>mse</u></a>	Mean squared normalized error performance
<a href="#"><u>regression</u></a>	Linear regression
<a href="#"><u>roc</u></a>	Receiver operating characteristic
<a href="#"><u>plotconfusion</u></a>	Plot classification confusion matrix
<a href="#"><u>ploterrhist</u></a>	Plot error histogram
<a href="#"><u>plotperform</u></a>	Plot network performance
<a href="#"><u>plotregression</u></a>	Plot linear regression
<a href="#"><u>plotroc</u></a>	Plot receiver operating characteristic
<a href="#"><u>plottrainstate</u></a>	Plot training state values
<a href="#"><u>crossentropy</u></a>	Neural network performance
<a href="#"><u>genFunction</u></a>	Generate MATLAB function for simulation



## 9.4.1 Receiver operating characteristic: roc

### Syntax

```
[tpr, fpr, thresholds] =  
roc(targets, outputs)
```

### Description

The *receiver operating characteristic* is a metric used to check the quality of classifiers. For each class of a classifier, `roc` applies threshold values across the interval  $[0, 1]$  to outputs. For each threshold, two values are calculated, the True Positive Ratio

(TPR) and the False Positive Ratio (FPR). For a particular class  $i$ , TPR is the number of outputs whose actual and predicted class is class  $i$ , divided by the number of outputs whose predicted class is class  $i$ . FPR is the number of outputs whose actual class is not class  $i$ , but predicted class is class  $i$ , divided by the number of outputs whose predicted class is not class  $i$ .

You can visualize the results of this function with `plotroc`.

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

targets	S-by-Q matrix, where each column vector contains a single 1 value. The index of the 1 indicates which of S categories that vector represents.
	S-by-Q matrix, where each column contains values in the range [0, 1].

outputs

element in the column indicates which of S categories that vector belongs to. Where values greater or equal to 0.5 indicate class membership, nonmembership.

and returns these values:

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

`roc(targets, outputs)` takes these arguments:

targets	1-by-Q matrix of Boolean values indicating class membership.
outputs	S-by-Q matrix, of values in [0,1] interval, where values greater than 0.5 indicate class membership.

and returns these values:

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

## Examples

```
load iris_dataset  
  
net =  
patternnet(20);  
  
net =  
train(net,irisInputs  
irisOutputs =  
sim(net,irisInputs);  
  
[tpr,fpr,thresholds]
```

=

roc(irisTargets, iris(

## 9.4.2 Plot receiver operating characteristic: plotroc

### Syntax

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

### Description

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

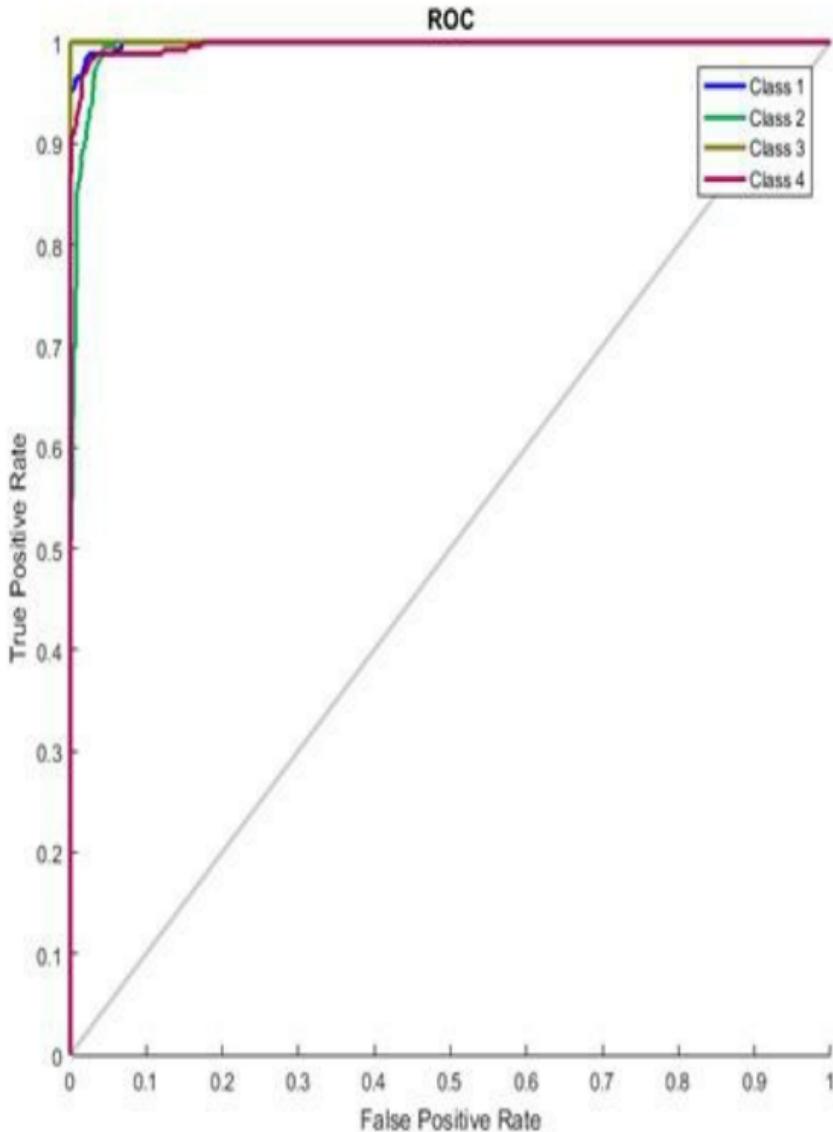
```
plotroc(targets1,outputs2,'name1')
```

multiple plots.

## **Examples: Plot Receiver Operating Characteristic**

```
load  
simplecluster_database  
  
net =  
patternnet(20);  
  
net =
```

```
train(net, simpleclus  
simpleclusterOutputs  
=  
sim(net, simplecluste:  
plotroc(simplecluste:
```





## 9.4.3 Plot classification confusion matrix: plotconfusion

### Syntax

- `plotconfusion(targets, outputs)`
- [example](#)
- `plotconfusion(targets, outputs, title)`
- `plotconfusion(targets1, outputs1, targets2, outputs2)`

### Description

`plotconfusion(targets, outputs)` re1

a confusion matrix plot for the target and output data in targets and outputs, respectively.

On the confusion matrix plot, the rows correspond to the predicted class (Output Class), and the columns show the true class (Target Class). The diagonal cells show for how many (and what percentage) of the examples the trained network correctly estimates the classes of observations. That is, it shows what percentage of the true and predicted classes match. The off diagonal cells show where the classifier has made mistakes. The column on the far right of the plot shows the accuracy for each predicted class, while the row

at the bottom of the plot shows the accuracy for each true class. The cell in the bottom right of the plot shows the overall accuracy.

`plotconfusion(targets, outputs, name)`  
a confusion matrix plot with the title starting with `name`.

`plotconfusion(targets1, outputs1, ..., name1)`  
several confusion plots in one figure, and prefixes the `name` arguments to the titles of the appropriate plots.

## Examples: Plot Confusion Matrix

This example shows how to train a

pattern recognition network and plot its accuracy.

Load the sample data.

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

cancerInputs is a 9x699 matrix defining nine attributes of 699 biopsies. cancerTargets is a 2x966 matrix where each column indicates a correct category with a one in either element 1 (benign) or element 2 (malignant). For more information on this dataset,

type help cancer\_dataset in the command line.

Create a pattern recognition network and train it using the sample data.

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

```
net =  
train(net,x,t);
```

Estimate the cancer status using the trained network, net .

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

Plot the confusion matrix.

```
plotconfusion(t,y)
```

### Confusion Matrix

		1	2	
		1	2	
Output Class	1	446 63.8%	5 0.7%	98.9% 1.1%
	2	12 1.7%	236 33.8%	95.2% 4.8%
	1	97.4% 2.6%	97.9% 2.1%	97.6% 2.4%
	2			
		Target Class		

In this figure, the first two diagonal cells show the number and percentage of correct classifications by the trained network. For example 446 biopsies are correctly classified as benign. This corresponds to 63.8% of all 699 biopsies. Similarly, 236 cases are correctly classified as malignant. This corresponds to 33.8% of all biopsies.

5 of the malignant biopsies are incorrectly classified as benign and this corresponds to 0.7% of all 699 biopsies in the data. Similarly, 12 of the benign biopsies are incorrectly classified as malignant and this corresponds to 1.7% of all data.

Out of 451 benign predictions, 98.9% are correct and 1.1% are wrong. Out of 248 malignant predictions, 95.2% are correct and 4.8% are wrong. Out of 458 benign cases, 97.4% are correctly predicted as benign and 2.6% are predicted as malignant. Out of 241 malignant cases, 97.9% are correctly classified as malignant and 2.1% are classified as benign.

Overall, 97.6% of the predictions are correct and 2.4% are wrong classifications.

## 9.4.4 Neural network performance: crossentropy

### Syntax

- `perf = crossentropy(net,targets,outputs)`
- `perf = crossentropy(__,Name,Value)`

### Description

perf =  
`crossentropy(net,targets,outputs)`  
a network performance given targets

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

The cross-entropy for each pair of output-target elements is calculated as:

$$ce = -t .* \log(y).$$

The aggregate cross-entropy performance is the mean of the individual values:

$$perf = \text{sum}(ce(:)) / \text{numel}(ce).$$

Special case ( $N = 1$ ): If an output consists of only one element, then the

outputs and targets are interpreted as binary encoding. That is, there are two classes with targets of 0 and 1, whereas in 1-of-N encoding, there are two or more classes. The binary cross-entropy expression is:

$$ce = -t \cdot \log(y) - (1-t) \cdot \log(1-y).$$

perf =

`crossentropy(__, Name, Value)` supports customization according to the specified name-value pair arguments.

## Examples: Calculate Network Performance

This example shows how to design a

classification network with cross-entropy and 0.1 regularization, then calculation performance on the whole dataset.

```
[x, t] =
```

```
iris_dataset;
```

```
net =
```

```
patternnet(10);
```

```
net.performParam.reg  
= 0.1;
```

```
net =  
train(net,x,t);
```

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

```
perf =  
crossentropy(net,t,y  
{1}, 'regularization'
```

```
perf =
```

0 . 0278

## 9.4.5 Construct and Train a Function Fitting Network

Load the training data.

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

The 1-by-94 matrix  $x$  contains the input values and the 1-by-94 matrix  $t$  contains the associated target output values.

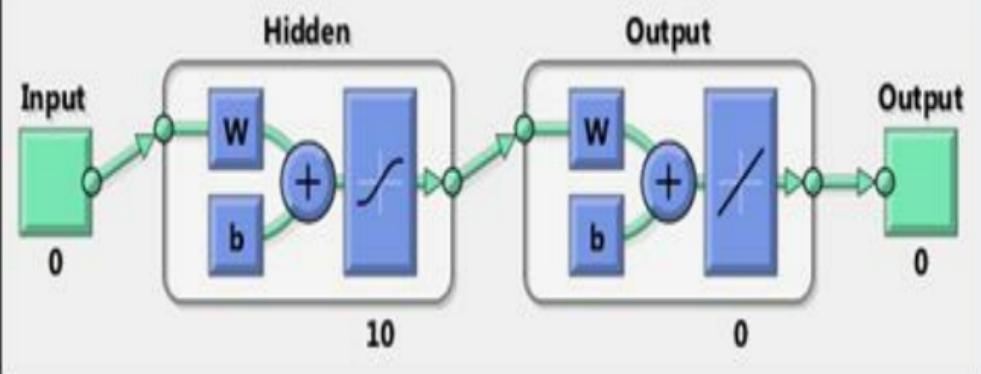
Construct a function fitting neural network with one hidden layer of size

10.

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

View the network.

```
view(net)
```



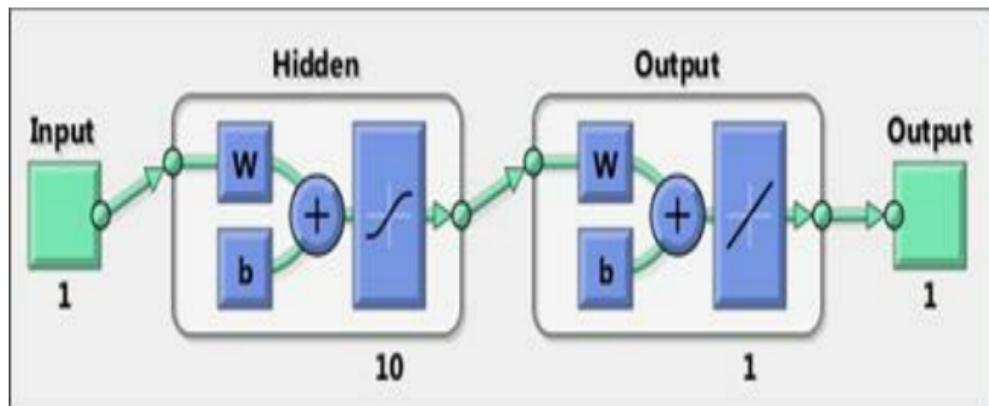
The sizes of the input and output are zero. The software adjusts the sizes of these during training according to the training data.

Train the network `net` using the training data.

```
net =  
train(net,x,t);
```

View the trained network.

```
view(net)
```



You can see that the sizes of the input and output are 1.

Estimate the targets using the trained network.

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

Assess the performance of the trained network. The default performance function is mean squared error.

```
perf =  
perform(net, y, t)
```

```
perf =
```

1.4639e-04

The default training algorithm for a function fitting network is Levenberg-Marquardt ('trainlm'). Use the Bayesian regularization training algorithm and compare the performance results.

```
net =  
fitnet(10, 'trainbr')
```

```
net =  
train(net,x,t);
```

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

```
perf =  
perform(net,y,t)
```

```
perf =
```

3.3416e-10

The Bayesian regularization training algorithm improves the performance of the network in terms of estimating the target values.

## 9.4.6 Create and train Feedforward Neural Network

```
feedforwardnet(hiddenSizes, train)
```

This command construct the feedforward neural network. 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, train)  
these arguments,

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

and returns a feedforward neural network.

This example shows how to use feedforward neural network to solve a simple problem.

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

```
net =  
feedforwardnet(10);
```

```
net =  
train(net, x, t);
```

```
view(net)
```

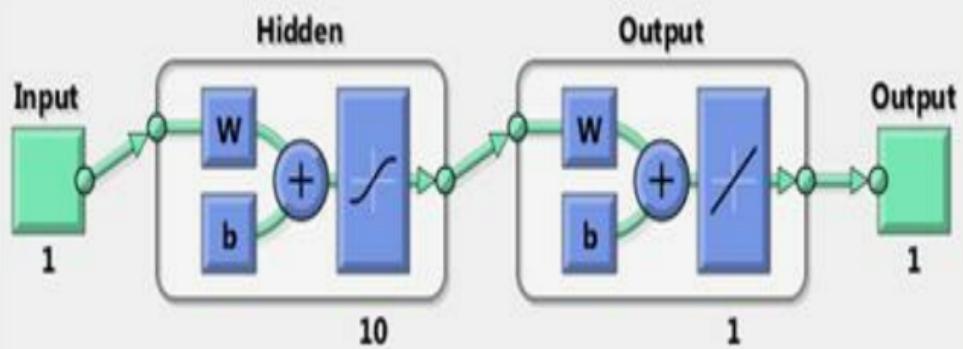
```
y = net(x);
```

```
perf =
```

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

```
perf =
```

```
1.4639e-04
```



## 9.4.7 Create and Train a Cascade Network

```
cascadeforwardnet(hiddenSizes, trainFcn)
```

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)
```

these arguments,

hiddenSizes	Row vector of one or more hidden layer sizes (default = [1 1 1])
trainFcn	Training function (default = 'trainlm')

and returns a new cascade-forward neural network.

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

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

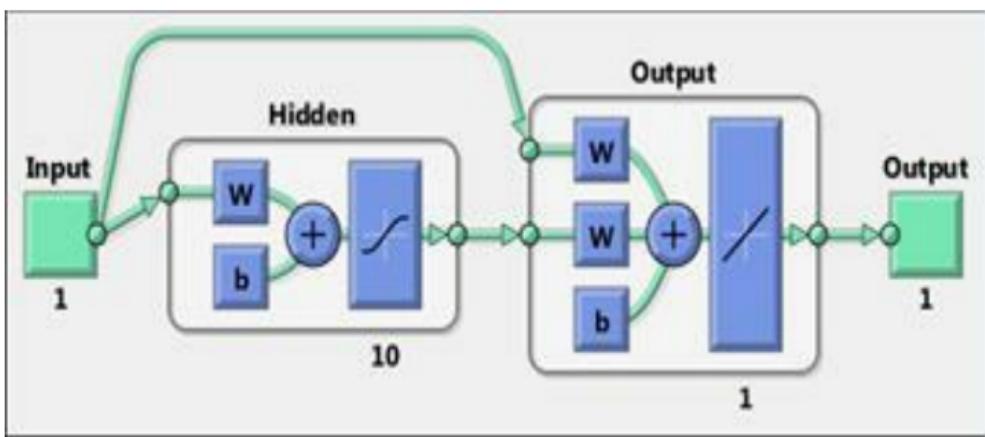
```
view(net)
```

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

```
perf =  
perform(net,y,t)
```

```
perf =
```

```
1.9372e-05
```



# **9.5 NETWORK PERFORMANCE**

In MATLAB `mse` is a network performance function. It measures the network's performance according to the mean of squared errors.

## 9.5.1 Description

`perf = mse(net, t, y, ew)` takes these arguments:

<code>net</code>	Neural network
<code>t</code>	Matrix or cell array of targets
<code>y</code>	Matrix or cell array of outputs
<code>ew</code>	Error weights (optional)

and returns the mean squared error.

This function has two optional parameters, which are associated with networks whose `net.trainFcn` is set to this function:

- '`regularization`' can be set to any value between 0 and 1. The greater the regularization value, the more squared weights and

biases are included in the performance calculation relative to errors. The default is 0, corresponding to no regularization.

'normalization' can be set to 'none' (the default); 'standard', which normalizes errors between -2 and 2, corresponding to normalizing outputs and targets between -1 and 1; and 'percent', which normalizes errors between -1 and 1. This feature is useful for networks with multi-element outputs. It ensures that the relative accuracy of output elements with

differing target value ranges are treated as equally important, instead of prioritizing the relative accuracy of the output element with the largest target value range.

You can create a standard network that uses `mse` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `mse`, set `net.performFcn` to '`'mse'`'. This automatically sets `net.performParam` to a structure with the default optional parameter values.

## 9.5.2 Examples

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

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

```
net.performFcn =  
'mse'; % Redundant,  
MSE is default  
  
net.performParam.reg  
= 0.01;  
  
net =  
train(net,x,t);  
  
y = net(x);  
  
perf =
```

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

Alternately, you can call this function directly.

```
perf =
```

```
mse(net,x,t,'regular')
```

# **9.6 FIT REGRESSION MODEL AND PLOT FITTED VALUES VERSUS TARGETS**

## 9.6.1 Description

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

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

and returns these outputs,

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

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

`plotregression(targets,outputs)` plots the

linear regression of targets relative to outputs.

plotregression(targs1,outs1,'name1',targs)  
generates multiple plots.

## 9.6.2 Examples

Train a feedforward network, then calculate and plot the regression between its targets and outputs.

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

y = net(x);

[r,m,b] =  
regression(t,y)

plotregression(t,y)

r =

1.0000

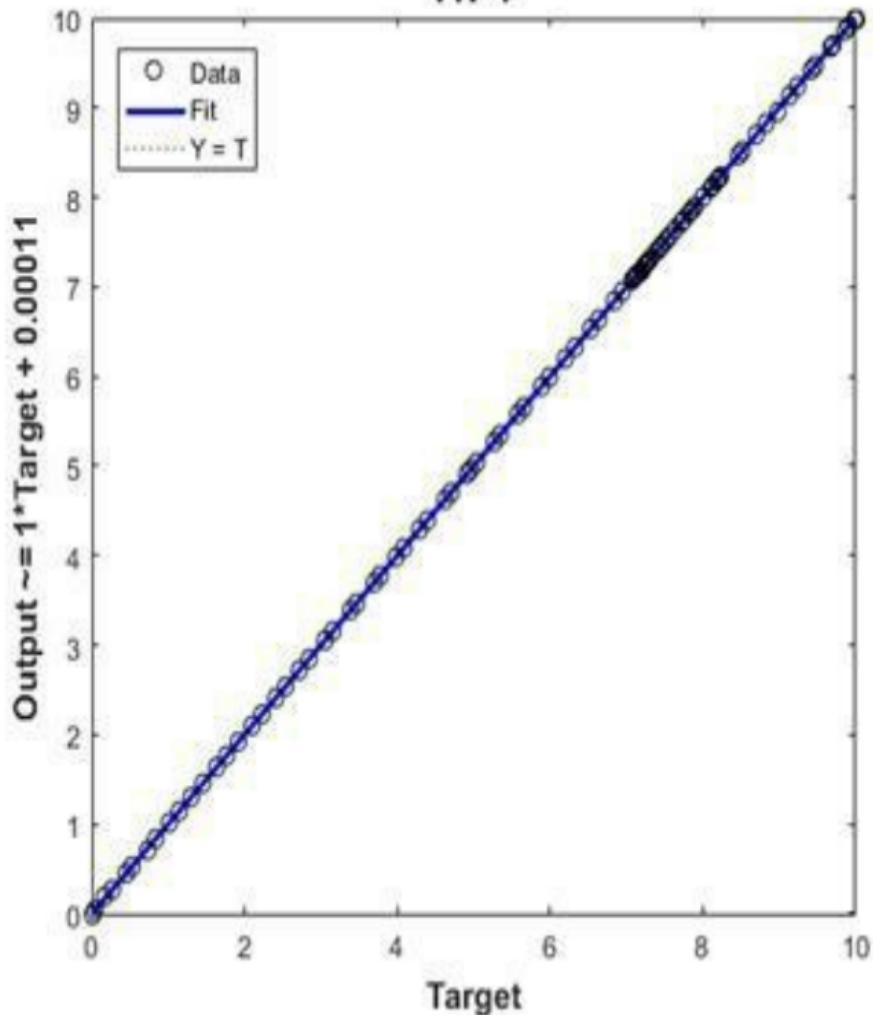
m =

1.0000

b =

1.0878e-04

: R=1

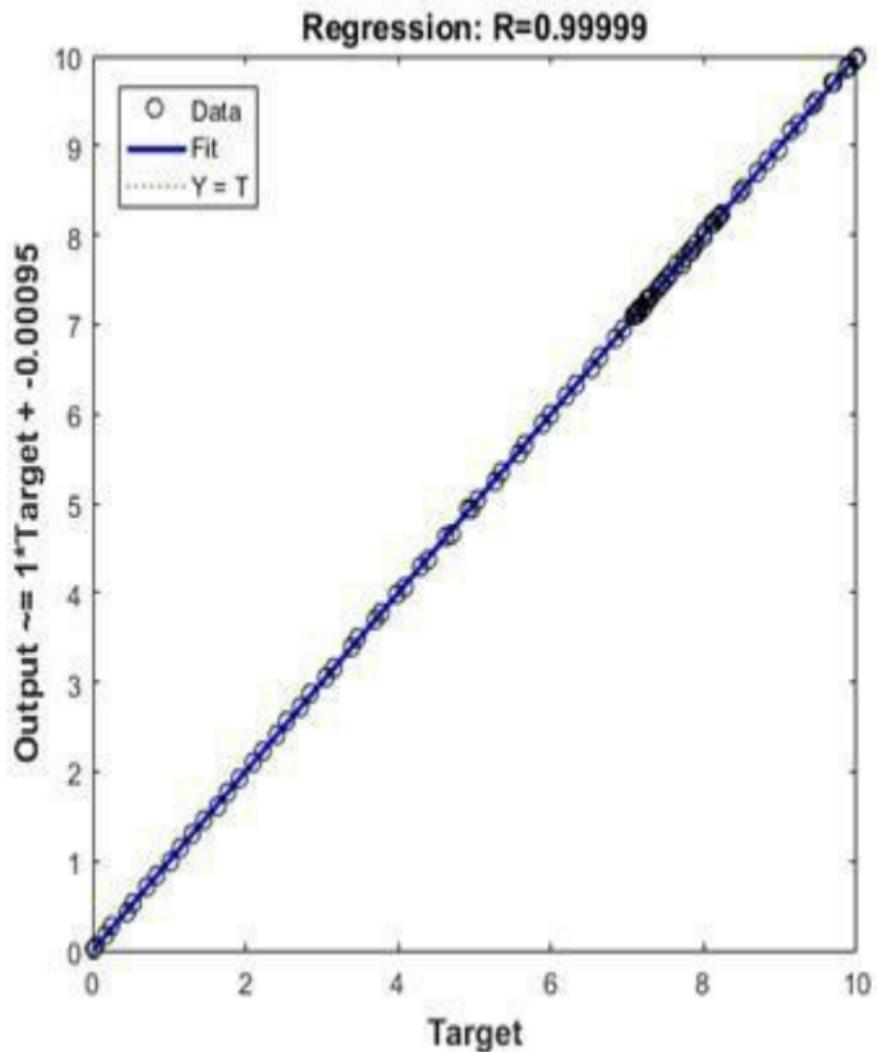


The next example Plot Linear

## Regression

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

plotregression(t,y,'



# **9.7 PLOT OUTPUT AND TARGET VALUES**

## 9.7.1 Description

`plotfit(net,inputs,targets)` plots the output function of a network across the range of the inputs `inputs` and also plots target `targets` and output data points associated with values in `inputs`. Error bars show the difference between outputs and `targets`.

The plot appears only for networks with one input.

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

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

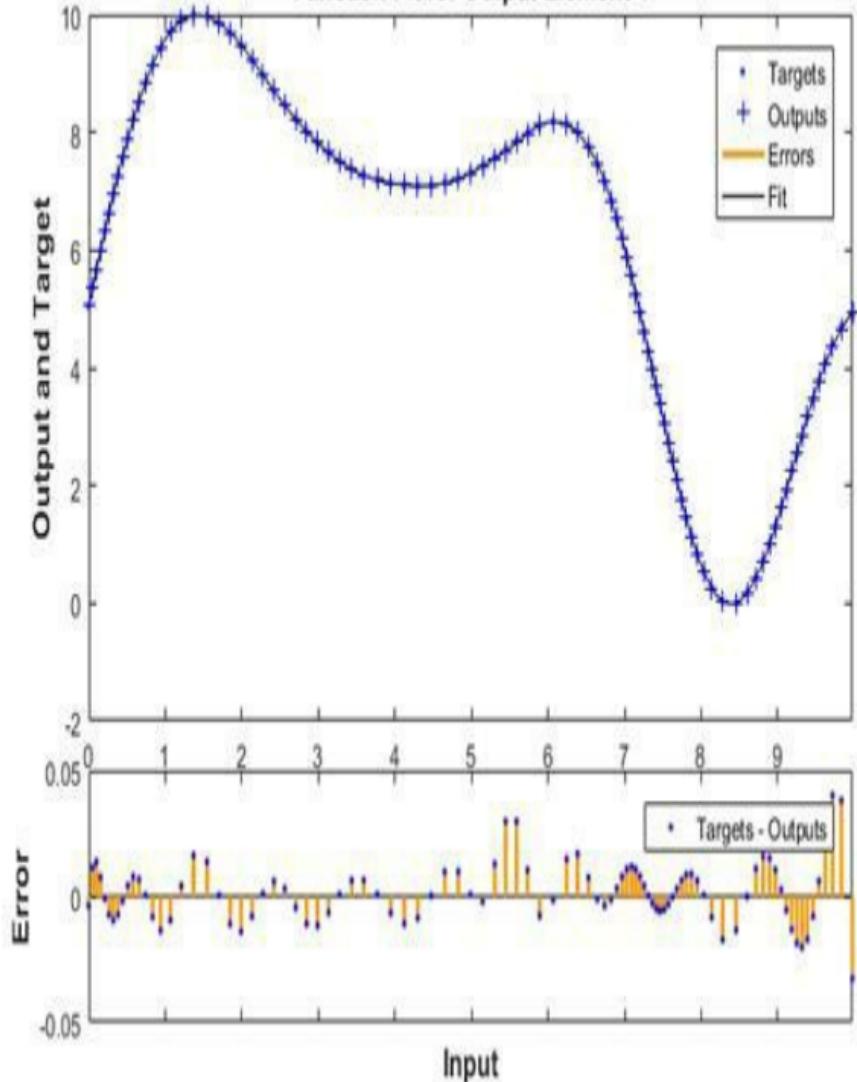
## 9.7.2 Examples

This example shows how to use a feed-forward network to solve a simple fitting problem.

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

plotfit (net, x, t)

### Function Fit for Output Element 1



## 9.8 PLOT TRAINING STATE VALUES

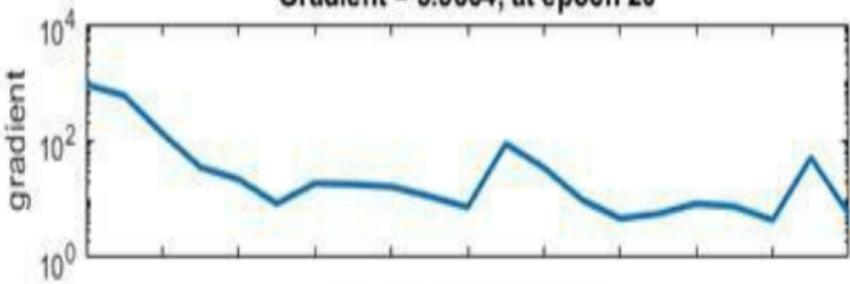
`plottrainstate(tr)` plots the training state from a training record `tr` returned by `train`.

Below is an example:

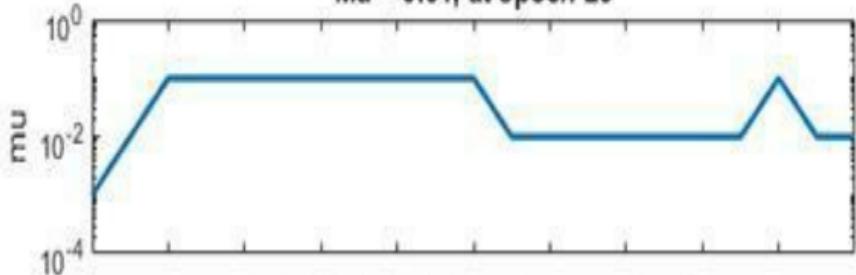
```
[x, t] =  
house_dataset;  
  
net =  
feedforwardnet(10);
```

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

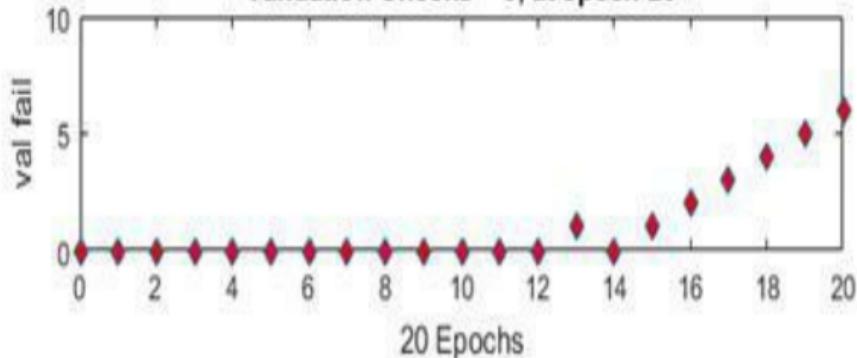
**Gradient = 5.5864, at epoch 20**



**Mu = 0.01, at epoch 20**



**Validation Checks = 6, at epoch 20**



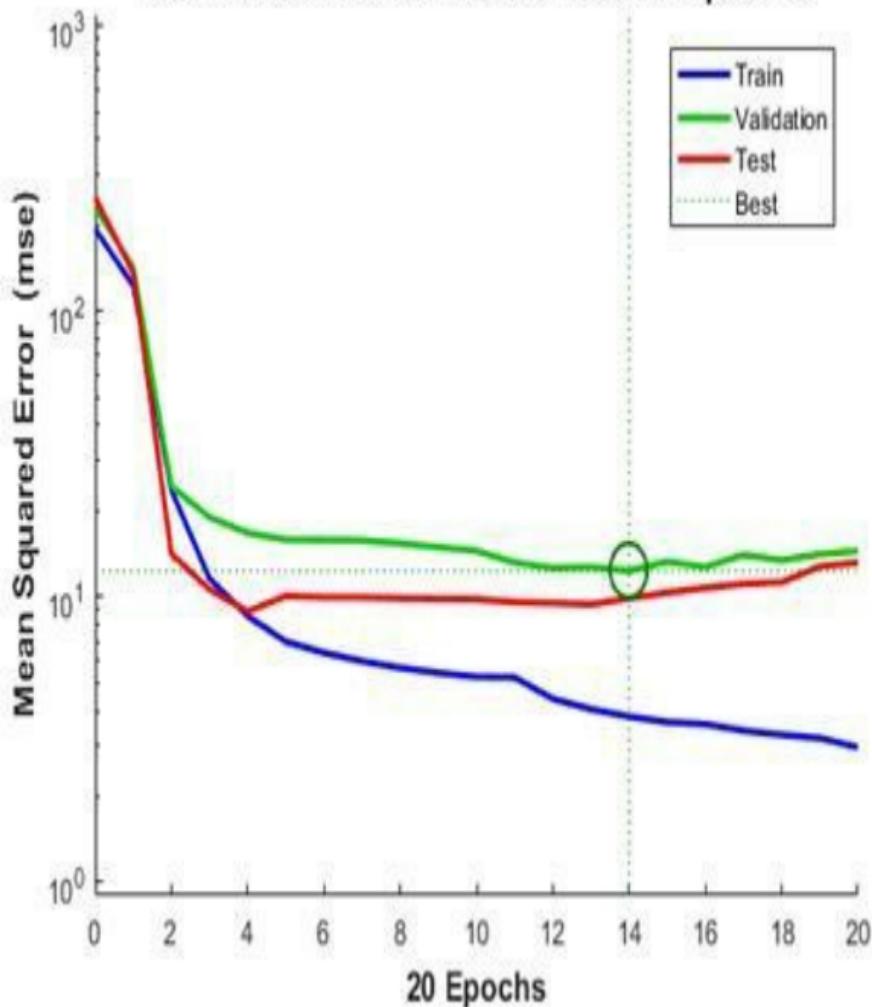
# 9.9 PLOT PERFORMANCES

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

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

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

Best Validation Performance is 12.331 at epoch 14



Generally, the error reduces after more epochs of training, but might start to

increase on the validation data set as the network starts overfitting the training data. In the default setup, the training stops after six consecutive increases in validation error, and the best performance is taken from the epoch with the lowest validation error.

# **9.10 PLOT HISTOGRAM OF ERROR VALUES**

## 9.10.1 Syntax

```
ploterrhist(e)
ploterrhist(e1,'name1',e2,'na
ploterrhist(...,'bins',bins)
```

## 9.10.2 Description

`ploterrhist(e)` plots a histogram of error values `e`.

`ploterrhist(e1, 'name1', e2, 'name2')`  
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.

## 9.10.3 Examples

Here a feedforward network is used to solve a simple fitting problem:

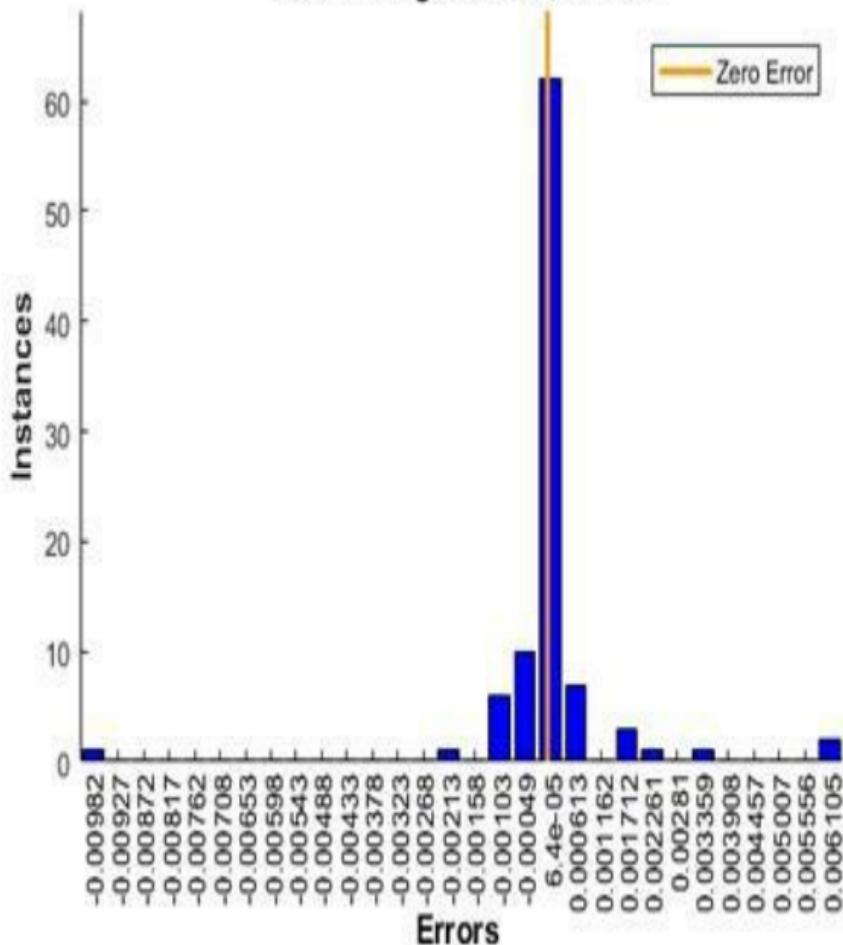
```
[x, t] =  
simplefit_dataset;  
  
net =  
feedforwardnet(20);  
  
net =  
train(net, x, t);
```

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

```
e = t - y;
```

```
ploterrhist(e, 'bins'
```

### Error Histogram with 30 Bins



# **9.11 GENERATE MATLAB FUNCTION FOR SIMULATING NEURAL NETWORK**

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

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

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

## 9.11.1 Create Functions from Static Neural Network

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

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

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

```
houseNet =  
feedforwardnet(10);
```

```
houseNet =  
train(houseNet,x,t);  
  
y = houseNet(x);
```

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

```
genFunction(houseNet)
```

```
y2 = houseFcn(x);
```

```
accuracy2 =  
max(abs(y-y2))
```

```
mcc -W lib:libHouse  
-T link:lib houseFcn
```

Next, generate another version of the MATLAB function that supports only matrix arguments (no cell arrays), and test the function. Use the MATLAB

Coder tool codegen to generate a MEX-function, which is also tested.

```
genFunction(houseNet
```

```
y3 = houseFcn(x);
```

```
accuracy3 =  
max(abs(y-y3))
```

```
x1Type =
```

```
coder.typeof(double([13 Inf])); % Coder  
type of input 1
```

```
codegen houseFcn.m -  
config:mex -o  
houseCodeGen -args  
{x1Type}
```

```
y4 =  
houseCodeGen(x);
```

```
accuracy4 =
```

max (abs (y-y4) )

## 9.11.2 Create Functions from Dynamic Neural Network

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

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

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

```
maglevNet =  
narxnet(1:2, 1:2, 10);
```

```
[X,Xi,Ai,T] =  
preparets(maglevNet,  
{ },t);  
  
maglevNet =  
train(maglevNet,X,T,:  
[y,xf,af] =  
maglevNet(X,Xi,Ai);
```

Next, generate and test a MATLAB

function. Use the function to create a shared/dynamically linked library with mcc.

```
genFunction(maglevNe
```

```
[y2,xf,af] =  
maglevFcn(X,Xi,Ai);
```

```
accuracy2 =  
max(abs(cell2mat(y) -  
cell2mat(y2)))
```

```
mcc -W lib:libMaglev  
-T link:lib  
maglevFcn
```

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

```
genFunction(maglevNe...
```

```
x1 =  
cell2mat(X(1,:)); %  
Convert each input  
to matrix
```

```
x2 =  
cell2mat(X(2,:));  
  
xil =  
cell2mat(Xi(1,:)); %  
Convert each input
```

state to matrix

xi2 =

cell2mat(Xi(2,:));

[y3,xf1,xf2] =

maglevFcn(x1,x2,xi1,:)

accuracy3 =

max(abs(cell2mat(y) -  
y3)))

```
x1Type =  
coder.typeof(double([1 Inf])); % Coder  
type of input 1
```

```
x2Type =  
coder.typeof(double([1 Inf])); % Coder  
type of input 2
```

```
xi1Type =  
coder.typeof(double([
```

```
[1 2]); % Coder type  
of input 1 states
```

```
xi2Type =  
coder.typeof(double([1 2])); % Coder type  
of input 2 states
```

```
codegen maglevFcn.m  
-config:mex -o  
maglevNetCodeGen -  
args {x1Type x2Type  
xi1Type xi2Type}
```

```
[y4,xf1,xf2] =  
maglevNetCodeGen(x1,:  
  
dynamic_codegen_accu:  
=  
max(abs(cell2mat(y) -  
y4)))
```

## **9.12 A COMPLETE EXAMPLE: HOUSE PRICE ESTIMATION**

This example illustrates how a function fitting neural network can estimate median house prices for a neighborhood based on neighborhood demographics.

# 9 .1 2 .1 The Problem: Estimate House Values

In this example we attempt to build a neural network that can estimate the median price of a home in a neighborhood described by thirteen demographic attributes:

- Per capita crime rate per town
- Proportion of residential land zoned for lots over 25,000 sq. ft.
- Proportion of non-retail business acres per town
- 1 if tract bounds Charles river, 0 otherwise
- Nitric oxides concentration

(parts per 10 million)

- Average number of rooms per dwelling
- Proportion of owner-occupied units built prior to 1940
- Weighted distances to five Boston employment centres
- Index of accessibility to radial highways
- Full-value property-tax rate per \$10,000
- Pupil-teacher ratio by town
- $1000(Bk - 0.63)^2$
- Percent lower status of the population

This is an example of a fitting problem, where inputs are matched up to associated target outputs, and we would like to create a neural network which not only estimates the known targets given known inputs, but can generalize to accurately estimate outputs for inputs that were not used to design the solution.

## 9 .1 2 .2 Why Networks?

Neural

Neural networks are very good at function fit problems. A neural network with enough elements (called neurons) can fit any data with arbitrary accuracy. They are particularly well suited for addressing non-linear problems. Given the non-linear nature of real world phenomena, like house valuation, neural networks are a good candidate for solving the problem.

The thirteen neighborhood attributes will act as inputs to a neural network, and the median home price will be the

target.

The network will be designed by using the attributes of neighborhoods whose median house value is already known to train it to produce the target valuations.

## 9.12.3 Preparing the Data

Data for function fitting problems are set up for a neural network by organizing the data into two matrices, the input matrix  $X$  and the target matrix  $T$ .

Each  $i$ th column of the input matrix will have thirteen elements representing a neighborhood whose median house value is already known.

Each corresponding column of the target matrix will have one element, representing the median house price in 1000's of dollars.

Here such a dataset is loaded.

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

We can view the sizes of inputs X and targets T.

Note that both X and T have 506 columns. These represent 506 neighborhood attributes (inputs) and associated median house values (targets).

Input matrix X has thirteen rows, for the thirteen attributes. Target matrix T has only one row, as for each example we only have one desired output, the median house value.

size(x)

size(t)

ans =

13 506

ans =

1 506

## 9.12.4 Fitting a Function with a Neural Network

The next step is to create a neural network that will learn to estimate median house values.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

```
setdemorandstream(49)
```

Two-layer (i.e. one-hidden-layer) feed forward neural networks can fit any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

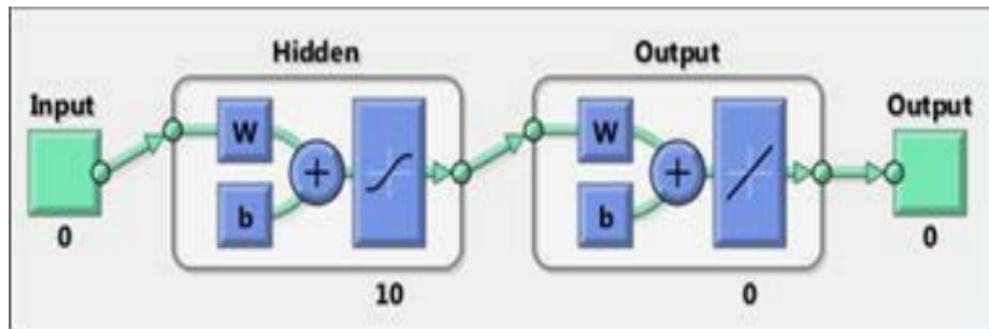
We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target

data. This will happen when the network is trained.

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

```
view(net)
```



Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets.

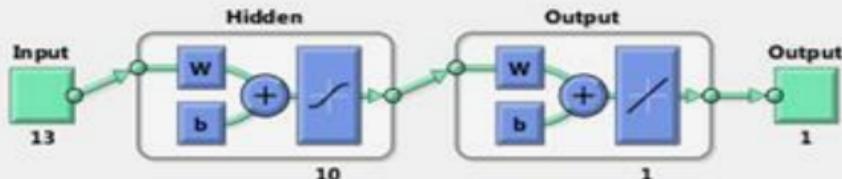
The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] =  
train(net,x,t);  
  
nntraintool
```

## Neural Network



## Algorithms

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

## Progress

Epoch:	0	16 iterations	1000
Time:		0:00:00	
Performance:	2.14e+03	8.69	0.00
Gradient:	5.99e+03	9.30	1.00e-07
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

## Plots



Validation stop.



Stop Training



Cancel

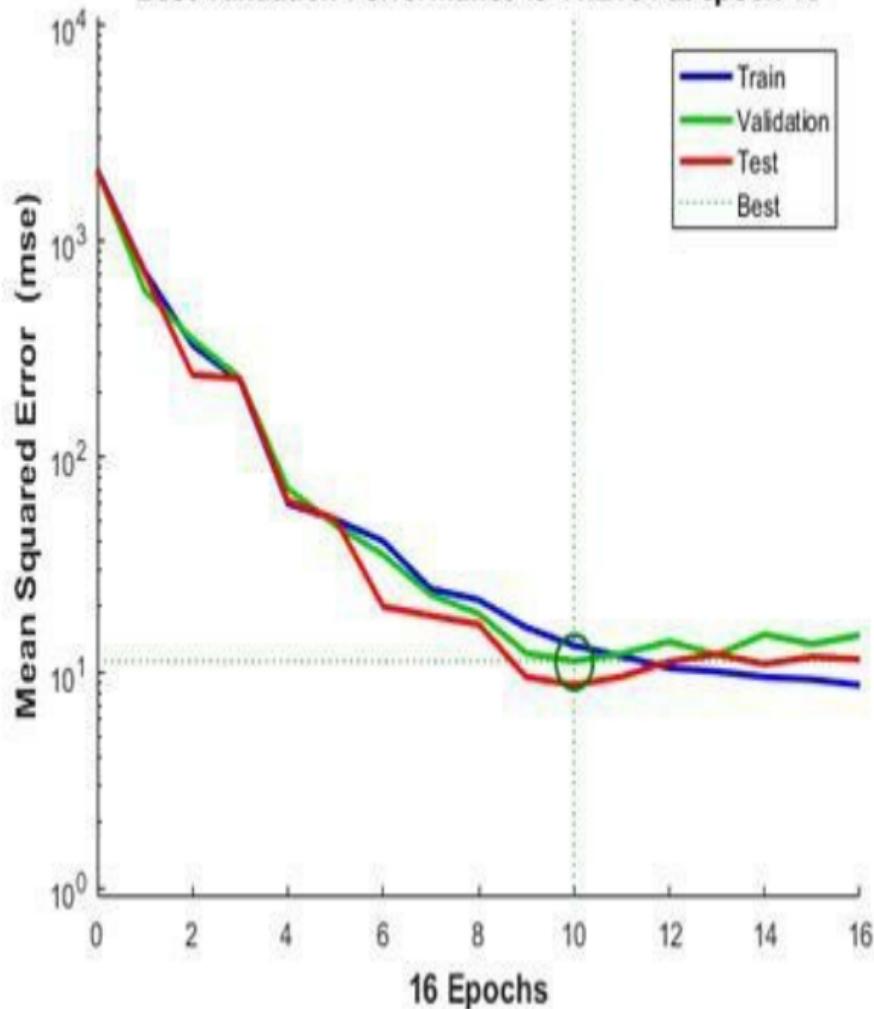
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

```
plotperform(tr)
```

Best Validation Performance is 11.2164 at epoch 10



## 9.1 2.5 Testing the Neural Network

The mean squared error of the trained neural network can now be measured with respect to the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

```
testX =
```

```
x(:, tr.testInd);
```

```
testT =
```

```
t(:,tr.testInd);
```

```
testY = net(testX);
```

```
perf =
```

```
mse(net,testT,testY)
```

```
perf =
```

8 . 6959

Another measure of how well the neural network has fit the data is the regression plot. Here the regression is plotted across all samples.

The regression plot shows the actual network outputs plotted in terms of the associated target values. If the network has learned to fit the data well, the linear fit to this output-target relationship should closely intersect the bottom-left

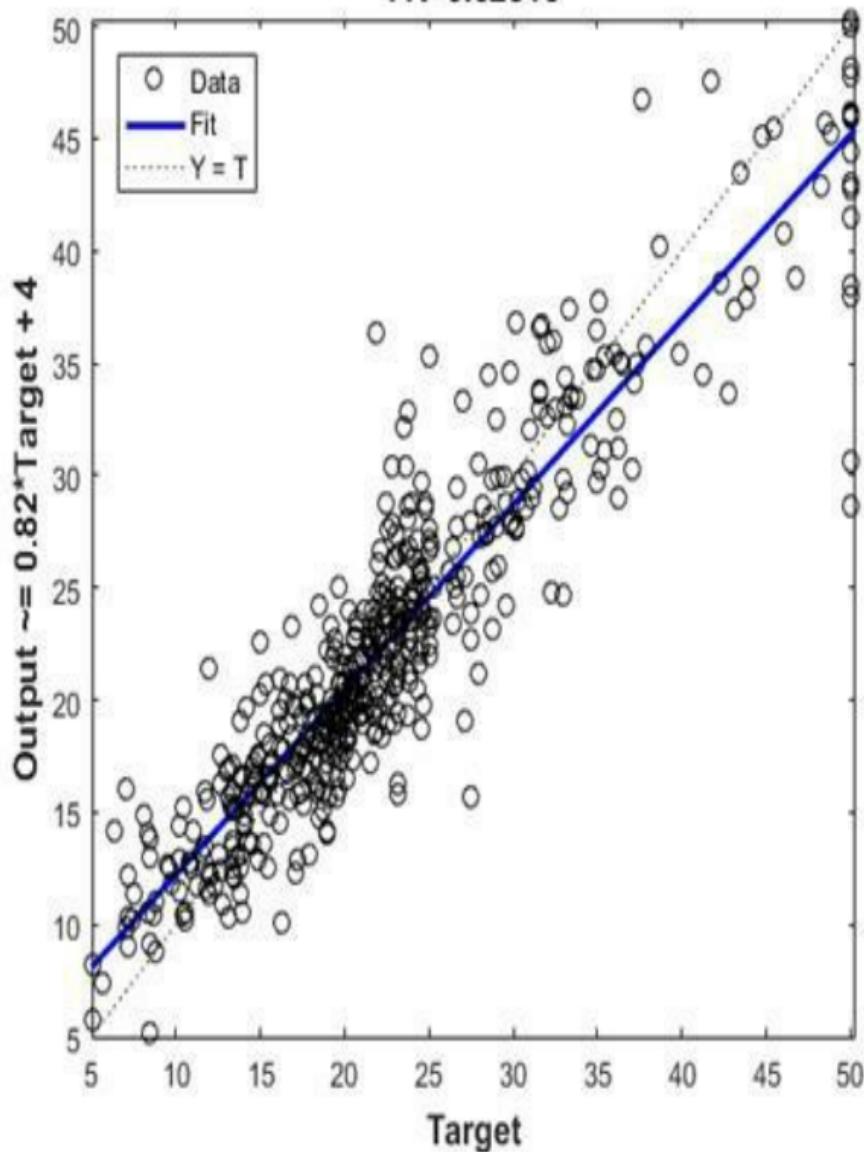
and top-right corners of the plot.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

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

```
plotregression(t,y)
```

: R=0.92516

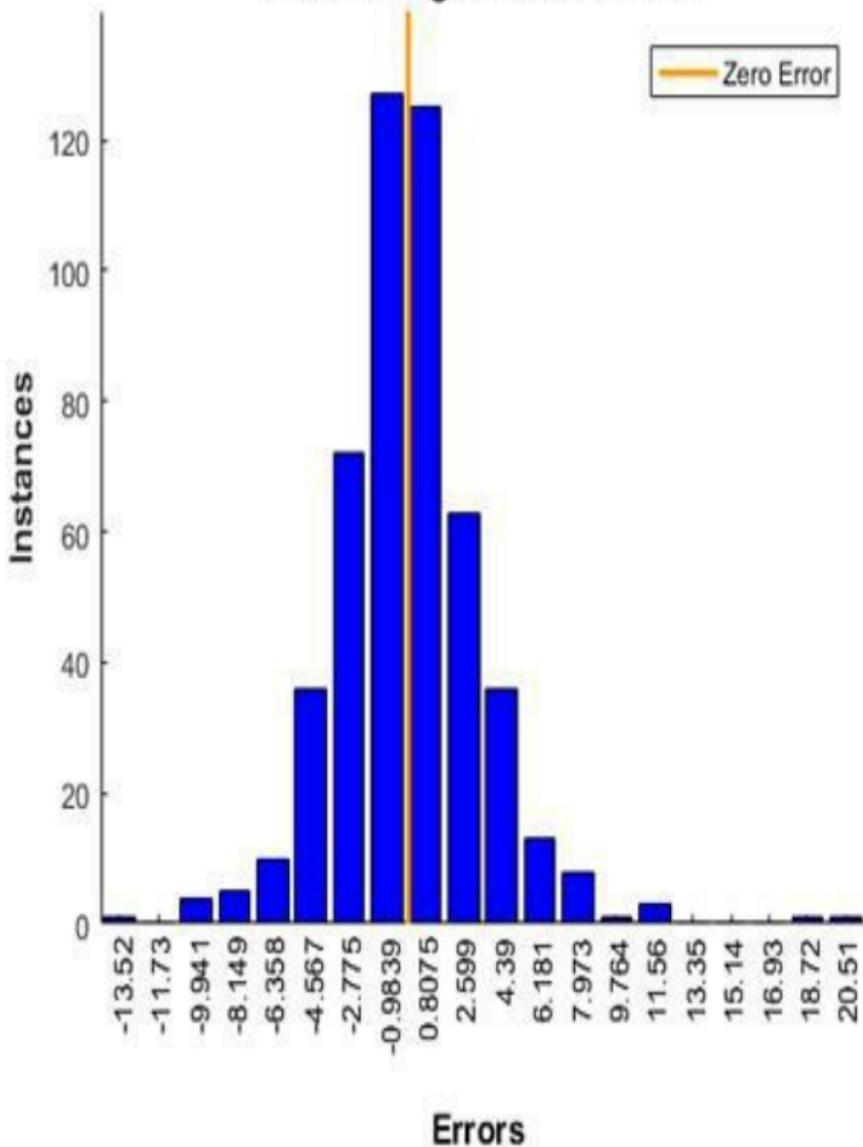


Another third measure of how well the neural network has fit data is the error histogram. This shows how the error sizes are distributed. Typically most errors are near zero, with very few errors far from that.

$$e = t - \hat{y};$$

```
ploterrhist(e)
```

## Error Histogram with 20 Bins



This example illustrated how to design a neural network that estimates the median house value from neighborhood characteristics.

# 9.13 AUTOENCODER CLASS

## Description

An Autoencoder object contains an autoencoder network, which consists of an encoder and a decoder. The encoder maps the input to a hidden representation. The decoder attempts to map this representation back to the original input.

## Construction

```
autoenc =  
trainAutoencoder (X) returns an
```

autoencoder trained using the training data in  $x$ .

`autoenc =`

`trainAutoencoder(x,hiddenSize)` re1  
an autoencoder with the hidden representation size of `hiddenSize`.

`autoenc =`

`trainAutoencoder(__,Name,Value)` f  
any of the above input arguments with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

`x` — training data

matrix | cell array of image data

`Hiddensize` — size of hidden representation of the autoencoder  
10 (default) | positive integer value

# Methods

<a href="#">decode</a>	Decode encoded data
<a href="#">encode</a>	Encode input data
<a href="#">generateFunction</a>	Generate a MATLAB function to run the autoencoder
<a href="#">generateSimulink</a>	Generate a Simulink model for the autoencoder
<a href="#">network</a>	Convert Autoencoder object into network
<a href="#">plotWeights</a>	Plot a visualization of the weights for the encoder
<a href="#">predict</a>	Reconstruct the inputs using trained autoencoder
<a href="#">stack</a>	Stack encoders from several autoencoders together
<a href="#">view</a>	View autoencoder

## 9.13.1 trainAutoencoder

Train an autoencoder

### Syntax

- `autoenc = trainAutoencoder(X)`
- `autoenc = trainAutoencoder(X, hiddenSize)`
- `autoenc = trainAutoencoder(__, Name, Value)`

### Description

[autoenc](#) =

`trainAutoencoder(X)` returns an autoencoder, `autoenc`, trained using the training data in `x`.

autoenc =  
trainAutoencoder(X,hiddenSize) re1  
an autoencoder autoenc, with the  
hidden representation size  
of hiddenSize.

autoenc =  
trainAutoencoder(\_\_, Name,Value) r  
an autoencoder autoenc, for any of the  
above input arguments with additional  
options specified by one or  
more Name,Value pair arguments.

For example, you can specify the  
sparsity proportion or the maximum  
number of training iterations.

**Examples. Train Sparse  
Autoencoder**

Load the sample data.

```
X = abalone_dataset;
```

x is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex (M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera weight, shell weight. For more information on the dataset, type `help abalone_dataset` in the command line.

Train a sparse autoencoder with default

settings.

```
autoenc =
```

```
trainAutoencoder(X);
```

Reconstruct the abalone shell ring data  
using the trained autoencoder.

```
XReconstructed =
```

```
predict(autoenc,X);
```

Compute the mean squared  
reconstruction error.

```
mseError = mse(X -
```

XReconstructed)

mseError =

0.0167

## Train Autoencoder with Specified Options

Load the sample data.

```
X = abalone_dataset;
```

x is an 8-by-4177 matrix defining eight attributes for 4177 different abalone shells: sex (M, F, and I (for infant)), length, diameter, height, whole weight, shucked weight, viscera weight, shell weight. For more information on the dataset, type `help abalone_dataset` in the command line.

Train a sparse autoencoder with hidden size 4, 400 maximum epochs, and linear transfer function for the decoder.

```
autoenc =
```

```
trainAutoencoder(X, 4  
    'DecoderTransferFunc'
```

Reconstruct the abalone shell ring data using the trained autoencoder.

```
XReconstructed =  
predict(autoenc, X);  
Compute the mean squared  
reconstruction error.
```

```
mseError = mse(X-  
XReconstructed)
```

```
mseError =
```

```
0.0056
```

## **Reconstruct Observations Using Sparse Autoencoder**

Generate the training data.

```
rng(0,'twister'); %  
For reproducibility  
  
n = 1000;  
  
r =  
linspace(-10,10,n)';  
  
x = 1 + r*5e-2 +  
sin(r)./r +  
0.2*randn(n,1);
```

Train autoencoder using the training data.

```
hiddenSize = 25;
```

```
autoenc =
```

```
trainAutoencoder(x',
```

```
'EncoderTransferFunc'
```

```
' DecoderTransferFunc  
' L2WeightRegularizat.  
' SparsityRegularizat.  
' SparsityProportion'  
Generate the test data.  
n = 1000;
```

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

Predict the test data using the trained autoencoder, autoenc.

```
xReconstructed =
```

```
predict(autoenc,xtes)
```

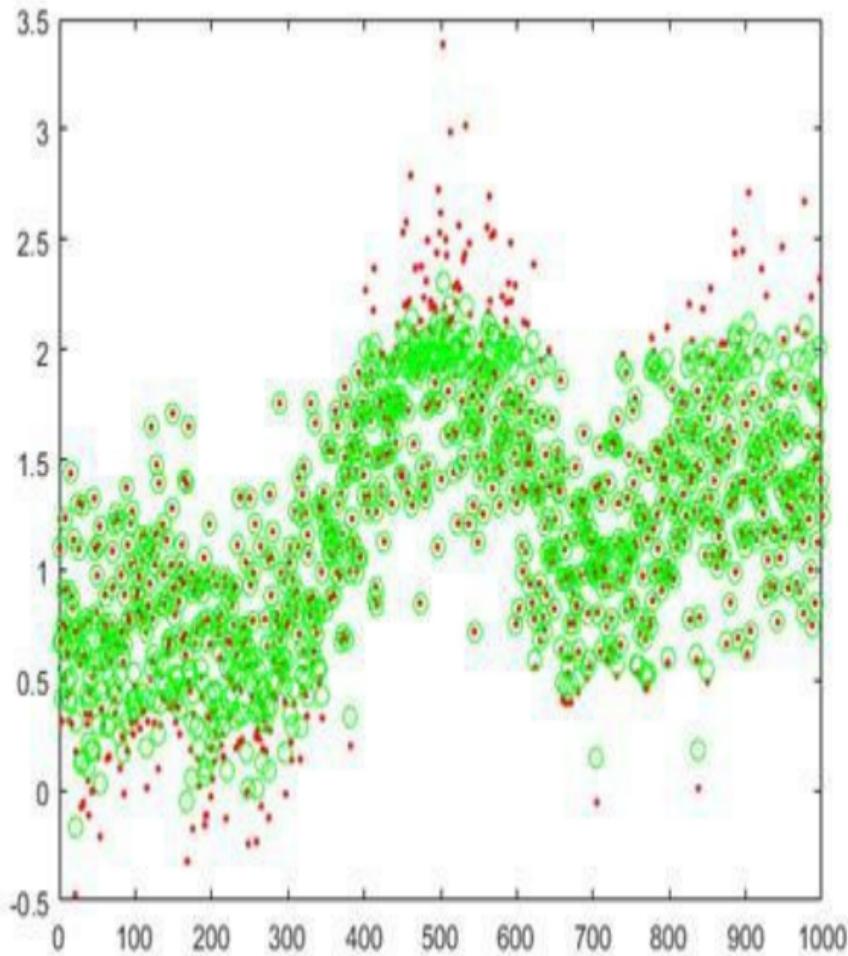
Plot the actual test data and the predictions.

```
figure;
```

```
plot(xtest,'r.');
```

```
hold on
```

```
plot(xReconstructed,
```



**Reconstruct Handwritten  
Digit Images Using Sparse**

# Autoencoder

Load the training data.

```
X =  
digittrain_dataset;
```

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

Train an autoencoder with a hidden layer containing 25 neurons.

```
hiddenSize = 25;
```

```
autoenc =
```

```
trainAutoencoder(X, h
```

```
'L2WeightRegularizat.
```

```
'SparsityRegularizat.
```

'SparsityProportion'

Load the test data.

```
x =  
digittest_dataset;
```

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

Reconstruct the test image data using the

trained autoencoder, autoenc.

```
xReconstructed =  
predict(autoenc,x);  
View the actual test data.  
  
figure;  
  
for i = 1:20  
  
    subplot(4,5,i);
```

```
imshow(X{i});
```

```
end
```

View the reconstructed test data.

```
figure;
```

```
for i = 1:20
```

```
    subplot(4,5,i);
```

```
    imshow(xReconstructed{i});
```

```
end
```

## 9 .1 3 .2 Construct Deep Network Using Autoencoders

Load the sample data.

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

Train an autoencoder with a hidden layer of size 10 and a linear transfer function for the decoder. Set the L2 weight regularizer to 0.001, sparsity regularizer to 4 and sparsity proportion to 0.05.

```
hiddenSize = 10;
```

```
autoenc1 =
```

```
trainAutoencoder(X, h
```

```
'L2WeightRegularizat.
```

```
'SparsityRegularizat.
```

```
'SparsityProportion'  
'DecoderTransferFunc'
```

Extract the features in the hidden layer.

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

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

```
hiddenSize = 10;
```

```
autoenc2 =
```

```
trainAutoencoder (fea
```

```
'L2WeightRegularizat.
```

```
'SparsityRegularizat.
```

```
'SparsityProportion'  
'DecoderTransferFunc'  
  
'ScaleData', false);  
Extract the features in the hidden layer.
```

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

```
softnet =
```

```
trainSoftmaxLayer(fe,
```

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

```
deepnet =
```

```
stack(autoencl, autoe:
```

Train the deep network on the wine data.

```
deepnet =
```

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

Estimate the wine types using the deep  
network, deepnet.

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

Plot the confusion matrix.

```
plotconfusion(T,wine_
```

### Confusion Matrix

		Target Class			
		1	2	3	4
Output Class	1	59 33.1%	0 0.0%	0 0.0%	100% 0.0%
	2	0 0.0%	71 39.9%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	48 27.0%	100% 0.0%	
4	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	

### 9.13.3 decode

Decode encoded data

## Syntax

- Y = decode(autoenc, Z)

## Description

Y = decode(autoenc, Z) returns the decoded data Y, using the autoencoder object `autoenc`.

Trained autoencoder, returned by the `trainAutoencoder` function as an object of the `Autoencoder` class.

Data encoded by `autoenc`, specified as a matrix. Each column of  $z$  represents an encoded sample (observation).

Decoded data, returned as a matrix or a cell array of image data.

If the autoencoder `autoenc` was trained on a cell array of image data, then  $y$  is also a cell array of images.

If the autoencoder `autoenc` was trained on a matrix, then  $y$  is also a matrix, where each column of  $y$  corresponds to one sample or observation.

## Example: Decode Encoded

# Data For New Images

Load the training data.

```
X =
```

```
digitTrainCellArrayD
```

x is a 1-by-5003 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder using the training data with a hidden size of 15.

```
hiddenSize = 15;
```

```
autoenc =
```

```
trainAutoencoder(X, h)
```

Extract the encoded data for new images  
using the autoencoder.

```
Xnew =
```

```
digitTestCellArrayData
```

```
features =  
encode(autoenc, Xnew)
```

Decode the encoded data from the autoencoder.

```
Y =  
decode(autoenc, features)  
Y is a 1-by-4997 cell array, where each  
cell contains a 28-by-28 matrix  
representing a synthetic image of a  
handwritten digit.
```

## 9.13.4 encode

Encode input data

### Syntax

```
Z = encode(autoenc, Xnew)
```

### Description

`Z = encode(autoenc, Xnew)` returns the encoded data, z, for the input data `Xnew`, using the autoencoder, `autoenc`.

### Example. Encode Decoded

# Data for New Images

Load the sample data.

```
X =
```

```
digitTrainCellArrayD
```

x is a 1-by-5003 cell array, where each cell contains a 28-by-28 matrix representing a synthetic image of a handwritten digit.

Train an autoencoder with a hidden size of 50 using the training data.

```
autoenc =  
trainAutoencoder(X, 5)  
Encode decoded data for new image  
data.
```

```
Xnew =  
digitTestCellArrayDa...  
  
Z =  
encode(autoenc, Xnew)
```

$x_{\text{new}}$  is a 1-by-4997 cell array.  $z$  is a 50-by-4997 matrix, where each column represents the image data of one handwritten digit in the new data  $x_{\text{new}}$ .

## 9.13.5 predict

Reconstruct the inputs using trained autoencoder

### Syntax

- Y = predict(autoenc, X)

### Description

Y = predict(autoenc, X) returns the predictions [Y](#) for the input data X, using the autoencoder [autoenc](#). The result Y is a reconstruction of X.

# **Examples: Predict Continuous Measurements Using Trained Autoencoder**

Load the training data.

```
X = iris_dataset;
```

The training data contains measurements on four attributes of iris flowers: Sepal length, sepal width, petal length, petal width.

Train an autoencoder on the training data using the positive saturating linear transfer function in the encoder and linear transfer function in the decoder.

```
autoenc =  
trainAutoencoder(X, ':  
'satlin', 'DecoderTra:
```

Reconstruct the measurements using the trained network, autoenc.

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

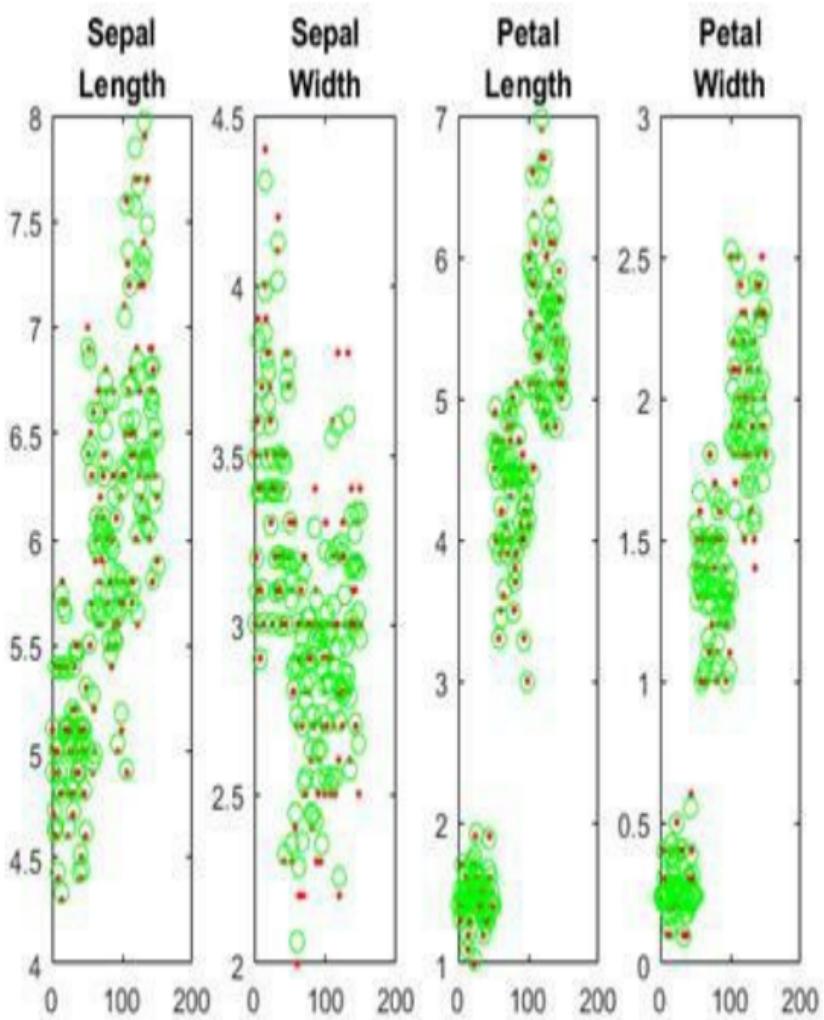
Plot the predicted measurement values along with the actual values in the training dataset.

```
for i = 1:4
```

```
h(i) =  
subplot(1,4,i);
```

```
plot(X(i,:),'r.') ;  
hold on  
  
plot(xReconstructed( . . . ) , . . . ) ;  
hold off;  
end  
  
title(h(1) ,  
{'Sepal';'Length'}) ;
```

```
title(h(2),  
{'Sepal';'Width'}) ;  
  
title(h(3),  
{'Petal';'Length'}) ;  
  
title(h(4),  
{'Petal';'Width'}) ;
```



The red dots represent the training data  
and the green circles represent the

reconstructed data.

## 9.13.6 stack

Stack encoders from several autoencoders together

## Syntax

- `stackednet = stack(autoenc1, autoenc2, ...)`
- `stackednet = stack(autoenc1, autoenc2, ..., ne`

## Description

`stackednet` =  
`stack(autoenc1, autoenc2, ...)` `return`

a network object created by stacking the encoders of the autoencoders, [autoenc1](#), autoenc2, and so on.

```
stackednet =  
stack(autoenc1, autoenc2, ..., net1)  
a network object created by stacking the  
encoders of the autoencoders and the  
network object net1.
```

The autoencoders and the network object can be stacked only if their dimensions match.

## Tips

- The size of the hidden

representation of one autoencoder must match the input size of the next autoencoder or network in the stack.

The first input argument of the stacked network is the input argument of the first autoencoder. The output argument from the encoder of the first autoencoder is the input of the second autoencoder in the stacked network. The output argument from the encoder of the second autoencoder is the input argument to the third autoencoder in the stacked network, and so on.

The stacked network object `stacknet` inherits its training parameters from the final input argument [`net1`](#).

# Examples. Create a Stacked Network

Load the training data.

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

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

to 4 and sparsity proportion to 0.05.

```
hiddenSize = 5;
```

```
autoenc =
```

```
trainAutoencoder(X,  
hiddenSize, ...)
```

```
'L2WeightRegularizat.  
0.001, ...
```

```
'SparsityRegularizat.  
4, ...  
  
'SparsityProportion'  
0.05, ...  
  
'DecoderTransferFunc'
```

Extract the features in the hidden layer.

```
features =  
encode(autoenc,X);  
Train a softmax layer for classification  
using the features.
```

```
softnet =  
trainSoftmaxLayer(fe...
```

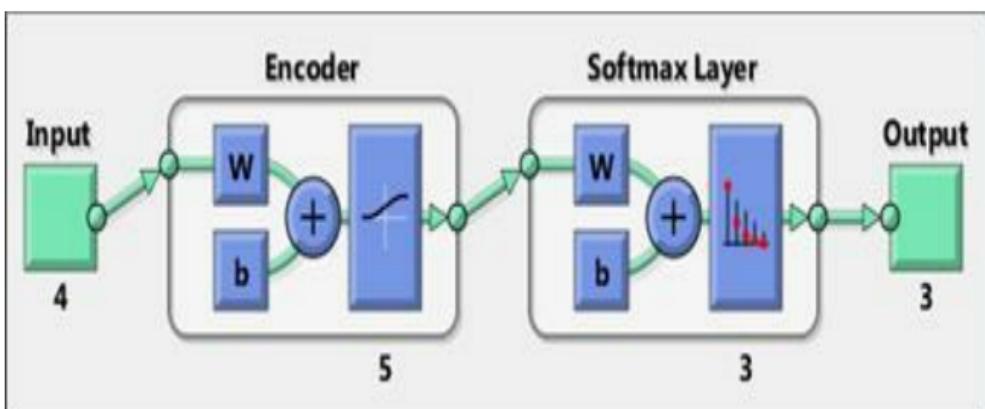
Stack the encoder and the softmax layer  
to form a deep network.

```
stackednet =
```

stack (autoenc, softne

View the stacked network.

view (stackednet) ;



# **Chapter 10**

## **MULTILAYER NEURAL NETWORK**

---

# **10.1 CREATE, CONFIGURE, AND INITIALIZE MULTILAYER NEURAL NETWORKS**

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

configured with the [configure](#) command.

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

```
load house_dataset
```

Loading this file creates two variables. The input matrix `houseInputs` consists of 506 column vectors of 13 real estate variables for 506 different houses. The

target matrix `houseTargets` consists of the corresponding 506 relative valuations.

The next step is to create the network. The following call

to [`feedforwardnet`](#) creates a two-layer network with 10 neurons in the hidden layer. (During the configuration step, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.)

```
net =  
feedforwardnet;
```

```
net =  
configure(net, houseI:
```

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

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

The [configure](#) command configures the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you might want to reinitialize the weights, or to perform a custom initialization.

[Initializing Weights](#)

[\(init\)](#) 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.

## 10.1.1 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.

# **10.2 FUNCTIONS FOR CREATE, CONFIGURE, AND INITIALIZE MULTILAYER NEURAL NETWORKS**

## 10.2.1 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);
```

## 10.2.2 feedforwardnet

Feedforward neural network

# Syntax

```
feedforwardnet(hiddenSizes, train)
```

## 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, train)`  
these arguments,

`hiddenSizes` Row vector of one or more hidden layer sizes (default = 10)

`trainFcn`  
`(default = 'trainlm')`

Training function

and returns a feedforward neural network.

# Examples

## Feedforward Neural Network

---

This example shows how to use feedforward neural network to solve a simple problem.

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

```
train(net,x,t);
```

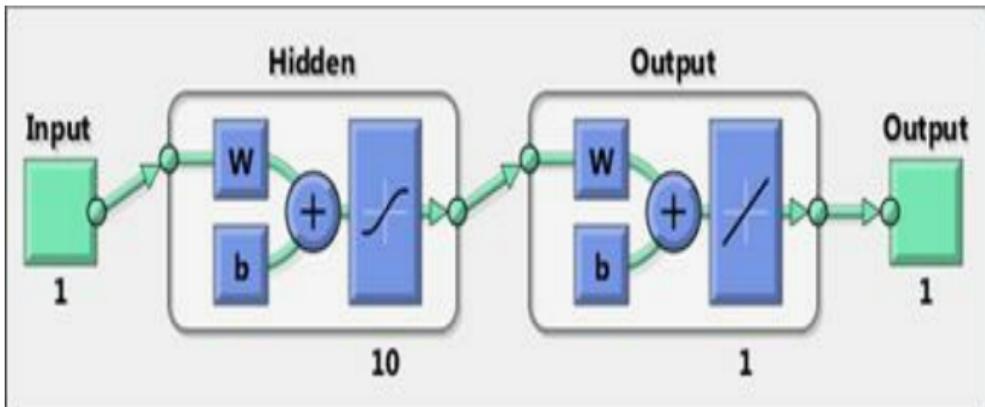
```
view(net)
```

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

```
perf =  
perform(net,y,t)
```

```
perf =
```

$1.4639e-04$



## 10.2.3 configure

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

# Syntax

---

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

# Description

---

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

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

sizes, ranges, processing settings, and initialization settings properties manually.

`net = configure(net, x, t)` takes input data `x` and target data `t`, and configures the network's inputs and outputs to match.

`net = configure(net, x)` configures only inputs.

`net =`  
`configure(net, 'inputs', x, i)` config the inputs specified with the index vector `i`. If `i` is not specified all inputs are configured.

`net =`  
`configure(net, 'outputs', t, i)` confi

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)
```

## 10.2.4 init

Initialize neural network

# Syntax

---

net = init(net)

# To Get Help

---

Type help network/init.

# Description

---

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

# Examples

---

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

`x = [0 1 0 1; 0 0 1 1];`

`t = [0 0 0 1];`

`net = perceptron;`

`net = configure(net,x,t);`

`net.iw{1,1}`

`net.b{1}`

Training the perceptron alters its weight and bias values.

```
net = train(net,x,t);
```

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

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

init reinitializes those weight and bias values.

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

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

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

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

# Algorithms

---

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

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

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

Other networks have net.layers{i}.initFcn set to 'initwb', which initializes each weight and bias

with its own initialization function. The most common weight and bias initialization function is rands, which generates random values between  $-1$  and  $1$ .

## 10.2.5 train

Train neural network

### Syntax

---

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

[net,\_\_] = train(\_\_,'useParallel',\_\_)

[net,\_\_] = train(\_\_,'useGPU',\_\_)

[net,\_\_] =

train(\_\_,'showResources',\_\_)

[net,\_\_] =

train(Xcomposite,Tcomposite,\_\_)

[net,\_\_] = train(Xgpu,Tgpu,\_\_)

net =

train(\_\_,'CheckpointFile','path/name')

### Description

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

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

net Network

X Network inputs

T Network targets (default = zeros)

Xi Initial input delay conditions  
(default = zeros)

Ai Initial layer delay conditions  
(default = zeros)

EW Error weights

and returns

net Newly trained network

tr Training record (epoch and perf)

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

train arguments can have two formats: matrices, for static problems and networks with single inputs and outputs, and cell arrays for multiple timesteps and networks with multiple inputs and outputs.

The matrix format is as follows:

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

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

X Ni-by-TS cell array      Each element  $X\{i,ts\}$  is an  $R_i$ -by-Q matrix.

T No-by-TS cell array      Each element  $T\{i,ts\}$  is a  $U_i$ -by-Q matrix.

$X_i$  Ni-by-ID cell array      Each element  $X_i\{i,k\}$  is an  $R_i$ -by-Q matrix.

$A_i$  Nl-by-LD cell array      Each element  $A_i\{i,k\}$  is an  $S_i$ -by-Q matrix.

EW No-by-TS cell array      Each element  $EW\{i,ts\}$  is a  $U_i$ -by-Q matrix

where

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

The columns of  $X_i$  and  $A_i$  are ordered from the oldest delay condition to the most recent:

$$X_i\{i,k\} = \text{Input } i \text{ at time ts} = k$$

- ID

$$A_i\{i,k\} = \text{Layer output } i \text{ at time } ts = k - LD$$

The error weights EW can also have a size of 1 in place of all or any of No, TS, Ui or Q. In that case, EW is automatically dimension extended to match the targets T. This allows for conveniently weighting the importance in any dimension (such as per sample) while having equal importance across another (such as time, with TS=1). If all dimensions are 1, for instance if EW = {1}, then all target values are treated with the same importance. That is the default value of EW.

The matrix format can be used if only

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

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

X	(sum of $R_i$ )-by-Q matrix
T	(sum of $U_i$ )-by-Q matrix
$X_i$	(sum of $R_i$ )-by-( $ID^*Q$ ) matrix
$A_i$	(sum of $S_i$ )-by-( $LD^*Q$ ) matrix
EW	(sum of $U_i$ )-by-Q matrix

As noted above, the error weights EW can be of the same dimensions as the targets T, or have some dimensions set to 1. For instance

if EW is 1-by-Q, then target samples will have different importances, but each element in a sample will have the same importance. If EW is (sum of  $U_i$ )-by-Q, then each output element has a different importance, with all samples treated with the same importance.

The training record TR is a structure whose fields depend on the network training function (net.NET.trainFcn). It can include fields such as:

- Training, data division, and performance functions and parameters
- Data division indices for training, validation and test sets

- Data division masks for training validation and test sets
- Number of epochs (num\_epochs) and the best epoch (best\_epoch).
- A list of training state names (states).
- Fields for each state name recording its value throughout training
- Performances of the best network  
(best\_perf, best\_vperf, best\_tperf)

[net,\_\_] =  
train(\_\_,'useParallel',\_\_), [net,\_\_] =  
train(\_\_,'useGPU',\_\_), or [net,\_\_] =

`train(__,'showResources',__)` accepts optional name/value pair arguments to control how calculations are performed. Two of these options allow training to happen faster or on larger datasets using parallel workers or GPU devices if Parallel Computing Toolbox is available. These are the optional name/value pairs:

'useParallel','no'	Calculations occur on normal MATLAB thread. This is the default 'useParallel' setting.
'useParallel','yes'	Calculations occur on parallel workers if a parallel pool is open. Otherwise calculations occur on the normal MATLAB thread.
'useGPU','no'	Calculations occur on the CPU. This is the default 'useGPU' setting.
'useGPU','yes'	Calculations occur on the current <code>gpuDevice</code> if it is a supported GPU (See Parallel Computing Toolbox for GPU requirements.) If the current <code>gpuDevice</code> is not supported, calculations remain on the CPU. If 'useParallel' is also 'yes' and a parallel pool is open, then each worker with a unique

	GPU uses that GPU, other workers run calculations on their respective CPU cores.
'useGPU','only'	If no parallel pool is open, then this setting is the same as 'yes'. If a parallel pool is open then only workers with unique GPUs are used. However, if a parallel pool is open, but no supported GPUs are available, then calculations revert to performing on all worker CPUs.
'showResources','no'	Do not display computing resources used at the command line. This is the default setting.
'showResources','yes'	Show at the command line a summary of the computing resources actually used. The actual resources may differ from the requested resources, if parallel or GPU computing is requested but a parallel pool is not open or a supported GPU is not available. When parallel workers are used, each worker's computation mode is described, including workers in the pool that are not used.
'reduction',N	For most neural networks, the default CPU training computation mode is a compiled MEX algorithm. However, for large networks the calculations might occur with a MATLAB calculation mode. This can be confirmed using 'showResources'. If MATLAB is being used and memory is an issue, setting the reduction option to a value N greater than 1, reduces much of the temporary storage required to train by a factor of N, in exchange for longer training times.

[net,\_\_] =  
train(Xcomposite,Tcomposite,\_\_)

takes Composite data and returns Composite results. If Composite data is used, then 'useParallel' is automatically set to 'yes'.

[net,\_\_] = train(Xgpu,Tgpu,\_\_)

takes gpuArray data and returns gpuArray results. If gpuArray data is used, then 'useGPU' is automatically set to 'yes'.

net =  
train(\_\_,'CheckpointFile','path/name','Ch')  
saves intermediate values of the neural network and training record during training to the specified file. This protects training results from power

failures, computer lock ups, Ctrl+C, or any other event that halts the training process before train returns normally.

The value for 'CheckpointFile' can be set to a filename to save in the current working folder, to a file path in another folder, or to an empty string to disable checkpoint saves (the default value).

The optional parameter 'CheckpointDelay' limits how often saves happen. Limiting the frequency of checkpoints can improve efficiency by keeping the amount of time saving checkpoints low compared to the time spent in calculations. It has a default value of 60, which means that checkpoint saves do not happen more

than once per minute. Set the value of 'CheckpointDelay' to 0 if you want checkpoint saves to occur only once every epoch.

**Note** Any NaN values in the inputs X or the targets T, are treated as missing data. If a column of X or T contains at least one NaN, that column is not used for training, testing, or validation.

## Examples

## Train and Plot Networks

---

Here input x and targets t define a simple function that you can plot:

```
x = [0 1 2 3 4 5 6 7 8];  
t = [0 0.84 0.91 0.14 -0.77 -0.96 -0.28  
0.66 0.99];  
plot(x,t,'o')
```

Here feedforwardnet creates a two-layer feed-forward network. The network has one hidden layer with ten neurons.

```
net = feedforwardnet(10);  
net = configure(net,x,t);  
y1 = net(x)  
plot(x,t,'o',x,y1,'x')
```

The network is trained and then resimulated.

```
net = train(net,x,t);  
y2 = net(x)  
plot(x,t,'o',x,y1,'x',x,y2,'*')
```

# Train NARX Time Series Network

---

This example trains an open-loop nonlinear-autoregressive network with external input, to model a levitated magnet system defined by a control current  $x$  and the magnet's vertical position response  $t$ , then simulates the network. The

function [prepares](#) prepares the data before training and simulation. It creates the open-loop network's combined inputs  $xo$ , which contains both the external input  $x$  and previous values of position  $t$ . It also prepares the delay states  $xi$ .

```
[x,t] = maglev_dataset;  
net = narxnet(10);  
[xo,xi,~,to] = preparets(net,x,{},t);  
net = train(net,xo,to,xi);  
y = net(xo,xi)
```

This same system can also be simulated in closed-loop form.

```
netc = closeloop(net);  
view(netc)  
[xc,xi,ai,tc] = preparets(netc,x,{},t);  
yc = netc(xc,xi,ai);
```

## Train a Network in Parallel on a Parallel Pool

---

Parallel Computing Toolbox™ allows Neural Network Toolbox™ to simulate

and train networks faster and on larger datasets than can fit on one PC. Parallel training is currently supported for backpropagation training only, not for self-organizing maps.

Here training and simulation happens across parallel MATLAB workers.

parpool

```
[X,T] = vinyl_dataset;  
net = feedforwardnet(10);  
net =  
train(net,X,T,'useParallel','yes','showRes'  
Y = net(X);
```

Use Composite values to distribute the data manually, and get back the results as a Composite value. If the data is loaded

as it is distributed then while each piece of the dataset must fit in RAM, the entire dataset is limited only by the total RAM of all the workers.

```
[X,T] = vinyl_dataset;
Q = size(X,2);
Xc = Composite;
Tc = Composite;
numWorkers = numel(Xc);
ind = [0 ceil((1:4)*(Q/4))];
for i=1:numWorkers
    indi = (ind(i)+1):ind(i+1);
    Xc{i} = X(:,indi);
    Tc{i} = T(:,indi);
end
net = feedforwardnet;
net = configure(net,X,T);
```

```
net = train(net,Xc,Tc);
```

```
Yc = net(Xc);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's inputs. This normally happens automatically when `train` is called, but when providing composite data this step must be done manually with non-Composite data.

## Train a Network on GPUs

---

Networks can be trained using the current GPU device, if it is supported by Parallel Computing Toolbox. GPU training is currently supported for backpropagation training only, not for

self-organizing maps.

```
[X,T] = vinyl_dataset;  
net = feedforwardnet(10);  
net = train(net,X,T,'useGPU','yes');  
y = net(X);
```

To put the data on a GPU manually:

```
[X,T] = vinyl_dataset;  
Xgpu = gpuArray(X);  
Tgpu = gpuArray(T);  
net = configure(net,X,T);  
net = train(net,Xgpu,Tgpu);  
Ygpu = net(Xgpu);  
Y = gather(Ygpu);
```

Note in the example above the function `configure` was used to set the dimensions and processing settings of the network's

inputs. This normally happens automatically when train is called, but when providing gpuArray data this step must be done manually with non-gpuArray data.

To run in parallel, with workers each assigned to a different unique GPU, with extra workers running on CPU:

```
net =  
train(net,X,T,'useParallel','yes','useGPU',  
y = net(X);
```

Using only workers with unique GPUs might result in higher speed, as CPU workers might not keep up.

```
net =  
train(net,X,T,'useParallel','yes','useGPU',
```

$Y = \text{net}(X);$

## Train Network Using Checkpoint Saves

---

Here a network is trained with checkpoints saved at a rate no greater than once every two minutes.

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

```
net = fitnet([60 30]);
```

```
net =
```

```
train(net,x,t,'CheckpointFile','MyCheckpc
```

After a computer failure, the latest network can be recovered and used to continue training from the point of failure. The checkpoint file includes a structure variable `checkpoint`, which

includes the network, training record, filename, time, and number.

```
[x,t] = vinyl_dataset;  
load MyCheckpoint  
net = checkpoint.net;  
net =  
train(net,x,t,'CheckpointFile','MyCheckpc
```

Another use for the checkpoint feature is when you stop a parallel training session (started with the 'UseParallel' parameter) even though the Neural Network Training Tool is not available during parallel training. In this case, set a 'CheckpointFile', use Ctrl+C to stop training any time, then load your checkpoint file to get the network and training record.

# Algorithms

---

train calls the function indicated by net.trainFcn, using the training parameter values indicated by net.trainParam.

Typically one epoch of training is defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.

Training occurs until a maximum number of epochs occurs, the performance goal is met, or any other stopping condition of

the function `net.trainFcn` occurs.

Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly for each epoch from concurrent input vectors (or sequences). [competlayer](#) returns networks that use [trainru](#), a training function that does this.

## 10.2.6 trainlm

Levenberg-Marquardt backpropagation

### Syntax

---

```
net.trainFcn = 'trainlm'  
[net,tr] = train(net,...)
```

### Description

---

trainlm is a network training function that updates weight and bias values according to Levenberg-Marquardt optimization.

trainlm is often the fastest backpropagation algorithm in the toolbox, and is highly recommended as a

first-choice supervised algorithm, although it does require more memory than other algorithms.

`net.trainFcn = 'trainlm'` sets the network `trainFcn` property.

`[net, tr] = train(net, ...)` trains the network with `trainlm`.

Training occurs according to `trainlm` training parameters, shown here with their default values:

`net.trainParam.epochs`

1000 Maximum number of epochs to train

`net.trainParam.goal`

0 Performance goal

`net.trainParam.max_fail`

6 Maximum validation failures

`net.trainParam.min_grad`

1e-7 Minimum performance gradient  
net.trainParam.mu  
0.001 Initial mu  
net.trainParam.mu\_dec  
0.1 mu decrease factor  
net.trainParam.mu\_inc  
10 mu increase factor  
net.trainParam.mu\_max  
1e10 Maximum mu  
net.trainParam.show  
25 Epochs between displays (NaN for no displays)  
net.trainParam.showCommandLine  
false Generate command-line output  
net.trainParam.showWindow  
true Show training GUI  
net.trainParam.time  
inf Maximum time to train in seconds

Validation vectors are used to stop

training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

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

## Network Use

---

You can create a standard network that uses `trainlm` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be

trained with `trainlm`,

1.

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

2.

Set `net.trainParam` properties  
to desired values.

In either case, calling `train` with the resulting network trains the network with `trainlm`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

## Examples

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

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

# Definitions

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$\mathbf{H} = \mathbf{J}^T \mathbf{J}$$

and the gradient can be computed as

$$\mathbf{g} = \mathbf{J}^T \mathbf{e}$$

where  $\mathbf{J}$  is the Jacobian matrix that contains first derivatives of the network

errors with respect to the weights and biases, and  $e$  is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique (see [[HaMe94](#)]) that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T \mathbf{J} + \mu \mathbf{I}]^{-1} \mathbf{J}^T \mathbf{e}$$

When the scalar  $\mu$  is zero, this is just Newton's method, using the approximate Hessian matrix. When  $\mu$  is large, this becomes gradient descent

with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus,  $\mu$  is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

The original description of the Levenberg-Marquardt algorithm is given in [[Marq63](#)]. The application of Levenberg-Marquardt to neural network training is described in [[HaMe94](#)] and

starting on page 12-19 of [[HDB96](#)]. This algorithm appears to be the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has an efficient implementation in MATLAB<sup>®</sup> software, because the solution of the matrix equation is a built-in function, so its attributes become even more pronounced in a MATLAB environment.

Try the *Neural Network Design* demonstration `nnd12m` [[HDB96](#)] for an illustration of the performance of the batch Levenberg-Marquardt algorithm.

# Limitations

---

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

# Algorithms

---

`trainlm` supports training with validation and test vectors if the network's `NET.divideFcn` property is set to a data division function. Validation vectors are used to stop training early if the network performance on the validation vectors fails to improve or remains the same for `max_fail` epochs

in a row. Test vectors are used as a further check that the network is generalizing well, but do not have any effect on training.

`trainlm` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate the Jacobian  $j_x$  of performance `perf` with respect to the weight and bias variables  $x$ . Each variable is adjusted according to Levenberg-Marquardt,

$$j_j = j_X * j_X$$

$$j_e = jX * E$$

$$dX = -(jj + I^*mu) \setminus j_e$$

where  $E$  is all errors and  $I$  is the identity matrix.

The adaptive value  $\mu$  is increased by  $\mu_{inc}$  until the change above results in a reduced performance value. The change is then made to the network and  $\mu$  is decreased by  $\mu_{dec}$ .

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is

exceeded.

Performance is minimized to the goal.

The performance gradient falls below `min_grad`.

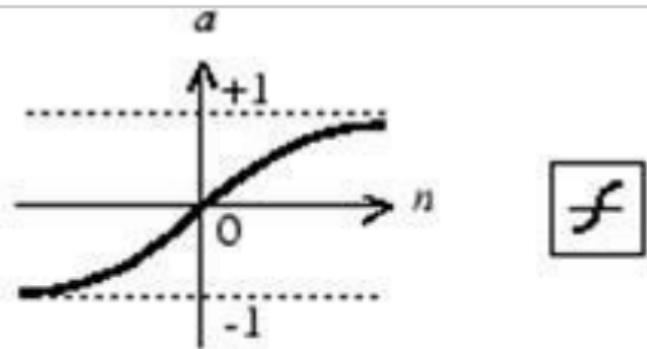
`mu` exceeds `mu_max`.

Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## 10.2.7 tansig

Hyperbolic tangent sigmoid transfer function

### Graph and Symbol



$$a = \tanh(n)$$

Tan-Sigmoid Transfer Function

### Syntax

---

`A = tansig(N, FP)`

### Description

---

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

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

`N`        s-by-Q matrix of net input  
(column) vectors

`FP`        Struct of function parameters  
(ignored)

and returns `A`, the s-by-Q matrix of `N`'s  
elements squashed into `[-1 1]`.

## Examples

---

Here is the code to create a plot of  
the `tansig` transfer function.

```
n = -5:0.1:5;
```

```
a = tansig(n);
```

```
plot(n,a)
```

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

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

## Algorithms

---

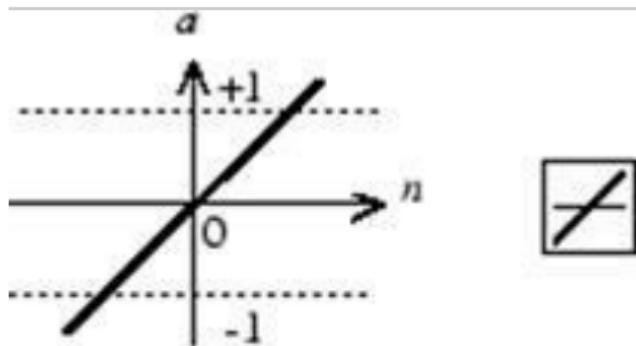
```
a = tansig(n) =  
2 / (1+exp(-2*n)) - 1
```

This is mathematically equivalent to  $\tanh(N)$ . It differs in that it runs faster than the MATLAB implementation of  $\tanh$ , but the results can have very small numerical differences. This function is a good tradeoff for neural networks, where speed is important and the exact shape of the transfer function is not.

## 10.2.8 purelin

Linear transfer function

### Graph and Symbol



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

Linear Transfer Function

### Syntax

`A = purelin(N, FP)`

`info = purelin('code')`

# Description

---

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

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

`N`        s-by-Q matrix of net input  
(column) vectors

`FP`        Struct of function parameters  
(ignored)

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

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

supported *code* string:

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

`purelin('output', FP)` returns the [min max] output range.

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

`purelin('fulllderiv')` returns 1 or 0, depending on whether `dA_dN` is s-by-s-by-Q or s-by-Q.

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

`purelin('fpdefaults')` returns the default function parameters.

# Examples

---

Here is the code to create a plot of the purelin transfer function.

```
n = -5:0.1:5;
```

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

```
plot(n,a)
```

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

```
net.layers{i}.transf
```

```
= 'purelin';
```

## Algorithms

---

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

## 10.2.9 cascadeforwardnet

Cascade-forward neural network

### Syntax

---

```
cascadeforwardnet(hiddenSizes)
```

### 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)`  
these arguments,

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

and returns a new cascade-forward neural network.

## Examples

### Create and Train a Cascade Network

---

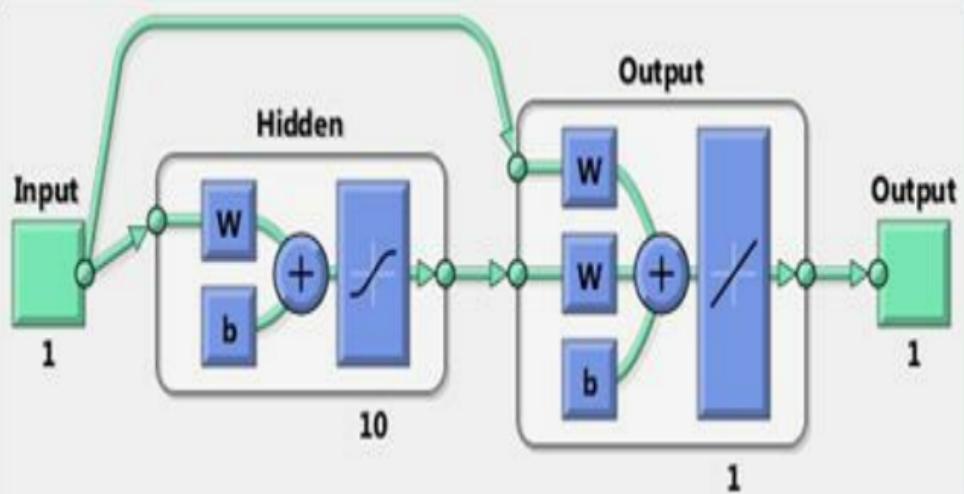
Here a cascade network is created and trained on a simple fitting problem.

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

```
perf =  
perform(net, y, t)
```

```
perf =
```

1.9372e-05



## 10.2.10 patternnet

Pattern recognition network

### Syntax

---

```
patternnet(hiddenSizes, trainF
```

### 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, )
```

these arguments,

hiddenSizes      Row vector of one or more hidden layer sizes (default = 10)

trainFcn          Training function  
(default = 'trainscg')

performFcn       Performance function  
(default = 'crossentropy')

and returns a pattern recognition neural network.

## Examples

### Pattern Recognition

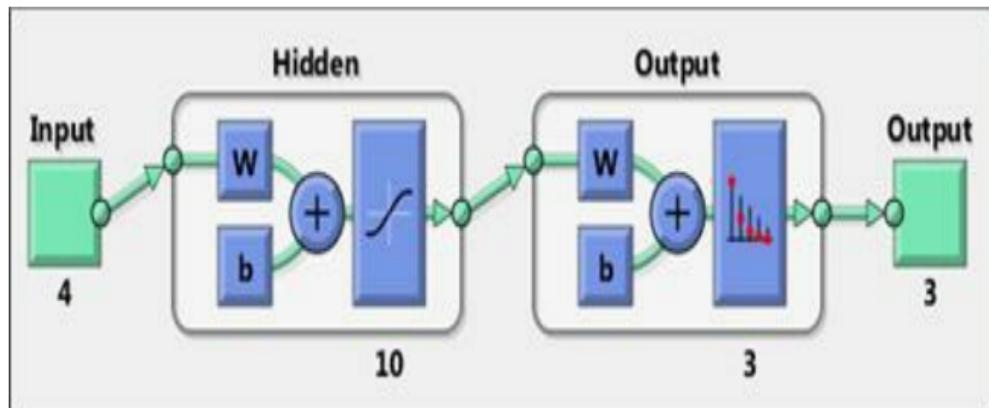
---

This example shows how to design a pattern recognition network to classify iris flowers.

```
[x,t] =  
iris_dataset;  
  
net =  
patternnet(10);  
  
net =  
train(net,x,t);  
  
view(net)  
  
y = net(x);
```

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

```
classes =  
vec2ind(y);
```



# 10.3 TRAIN AND APPLY MULTILAYER NEURAL NETWORKS

This topic presents part of a typical multilayer network workflow.

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

The process of training a neural network

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

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

(Individual squared errors can also be weighted. There are two different ways in which training can be implemented: incremental mode and batch mode. In

incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs in the training set are applied to the network before the weights are updated. This topic describes batch mode training with the [train](#) command. Incremental training with the [adapt](#) command is discussed in [Incremental Training with adapt](#). 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](#)].

## 10.3.1 Training Algorithms

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

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

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

is iterated until the network converges.

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

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

Function	Algorithm
<a href="#"><u>trainlm</u></a>	Levenberg-Marquardt
<a href="#"><u>trainbr</u></a>	Bayesian Regularization
<a href="#"><u>trainbfg</u></a>	BFGS Quasi-Newton
<a href="#"><u>trainrp</u></a>	Resilient Backpropagation
<a href="#"><u>trainscg</u></a>	Scaled Conjugate Gradient
<a href="#"><u>traincgb</u></a>	Conjugate Gradient with Powell/Beale Restarts
<a href="#"><u>traincgf</u></a>	Fletcher-Powell Conjugate Gradient
<a href="#"><u>traincqp</u></a>	Polak-Ribière Conjugate Gradient
<a href="#"><u>trainoss</u></a>	One Step Secant

<a href="#">trainqdx</a>	Variable Learning Rate Gradient Descent
<a href="#">trainqdm</a>	Gradient Descent with Momentum
<a href="#">trainqd</a>	Gradient Descent

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

When training large networks, and when

training pattern recognition networks, [`trainscg`](#) and [`trainrp`](#) are good choices. Their memory requirements are relatively small, and yet they are much faster than standard gradient descent algorithms.

See [Choose a Multilayer Neural Network Training Function](#) for a full comparison of the performances of the training algorithms shown in the table above.

As a note on terminology, the term "backpropagation" is sometimes used to refer specifically to the gradient descent algorithm, when applied to neural network training. That terminology is not used here, since the process of

computing the gradient and Jacobian by performing calculations backward through the network is applied in all of the training functions listed above. It is clearer to use the name of the specific optimization algorithm that is being used, rather than to use the term backpropagation alone.

Also, the multilayer network is sometimes referred to as a backpropagation network. However, the backpropagation technique that is used to compute gradients and Jacobians in a multilayer network can also be applied to many different network architectures. In fact, the gradients and Jacobians for any network that has differentiable

transfer functions, weight functions and net input functions can be computed using the Neural Network Toolbox software through a backpropagation process. You can even create your own custom networks and then train them using any of the training functions in the table above. The gradients and Jacobians will be automatically computed for you.

## 10.3.2 Training Example

To illustrate the training process, execute the following commands:

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

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

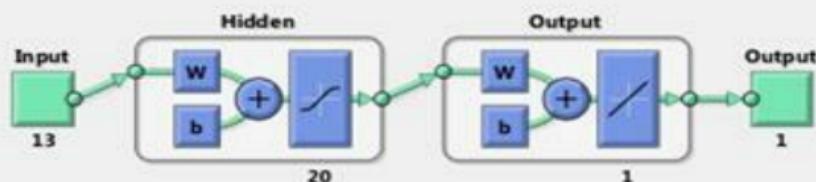
parameter `net.trainParam.min_grad`.

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.)

# Neural Network Training (nntraintool)

## Neural Network



## Algorithms

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

## Progress

Epoch:	0	12 iterations	1000
Time:		0:00:02	
Performance:	517	1.78	0.00
Gradient:	$1.78 \times 10^3$	16.0	$1.00 \times 10^{-7}$
Mu:	0.00100	0.0100	$1.00 \times 10^{10}$
Validation Checks:	0	6	6

## Plots

**Performance** (plotperform)

**Training State** (plottrainstate)

**Error Histogram** (ploterrhist)

**Regression** (plotregression)

Plot Interval:



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 might want to do this if the performance function fails to decrease significantly over many iterations. It is always possible to continue the training by reissuing the [train](#) command shown above. It will continue to train the network from the completion of the

previous run.

From the training window, you can access four plots: performance, training state, error histogram, and regression. The performance plot shows the value of the performance function versus the iteration number. It plots training, validation, and test performances. The training state plot shows the progress of other training variables, such as the gradient magnitude, the number of validation checks, etc. The error histogram plot shows the distribution of the network errors. The regression plot shows a regression between network outputs and network targets. You can use the histogram and regression plots to

validate network performance.

### 10.3.3 Use the Network

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

```
a =
```

```
net(houseInputs(:, 5))
```

```
a =
```

34.3922

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

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

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

# 10.4 TRAIN ALGORITHMS IN MULTILAYER NEURAL NETWORKS

The next table resume the MATLAB training algorithms for multilayer neural networks

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
traincfg	Fletcher-Powell Conjugate Gradient
traincgp	Polak-Ribière Conjugate Gradient
trainoss	One Step Secant
traingdx	Variable Learning Rate Gradient Descent
traingdm	Gradient Descent with Momentum
traingd	Gradient Descent

## 10.4.1 trainbr:Bayesian Regularization

### Syntax

```
net.trainFcn = 'trainbr'  
[net,tr] = train(net,...)
```

### Description

trainbr is a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes

well. The process is called Bayesian regularization.

`net.trainFcn = 'trainbr'` sets the network `trainFcn` property.

`[net, tr] = train(net, ...)` trains the network with `trainbr`.

Training occurs according to `trainbr` training parameters, shown here with their default values:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.mu</code>	0.005	Marquardt algorithm step size
<code>net.trainParam.mu_dec</code>	0.1	Decrease factor for mu
<code>net.trainParam.mu_inc</code>	10	Increase factor for mu
<code>net.trainParam.mu_max</code>	$1e+10$	Maximum value for mu
<code>net.trainParam.max_fail</code>	0	Maximum number of failed iterations
<code>net.trainParam.min_grad</code>	$1e-7$	Minimum performance gradient
<code>net.trainParam.show</code>	25	Epochs between progress displays
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training progress window
<code>net.trainParam.time</code>	inf	Maximum time to run

Validation stops are disabled by default (`max_fail = 0`) so that training can continue until an optimal combination of errors and weights is found. However, some weight/bias minimization can still be achieved with shorter training times if validation is enabled by setting `max_fail` to 6 or some other strictly positive value.

## Network Use

You can create a standard network that uses `trainbr` with `feedforwardnet` or . To prepare a custom network to be trained with `trainbr`,

Set `NET.trainFcn` to '`'trainbr'`'.  
This  
sets `NET.trainParam` to `'trainbr'`'  
default parameters.

Set `NET.trainParam` properties  
to desired values.

In either case, calling `train` with the  
resulting network trains the network  
with `trainbr`.

See `feedforwardnet` and `cascadeforw`  
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*rand;
```

A feed-forward network is created with a hidden layer of 2 neurons.

```
net =
```

```
feedforwardnet(2,'tr');
```

Here the network is trained and tested.

```
net =  
train(net,p,t);
```

```
a = net(p)
```

## Limitations

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

## Algorithms

`trainbr` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Bayesian regularization minimizes a linear combination of squared errors and weights. It also modifies the linear combination so that at the end of training the resulting network has good generalization qualities. See MacKay (*Neural Computation*, Vol. 4, No. 3, 1992, pp. 415 to 447) and Foresee and Hagan (*Proceedings of the International Joint Conference on Neural Networks*, June, 1997) for more detailed discussions of Bayesian regularization.

This Bayesian regularization takes place within the Levenberg-Marquardt algorithm. Backpropagation is used to calculate the Jacobian  $\frac{\partial \text{perf}}{\partial x}$  of performance  $\text{perf}$  with respect to the weight and bias variables  $x$ . Each variable is adjusted according to Levenberg-Marquardt,

$$\frac{\partial J}{\partial j} = \frac{\partial \text{perf}}{\partial x} * \frac{\partial x}{\partial j}$$

$$\frac{\partial J}{\partial e} = \frac{\partial \text{perf}}{\partial x} * E$$

$$dX = -(\frac{\partial J}{\partial j} + I * mu) \backslash \frac{\partial J}{\partial e}$$

where  $E$  is all errors and  $I$  is the identity

matrix.

The adaptive value  $\mu$  is increased by  $\mu_{inc}$  until the change shown above results in a reduced performance value. The change is then made to the network, and  $\mu$  is decreased by  $\mu_{dec}$ .

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below  $min\_grad$ .

• mu exceeds mu\_max.

## 10.4.2 trainscg: Scaled conjugate gradient backpropagation

### Syntax

```
net.trainFcn = 'trainscg'  
[net,tr] = train(net,...)
```

### Description

`trainscg` is a network training function that updates weight and bias values according to the scaled conjugate gradient method.

`net.trainFcn = 'trainscg'` sets the network `trainFcn` property.

[net,tr] = train(net,...) trains  
the network with trainscg.

Training occurs according  
to trainscg training parameters, shown  
here with their default values:

net.trainParam.epochs	1000	Maximum number
net.trainParam.show	25	Epochs between di
net.trainParam.showCommandLine	false	Generate command
net.trainParam.showWindow	true	Show training GU
net.trainParam.goal	0	Performance goal
net.trainParam.time	inf	Maximum time to
net.trainParam.min_grad	1e-6	Minimum perform
net.trainParam.max_fail	6	Maximum validati
net.trainParam.sigma	5.0e-5	Determine change approximation
net.trainParam.lambda	5.0e-7	Parameter for regul

## Network Use

You can create a standard network that  
uses trainscg with feedforwardnet or

To prepare a custom network to be trained with `trainscg`,

.

Set `net.trainFcn` to '`trainscg`'  
This  
sets `net.trainParam` to `trainscg`  
default parameters.

- Set `net.trainParam` properties to desired values.
- In either case, calling `train` with the resulting network trains the network with `trainscg`.

## Examples

Here is a problem consisting of inputs  $p$  and targets  $t$  to be solved with a network.

$p = [0 \ 1 \ 2 \ 3 \ 4 \ 5];$

$t = [0 \ 0 \ 0 \ 1 \ 1 \ 1];$

A two-layer feed-forward network with two hidden neurons and this training function is created.

`net =  
feedforwardnet(2, 'tr.  
Here the network is trained and retested.`

```
net =  
train(net,p,t);
```

```
a = net(p)
```

See `help feedforwardnet` and `help cascadeforwardnet` for other examples.

## Algorithms

`trainscg` can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance `perf` with

respect to the weight and bias variables  $x$ .

The scaled conjugate gradient algorithm is based on conjugate directions, as in `traincgp`, `traincfg`, and `traincgb`, but this algorithm does not perform a line search at each iteration. See Moller (*Neural Networks*, Vol. 6, 1993, pp. 525–533) for a more detailed discussion of the scaled conjugate gradient algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is

exceeded.

- Performance is minimized to the goal.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## 10.4.3 trainrp: Resilient backpropagation

### Syntax

```
net.trainFcn = 'trainrp'  
[net,tr] = train(net,...)
```

### Description

trainrp is a network training function that updates weight and bias values according to the resilient backpropagation algorithm (Rprop).

net.trainFcn = 'trainrp' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainrp.

Training occurs according to trainrp training parameters, shown here with their default values:

net.trainParam.epochs	1000	Maximum number of epochs
net.trainParam.show	25	Epochs between progress displays
net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showWindow	true	Show training progress window
net.trainParam.goal	0	Performance goal
net.trainParam.time	inf	Maximum time to train
net.trainParam.min_grad	1e-5	Minimum performance gradient
net.trainParam.max_fail	6	Maximum validation failures
net.trainParam.lr	0.01	Learning rate
net.trainParam.delt_inc	1.2	Increment to weight change
net.trainParam.delt_dec	0.5	Decrement to weight change
net.trainParam.delta0	0.07	Initial weight change
net.trainParam.deltamax	50.0	Maximum weight change

## Network Use

You can create a standard network that

uses trainrp with feedforwardnet or cascadeforwardnet.

To prepare a custom network to be trained with trainrp,

- Set net.trainFcn to 'trainrp'. This sets net.trainParam to trainrp's default parameters.
- Set net.trainParam properties to desired values.

In either case, calling train with the resulting network trains the network with trainrp.

## Examples

Here is a problem consisting of

inputs p and targets t to be solved with a network.

$$p = [0 \ 1 \ 2 \ 3 \ 4 \ 5];$$

$$t = [0 \ 0 \ 0 \ 1 \ 1 \ 1];$$

A two-layer feed-forward network with two hidden neurons and this training function is created.

Create and test a network.

$$\text{net} = \text{feedforwardnet}(2, \text{'trainrp'});$$

Here the network is trained and retested.

$$\text{net.trainParam.epochs} = 50;$$

$$\text{net.trainParam.show} = 10;$$

$$\text{net.trainParam.goal} = 0.1;$$

```
net = train(net,p,t);
```

```
a = net(p)
```

See `help feedforwardnet` and `help cascadeforwardnet` for other examples.

## Definitions

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called "squashing" functions, because they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slopes must approach zero as the input gets large. This causes a problem when you use steepest descent to train a

multilayer network with sigmoid functions, because the gradient can have a very small magnitude and, therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation (Rprop) training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative can determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and

bias is increased by a factor  $\text{delt\_inc}$  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  $\text{delt\_dec}$  whenever the derivative with respect to that weight changes sign from the previous iteration. If the derivative is zero, the update value remains the same. Whenever the weights are oscillating, the weight change is reduced. If the weight continues to change in the same direction for several iterations, the magnitude of the weight change increases. A complete description of the Rprop algorithm is

given in [[RiBr93](#)].

The following code recreates the previous network and trains it using the Rprop algorithm. The training parameters

for `trainrp` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `delt_inc`, `delt_dec`, `d` and `deltamax`. The first eight parameters have been previously discussed. The last two are the initial step size and the maximum step size, respectively. The performance of Rprop is not very sensitive to the settings of the training parameters. For the example below, the training parameters are left at the default values:

$$p = [-1 \ -1 \ 2 \ 2; 0 \ 5 \ 0 \ 5];$$

```
t = [-1 -1 1 1];
```

```
net = feedforwardnet(3,'trainrp');
```

```
net = train(net,p,t);
```

```
y = net(p)
```

rprop is generally much faster than the standard steepest descent algorithm. It also has the nice property that it requires only a modest increase in memory requirements. You do need to store the update values for each weight and bias, which is equivalent to storage of the gradient.

## Algorithms

trainrp can train any network as long as

its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

$$dX = \text{delta}X \cdot \text{sign}(gX);$$

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  $\delta X$  is increased by  $\delta_{inc}$ . See Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, 1993, pp. 586–591.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to

the goal.

- The performance gradient falls below `min_grad`.

- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## 10.4.4 trainbfg: BFGS quasi-Newton backpropagation

### Syntax

```
net.trainFcn = 'trainbfg'  
[net,tr] = train(net,...)
```

### Description

trainbfg is a network training function that updates weight and bias values according to the BFGS quasi-Newton method.

net.trainFcn = 'trainbfg' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainbfg.

Training occurs according to trainbfg training parameters, shown here with their default values:

net.trainParam.epochs	1000	Maximum
net.trainParam.showWindow	true	Show training progress
net.trainParam.show	25	Epochs between progress displays
net.trainParam.showCommandLine	false	Generate command line output
net.trainParam.goal	0	Performance goal
net.trainParam.time	inf	Maximum time to run
net.trainParam.min_grad	1e-6	Minimum gradient
net.trainParam.max_fail	6	Maximum number of consecutive failures
net.trainParam.searchFcn	'srchbac'	Name of line search function

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to determine step size
net.trainParam.alpha	0.001	Scale factor that determines step length
net.trainParam.beta	0.1	Scale factor that determines step length
net.trainParam.delta	0.01	Initial step size in interval
net.trainParam.gama	0.1	Parameter to avoid small step sizes (see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in parameter
net.trainParam.up_lim	0.5	Upper limit on change in parameter
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 nonzero value implies a fix conditions of a previous training conditions for the next epoch

# Network Use

You can create a standard network that uses trainbfg with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with trainbfg:

Set NET.trainFcn to 'trainbfg'  
This sets NET.trainParam to trainbfg default parameters.

Set `NET.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainbfg`.

## Examples

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

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

```
net =  
feedforwardnet(10,'t';  
  
net =  
train(net,x,t);  
  
y = net(x)
```

## Definitions

Newton's method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton's method is

$$\mathbf{X}_{k+1} = \mathbf{X}_k - \mathbf{A}_{-1k} \mathbf{g}_k$$

where  $\mathbf{A}_{-1k}$  is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton's method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks. There is a class of algorithms that is based on Newton's method, but which does not require calculation of second derivatives. These are called quasi-Newton (or secant) methods. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. The quasi-Newton method that

has been most successful in published studies is the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update. This algorithm is implemented in the `trainbfg` routine.

The BFGS algorithm is described in [DeSc83]. This algorithm requires more computation in each iteration and more storage than the conjugate gradient methods, although it generally converges in fewer iterations. The approximate Hessian must be stored, and its dimension is  $n \times n$ , where  $n$  is equal to the number of weights and biases in the network. For very large networks it might be better to use Rprop or one of the conjugate gradient algorithms. For

smaller networks, however, `trainbfg` can be an efficient training function.

## Algorithms

`trainbfg` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `x`. Each variable is adjusted according to the following:

$$X = X + a * dX;$$

where  $\text{d}x$  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:

$$\text{d}X = -H \backslash gX;$$

where  $gX$  is the gradient and  $H$  is a approximate Hessian matrix. See page 119 of Gill, Murray, and Wright (*Practical Optimization*, 1981) for a

more detailed discussion of the BFGS quasi-Newton method.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min\_grad.
- Validation performance has increased more than max\_fail times since the last time it decreased (when using

validation).

# **10.4.5 traincgb: Conjugate gradient backpropagation with Powell-Beale restarts**

## **Syntax**

```
net.trainFcn = 'traincgb'  
[net,tr] = train(net,...)
```

## **Description**

`traincgb` is a network training function that updates weight and bias values according to the conjugate gradient backpropagation with Powell-Beale restarts.

`net.trainFcn = 'traincgb'` sets the network `trainFcn` property.

`[net, tr] = train(net, ...)` trains the network with `traincgb`.

Training occurs according to `traincgb` training parameters, shown here with their default values:

<code>net.trainParam.epochs</code>	1000	Maximum
<code>net.trainParam.show</code>	25	Epochs be
<code>net.trainParam.showCommandLine</code>	false	Generate c
<code>net.trainParam.showWindow</code>	true	Show trai
<code>net.trainParam.goal</code>	0	Performan
<code>net.trainParam.time</code>	inf	Maximum
<code>net.trainParam.min_grad</code>	1e-10	Minimum
<code>net.trainParam.max_fail</code>	6	Maximum
<code>net.trainParam.searchFcn</code>	'srchcha'	Name of l

Parameters related to line search methods (not all used for all methods):

<code>net.trainParam.scal_tol</code>	20	Divide into delta to determ
<code>net.trainParam.alpha</code>	0.001	Scale factor that determines si

net.trainParam.beta	0.1	Scale factor that determines si
net.trainParam.delta	0.01	Initial step size in interval lo
net.trainParam.gama	0.1	Parameter to avoid small r to 0.1(see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in ste
net.trainParam.up_lim	0.5	Upper limit on change in ste
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e- 6	Minimum step length
net.trainParam.bmax	26	Maximum step size

## Network Use

You can create a standard network that uses `traincgb` with `feedforwardnet` or

To prepare a custom network to be trained with `traincgb`,

.

Set `net.trainFcn` to '`traincgb`'  
This

sets `net.trainParam` to `traincgb` default parameters.

Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincgb`.

## Examples

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

[`x, t`] =

```
house_dataset;  
  
net =  
feedforwardnet(10, 't:  
  
net =  
train(net, x, t);  
  
y = net(x)
```

## Definitions

For all conjugate gradient algorithms, the search direction is periodically reset to the negative of the gradient. The

standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods that can improve the efficiency of training. One such reset method was proposed by Powell [[Powe77](#)], based on an earlier version proposed by Beale [[Beal72](#)]. This technique restarts if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality:

If this condition is satisfied, the search direction is reset to the negative of the gradient.

The `traincgb` routine has somewhat

better performance than `traincgp` for some problems, although performance on any given problem is difficult to predict. The storage requirements for the Powell-Beale algorithm (six vectors) are slightly larger than for Polak-Ribière (four vectors).

## Algorithms

`traincgb` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias

variables  $x$ . Each variable is adjusted according to the following:

$$X = X + a * dX;$$

where  $dX$  is the search direction. The parameter  $a$  is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous search direction according to the formula

$dX = -gX + dX_{old} * z;$   
where  $gX$  is the gradient. The parameter  $z$  can be computed in several different ways. The Powell-Beale variation of conjugate gradient is distinguished by two features. First, the algorithm uses a test to determine when to reset the search direction to the negative of the gradient. Second, the search direction is computed from the negative gradient, the previous search direction, and the last search direction before the previous reset. See Powell, *Mathematical Programming*, Vol. 12, 1977, pp. 241 to 254, for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

# **10.4.6 traincfgf: Conjugate gradient backpropagation with Fletcher-Reeves updates**

## **Syntax**

```
net.trainFcn = 'traincfgf'  
[net,tr] = train(net,...)
```

## **Description**

traincfgf is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Fletcher-Reeves updates.

`net.trainFcn = 'traincfg'` sets the network `trainFcn` property.

`[net,tr] = train(net,...)` trains the network with `traincfg`.

Training occurs according to `traincfg` training parameters, shown here with their default values:

<code>net.trainParam.epochs</code>	1000	Maximum number of epochs to train
<code>net.trainParam.show</code>	25	Epochs between displays (NaN for no displays)
<code>net.trainParam.showCommandLine</code>	false	Generate command-line output
<code>net.trainParam.showWindow</code>	true	Show training GUI
<code>net.trainParam.goal</code>	0	Performance goal
<code>net.trainParam.time</code>	inf	Maximum time to train in seconds
		Minimum

net.trainParam.min_grad	1e-10	performance gradient
net.trainParam.max_fail	6	Maximum validation failures
net.trainParam.searchFcn	'srchcha'	Name of line search routine to use

Parameters related to line search methods (not all used for all methods):

net.trainParam.scal_tol	20	Divide into delta to c
net.trainParam.alpha	0.001	Scale factor that deter
net.trainParam.beta	0.1	Scale factor that deter
net.trainParam.delta	0.01	Initial step size in inte
net.trainParam.gama	0.1	Parameter to avoid sm to 0.1(see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change
net.trainParam.up_lim	0.5	Upper limit on change
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 `traincfg` with `feedforwardnet` or `cascadeforwardnet`.

To prepare a custom network to be trained with `traincfg`,

- Set `net.trainFcn` to '`traincfg`'. This sets `net.trainParam` to `traincfg`'s default parameters.
- Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincfg`.

# Examples

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

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

```
net = feedforwardnet(10,'traincfg');
```

```
net = train(net,x,t);
```

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

# Definitions

All the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

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

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of the conjugate

gradient algorithm are distinguished by the manner in which the constant  $\beta_k$  is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [[FlRe64](#)] or [[HDB96](#)] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are

sometimes faster than trainrp, although the results vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms. Therefore, these algorithms are good for networks with a large number of weights.

Try the *Neural Network Design* demonstration nnd12cg [[HDB96](#)] for an illustration of the performance of a conjugate gradient algorithm.

## Algorithms

traincfg can train any network as long as its weight, net input, and transfer

functions have derivative functions.

Backpropagation is used to calculate derivatives of performance perf with respect to the weight and bias variables X. Each variable is adjusted according to the following:

$$X = X + a * dX;$$

where  $dX$  is the search direction. The parameter  $a$  is selected to minimize the performance along the search direction. The line search function searchFcn is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new

gradient and the previous search direction, according to the formula

$$dX = -gX + dX_{\text{old}} * Z;$$

where  $gX$  is the gradient. The parameter  $Z$  can be computed in several different ways. For the Fletcher-Reeves variation of conjugate gradient it is computed according to

$$Z = \text{normnew\_sqr} / \text{norm\_sqr};$$

where  $\text{norm\_sqr}$  is the norm square of the previous gradient and  $\text{normnew\_sqr}$  is the norm square of the current gradient. See page 78 of Scales (*Introduction to Non-Linear Optimization*) for a more detailed discussion of the algorithm.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

# **10.4.7 traincgp: Conjugate gradient backpropagation with Polak-Ribière updates**

## **Syntax**

```
net.trainFcn = 'traincgp'  
[net, tr] = train(net, ...)
```

## **Description**

`traincgp` is a network training function that updates weight and bias values according to conjugate gradient backpropagation with Polak-Ribière updates.

`net.trainFcn = 'traincgp'` sets the network `trainFcn` property.

`[net, tr] = train(net, ...)` trains the network with `traincgp`.

Training occurs according to `traincgp` training parameters, shown here with their default values:

<code>net.trainParam.epochs</code>	1000	Maximum
<code>net.trainParam.show</code>	25	Epochs be
<code>net.trainParam.showCommandLine</code>	false	Generate c
<code>net.trainParam.showWindow</code>	true	Show trai
<code>net.trainParam.goal</code>	0	Performan
<code>net.trainParam.time</code>	inf	Maximum
<code>net.trainParam.min_grad</code>	1e-10	Minimum
<code>net.trainParam.max_fail</code>	6	Maximum
<code>net.trainParam.searchFcn</code>	'srchcha'	Name of l

Parameters related to line search methods (not all used for all methods):

<code>net.trainParam.scal_tol</code>	20	Divide into delta to determ
<code>net.trainParam.alpha</code>	0.001	Scale factor that determines si

net.trainParam.beta	0.1	Scale factor that determines si
net.trainParam.delta	0.01	Initial step size in interval lo
net.trainParam.gama	0.1	Parameter to avoid small r to 0.1(see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in ste
net.trainParam.up_lim	0.5	Upper limit on change in ste
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e- 6	Minimum step length
net.trainParam.bmax	26	Maximum step size

## Network Use

You can create a standard network that uses traincgp with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traincgp,

Set net.trainFcn to 'traincgp' sets net.trainParam to traincgp's default parameters.

Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traincgp`.

## Examples

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

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

```
net =  
feedforwardnet(10,'t';  
  
net =  
train(net,x,t);  
  
y = net(x)
```

## Definitions

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, `traincgf`, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribière update, the constant  $\beta_k$  is computed by

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [[FlRe64](#)] or [[HDB96](#)] for a discussion of the Polak-Ribière conjugate gradient algorithm.

The `traincgp` routine has performance similar to `traincgf`. It is difficult to predict which algorithm will perform best on a given problem. The storage

requirements for Polak-Ribi  re (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

## Algorithms

`traincgp` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `x`. Each variable is adjusted according to the following:

$$X = X + a * dX;$$

where  $\text{dx}$  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

$$\text{dX} = -g\text{X} + \text{dX\_old}^* z;$$

where  $g\text{X}$  is the gradient. The parameter  $z$  can be computed in several different ways. For the Polak-Ribière

variation of conjugate gradient, it is computed according to

$$Z = ((gX - gX_{old})' * gX) / \text{norm\_sq}$$

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).

## 10.4.8 trainoss: One-step secant backpropagation

### Syntax

```
net.trainFcn = 'trainoss'  
[net,tr] = train(net,...)
```

### Description

trainoss is a network training function that updates weight and bias values according to the one-step secant method.

net.trainFcn = 'trainoss' sets the network trainFcn property.

[net,tr] = train(net,...) trains the network with trainoss.

Training occurs according to trainoss training parameters, shown here with their default values:

net.trainParam.epochs	1000	Maximum
net.trainParam.goal	0	Performance
net.trainParam.max_fail	6	Maximum
net.trainParam.min_grad	1e-10	Minimum
net.trainParam.searchFcn	'srchbac'	Name of line search function
net.trainParam.show	25	Epochs between displays
net.trainParam.showCommandLine	false	Generate command line output
net.trainParam.showWindow	true	Show training progress window
net.trainParam.time	inf	Maximum time per epoch

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
		Parameter to avoid small

net.trainParam.gama	0.1	reductions in performance, usually set to 0.1(see srch_cha)
net.trainParam.low_lim	0.1	Lower limit on change in step size
net.trainParam.up_lim	0.5	Upper limit on change in step size
net.trainParam.maxstep	100	Maximum step length
net.trainParam.minstep	1.0e-6	Minimum step length
net.trainParam.bmax	26	Maximum step size

## Network Use

You can create a standard network that uses trainoss with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with trainoss:

Set net.trainFcn to 'trainoss'  
This sets net.trainParam to

`trainoss`'s default parameters.

Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `trainoss`.

## Examples

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

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

```
net =  
feedforwardnet(10, 't');
```

```
net =  
train(net, x, t);
```

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

## Definitions

Because the BFGS algorithm requires more storage and computation in each iteration than the conjugate gradient algorithms, there is need for a secant approximation with smaller storage and

computation requirements. The one step secant (OSS) method is an attempt to bridge the gap between the conjugate gradient algorithms and the quasi-Newton (secant) algorithms. This algorithm does not store the complete Hessian matrix; it assumes that at each iteration, the previous Hessian was the identity matrix. This has the additional advantage that the new search direction can be calculated without computing a matrix inverse.

The one step secant method is described in [[Batt92](#)]. This algorithm requires less storage and computation per epoch than the BFGS algorithm. It requires slightly more storage and computation per epoch

than the conjugate gradient algorithms. It can be considered a compromise between full quasi-Newton algorithms and conjugate gradient algorithms.

## Algorithms

trainoss can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `x`. Each variable is adjusted according to the following:

$$X = X + a^* dX;$$

where  $dX$  is the search direction. The parameter  $a$  is selected to minimize the performance along the search direction. The line search function `searchFcn` is used to locate the minimum point. The first search direction is the negative of the gradient of performance. In succeeding iterations the search direction is computed from the new gradient and the previous steps and gradients, according to the following formula:

$$dX = -gX + A_c * X_{\text{step}}$$

$$+ B C^* d g X;$$

where  $g_X$  is the gradient,  $x_{\text{step}}$  is the change in the weights on the previous iteration, and  $d g_X$  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).

## **10.4.9 traingdx: Gradient descent with momentum and adaptive learning rate backpropagation**

### **Syntax**

```
net.trainFcn = 'traingdx'  
[net,tr] = train(net,...)
```

### **Description**

traingdx is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

net.trainFcn = 'traingdx' sets the

network trainFcn property.

[net, tr] = train(net, ...) trains the network with traingdx.

Training occurs according to traingdx training parameters, shown here with their default values:

net.trainParam.epochs	1000	Maximum number of epochs
net.trainParam.goal	0	Performance goal
net.trainParam.lr	0.01	Learning rate
net.trainParam.lr_inc	1.05	Ratio to increase learning rate
net.trainParam.lr_dec	0.7	Ratio to decrease learning rate
net.trainParam.max_fail	6	Maximum number of consecutive failed attempts
net.trainParam.max_perf_inc	1.04	Maximum performance increase
net.trainParam.mc	0.9	Momentum coefficient
net.trainParam.min_grad	1e-5	Minimum gradient
net.trainParam.show	25	Epochs between progress displays
net.trainParam.showCommandLine	false	Generate command line output
net.trainParam.showWindow	true	Show training progress window
net.trainParam.time	inf	Maximum time to run

## Network Use

You can create a standard network that uses `traingdx` with `feedforwardnet` or `cascadeforwardnet`. To prepare a custom network to be trained with `traingdx`,

Set `net.trainFcn` to '`traingdx`'.  
This sets `net.trainParam` to `traingdx` default parameters.

Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingdx`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

## Definitions

The function `traingdx` combines adaptive learning rate with momentum training. It is invoked in the same way as `traingda`, except that it has the momentum coefficient `mc` as an additional training parameter.

## Algorithms

`traingdx` can train any network as long as its weight, net input, and transfer

functions have derivative functions.

Backpropagation is used to calculate derivatives of performance  $\text{perf}$  with respect to the weight and bias variables  $x$ . Each variable is adjusted according to gradient descent with momentum,

$$dX = mc * dX_{\text{prev}} + lr * mc * \frac{d\text{perf}}{dX}$$

where  $dX_{\text{prev}}$  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).

## 10.4.10 traingdm: Gradient descent with momentum backpropagation

### Syntax

```
net.trainFcn = 'traingdm'  
[net,tr] = train(net,...)
```

### Description

traingdm is a network training function that updates weight and bias values according to gradient descent with momentum.

net.trainFcn = 'traingdm' sets the network trainFcn property.

[net, tr] = train(net, ...) trains the network with traingdm.

Training occurs according to traingdm training parameters, shown here with their default values:

net.trainParam.epochs	1000	Maxii
net.trainParam.goal	0	Perfor
net.trainParam.lr	0.01	Learni
net.trainParam.max_fail	6	Maxii
net.trainParam.mc	0.9	Mom
net.trainParam.min_grad	1e-5	Minir
net.trainParam.show	25	Epoch
net.trainParam.showCommandLine	false	Gener
net.trainParam.showWindow	true	Show
net.trainParam.time	inf	Maxii

## Network Use

You can create a standard network that uses traingdm with feedforwardnet or cascadeforwardnet. To prepare a

custom network to be trained with traingdm,

1.

Set `net.trainFcn` to '`traingdm`'.

This

sets `net.trainParam` to `traingdm`'s default parameters.

2.

Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingdm`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

# Definitions

In addition to `traingd`, there are three other variations of gradient descent.

Gradient descent with momentum, implemented by `traingdm`, allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum. See page 12–9 of [[HDB96](#)] for a discussion of momentum.

Gradient descent with momentum depends on two training parameters. The parameter `lr` indicates the learning rate, similar to the simple gradient descent. The parameter `mc` is the momentum constant that defines the amount of momentum. `mc` is set between 0 (no momentum) and values close to 1 (lots of momentum). A momentum constant of 1 results in a network that is completely insensitive to the local gradient and, therefore, does not learn properly.)

```
p = [-1 -1 2 2; 0 5  
0 5];
```

```
t = [-1 -1 1 1];  
  
net =  
feedforwardnet(3, 'tr.  
  
net.trainParam.lr =  
0.05;  
  
net.trainParam.mc =  
0.9;  
  
net =  
train(net, p, t);
```

$y = \text{net}(p)$

Try the *Neural Network Design* demonstration `nnd12mo` [[HDB96](#)] for an illustration of the performance of the batch momentum algorithm.

## Algorithms

`traingdm` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `x`. Each variable is adjusted

according to gradient descent with momentum,

$$dX = mc * dX_{prev} + lr * (1 - mc) * \frac{dperf}{dX}$$

where  $dX_{prev}$  is the previous change to the weight or bias.

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.

- The performance gradient falls below `min_grad`.
- Validation performance has increased more than `max_fail` times since the last time it decreased (when using validation).

## 10.4.11 traingd: Gradient descent backpropagation

### Syntax

```
net.trainFcn = 'traingd'  
[net,tr] = train(net,...)
```

### Description

traingd is a network training function that updates weight and bias values according to gradient descent.

net.trainFcn = 'traingd' sets the network trainFcn property.

[net,tr] = train(net,...) trains

the network with traingd.

Training occurs according to traingd training parameters, shown here with their default values:

net.trainParam.epochs	1000	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.showCommandLine	false	Generate command-line output
net.trainParam.showWindow	true	Show training GUI
net.trainParam.lr	0.01	Learning rate
net.trainParam.max_fail	6	Maximum validation failures
net.trainParam.min_grad	1e-5	Minimum performance gradient
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam.time	inf	Maximum time to train in seconds

# Network Use

You can create a standard network that uses traingd with feedforwardnet or cascadeforwardnet. To prepare a custom network to be trained with traingd,

- Set `net.trainFcn` to 'traingd'. This sets `net.trainParam` to 'traingd' default parameters.

- Set `net.trainParam` properties to desired values.

In either case, calling `train` with the resulting network trains the network with `traingd`.

See `help feedforwardnet` and `help cascadeforwardnet` for examples.

## Definitions

The batch steepest descent training function is `traingd`. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network `trainFcn` to `traingd`, and then call the function `train`. There is only one training function associated

with a given network.

There are seven training parameters associated with `traingd`:

- `epochs`
- `show`
- `goal`
- `time`
- `min_grad`
- `max_fail`
- `lr`

The learning rate `lr` is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger

the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge. See page 12-8 of [[HDB96](#)] for a discussion of the choice of learning rate.

The training status is displayed for every `show` iterations of the algorithm. (If `show` is set to `NaN`, then the training status is never displayed.) The other parameters determine when the training stops. The training stops if the number of iterations exceeds `epochs`, if the performance function drops below `goal`, if the magnitude of the gradient is less than `mingrad`, or if the training time is longer than `time` seconds. `max_fail`,

which is associated with the early stopping technique, is discussed in [Improving Generalization](#).

The following code creates a training set of inputs  $p$  and targets  $t$ . For batch training, all the input vectors are placed in one matrix.

```
p = [-1 -1 2 2; 0 5  
0 5];
```

```
t = [-1 -1 1 1];
```

Create the feedforward network.

```
net =  
feedforwardnet(3, 'tr.  
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.epoch  
= 300;
```

```
net.trainParam.goal  
= 1e-5;
```

If you want to use the default training parameters, the preceding commands are not necessary.

Now you are ready to train the network.

```
[net, tr] =
```

```
train(net,p,t);
```

The training record `tr` contains information about the progress of training.

Now you can simulate the trained network to obtain its response to the inputs in the training set.

```
a = net(p)
```

```
a =
```

-1.0026

-0.9962 1.0010

0 . 9960

Try the *Neural Network Design* demonstration nnd12sd1 [[HDB9](#)] for an illustration of the performance of the batch gradient descent algorithm.

## Algorithms

`traingd` can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance `perf` with respect to the weight and bias variables `x`. Each variable is adjusted according to gradient descent:

$$dX = lr * dperf/dX$$

Training stops when any of these conditions occurs:

- The maximum number of epochs (repetitions) is reached.
- The maximum amount of time is exceeded.
- Performance is minimized to the goal.
- The performance gradient falls below min\_grad.
- Validation performance has increased more than max\_fail times since the last time it decreased (when using

validation).

# **Chapter 11**

**ANALYZE AND DEPLOY  
TRAINED NEURAL  
NETWORK**

---

# 11.1 ANALYZE NEURAL NETWORK PERFORMANCE

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

`tr`

tr =

struct with fields:

trainFcn: 'trainlm'

trainParam: [1×1 struct]

performFcn: 'mse'

performParam: [1×1 struct]

derivFcn: 'defaultderiv'

divideFcn: 'dividerand'

divideMode: 'sample'

divideParam: [1×1 struct]

trainInd: [1×354 double]

valInd: [1×76 double]

testInd: [1×76 double]

stop: 'Validation stop.'

num\_epochs: 12

```
trainMask: {[1×506 double]}\nvalMask: {[1×506 double]}\ntestMask: {[1×506 double]}\nbest_epoch: 6\ngoal: 0\nstates: {1×8 cell}\nepoch: [0 1 2 3 4 5 6 7 8 9 10 11]\n12]
```

```
time: [1×13 double]\nperf: [1×13 double]\nvperf: [1×13 double]\ntperf: [1×13 double]\nmu: [1×13 double]\ngradient: [1×13 double]\nval_fail: [0 0 0 0 0 1 0 1 2 3 4 5 6]\nbest_perf: 7.0111\nbest_vperf: 10.3333
```

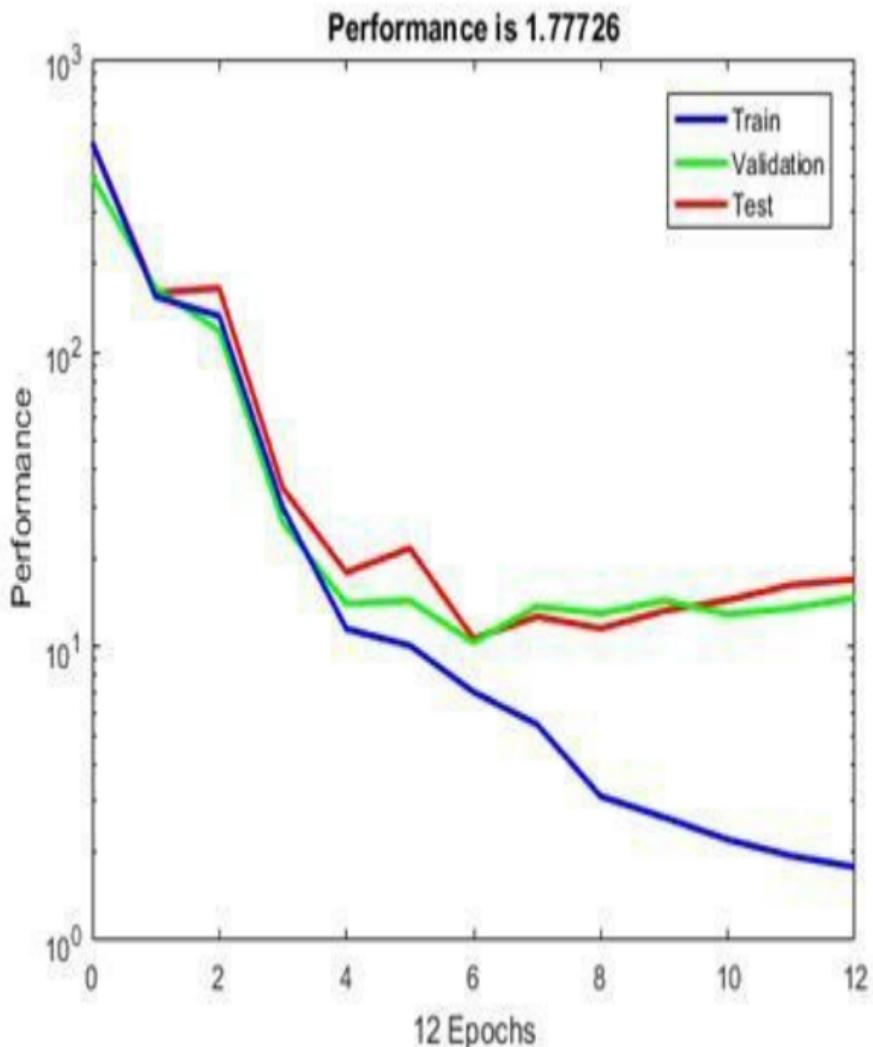
best\_tperf: 10.6567

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

The tr structure also keeps track of several variables during the course of training, such as the value of the performance function, the magnitude of the gradient, etc. You can use the training

record to plot the performance progress by using the `plotperf` command:

**`plotperf(tr)`**



The property `tr.best_epoch` indicates the iteration at which the validation

performance reached a minimum. The training continued for 6 more iterations before the training stopped.

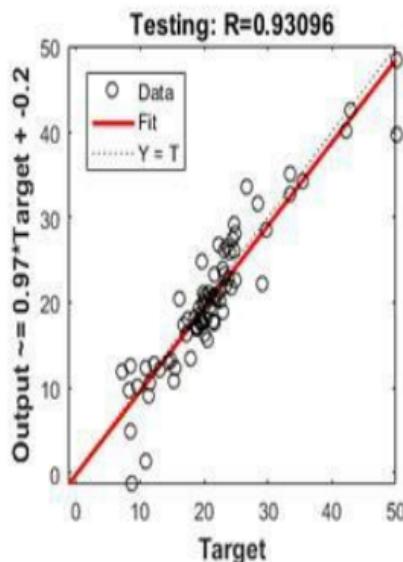
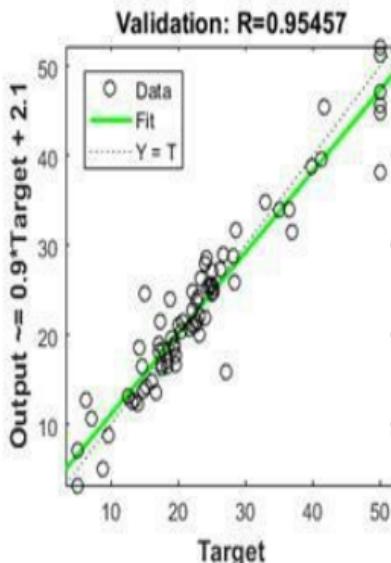
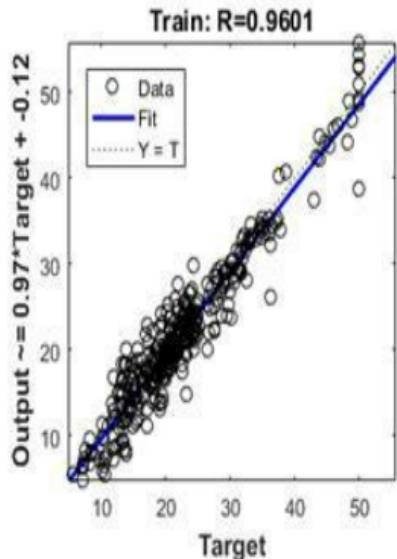
This figure does not indicate any major problems with the training. The validation and test curves are very similar. If the test curve had increased significantly before the validation curve increased, then it is possible that some overfitting might have occurred.

The next step in validating the network is to create a regression plot, which shows the relationship between the outputs of the network and the targets. If the training were perfect, the network outputs and the targets would be exactly equal, but the relationship is rarely

perfect in practice. For the housing example, we can create a regression plot with the following commands. The first command calculates the trained network response to all of the inputs in the data set. The following six commands extract the outputs and targets that belong to the training, validation and test subsets. The final command creates three regression plots for training, testing and validation.

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

```
plotregression(trTarg,trOut,'Train',vTarg,  
tsTarg,tsOut,'Testing')
```



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

For this example, the training data indicates a good fit. The validation and test results also show R values that greater than 0.9. The scatter plot is

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

## 11.2 IMPROVING RESULTS

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

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

```
net =  
train(net, houseInput,
```

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.

# **11.3 DEPLOYMENT FUNCTIONS AND TOOLS FOR TRAINED NETWORKS**

The function [genFunction](#) allows stand-alone MATLAB® functions for a trained neural network. The generated code contains all the information needed to simulate a neural network, including settings, weight and bias values, module functions, and calculations.

The generated MATLAB function can be used to inspect the exact simulation calculations that a particular neural network performs, and makes it easier to

deploy neural networks for many purposes with a wide variety of MATLAB deployment products and tools.

The function `genFunction` is introduced in the deployment panels in the tools [nftool](#), [nctool](#), [nprtool](#) and [ntstool](#). For information on these tool features, see [Fit Data with a Neural Network](#), [Classify Patterns with a Neural Network](#), [Cluster Data with a Self-Organizing Map](#), and [Neural Network Time-Series Prediction and Modeling](#).



## Deploy Solution

Generate deployable versions of your trained neural network.

### Application Deployment

Prepare neural network for deployment with MATLAB Compiler and Builder tools.

- ▶ Generate a stand-alone MATLAB function:

(genFunction)



### Code Generation

Prepare neural network for deployment with MATLAB Coder tools.

- ▶ Generate a MATLAB function with matrix-only arguments (no cell array support):

(genFunction)



### Simulink Deployment

Simulate neural network in Simulink or deploy with Simulink Coder tools.

- ▶ Generate a Simulink diagram:

(gensim)



### Graphics

- ▶ Generate a graphical diagram of the neural network:

(networkView)



Deploy a neural network or click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

The advanced scripts generated on the

Save Results panel of each of these tools includes an example of deploying networks with genFunction.

# 11.4 GENERATE NEURAL NETWORK FUNCTIONS FOR APPLICATION DEPLOYMENT

The function [genFunction](#) generates a stand-alone MATLAB function for simulating any trained neural network and preparing it for deployment. This might be useful for several tasks:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network

- Use the MATLAB Function block to create a Simulink® block
- Use MATLAB Compiler™ to:
  - o Generate stand-alone executables
  - o Generate Excel® add-ins
- Use MATLAB Compiler SDK™ to:
  - o Generate C/C++ libraries
  - o Generate .COM components
  - o Generate Java® components
  - o Generate .NET components
- Use MATLAB Coder™ to:
  - o Generate C/C++ code
  - o Generate efficient MEX-

## functions

`genFunction(net,'pathname')` takes a neural network and file path, and produces a standalone MATLAB function file `filename.m`.

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

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

Here a static network is trained and its outputs calculated.

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

```
houseNet = feedforwardnet(10);
```

```
houseNet = train(houseNet,x,t);
```

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

The following code generates, tests, and displays a MATLAB function with the same interface as the neural network object.

```
genFunction(houseNet,'houseFcn');
```

```
y2 = houseFcn(x);
```

```
accuracy2 = max(abs(y-y2))
```

```
edit houseFcn
```

You can compile the new function with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with mcc.

```
mcc -W lib:libHouse -T link:lib  
houseFcn
```

The next code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool codegen (license required), which is also tested.

```
genFunction(houseNet,'houseFcn','Matrix'
y3 = houseFcn(x);
accuracy3 = max(abs(y-y3))
```

```
x1Type = coder.typeof(double(0),[13
Inf]); % Coder type of input 1
codegen houseFcn.m -config:mex -o
houseCodeGen -args {x1Type}
y4 = houseCodeGen(x);
accuracy4 = max(abs(y-y4))
```

Here a dynamic network is trained and its outputs calculated.

```
[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = prepares(maglevNet,x,
{},t);
maglevNet =
```

```
train(maglevNet,X,T,Xi,Ai);  
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next a MATLAB function is generated and tested. The function is then used to create a shared/dynamically linked library with mcc.

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

The following code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is

used to generate a MEX-function with the MATLAB Coder tool codegen, which is also tested.

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

x1Type      = coder.typeof(double(0),[1
Inf]); % Coder type of input 1
x2Type      = coder.typeof(double(0),[1
Inf]); % Coder type of input 2
```

```
xi1Type = coder.typeof(double(0),[1 2]);  
% Coder type of input 1 states  
xi2Type = coder.typeof(double(0),[1 2]);  
% Coder type of input 2 states  
codegen maglevFcn.m -config:mex -o  
maglevNetCodeGen ...  
          -args {x1Type x2Type  
xi1Type xi2Type}  
[y4,xf1,xf2] =  
maglevNetCodeGen(x1,x2,xi1,xi2);  
dynamic_codegen_accuracy =  
max(abs(cell2mat(y)-y4))
```

# 11.5 DEPLOY NEURAL NETWORK SIMULINK DIAGRAMS

The function `gensim` generates block descriptions of networks so you can simulate them using Simulink® software.

`gensim(net, st)`

The second argument to `gensim` determines the sample time, which is normally chosen to be some positive real value.

If a network has no delays associated

with its input weights or layer weights, this value can be set to -1. A value of -1 causes gensim to generate a network with continuous sampling.

## 11.5.1 Example

Here is a simple problem defining a set of inputs  $p$  and corresponding targets  $t$ .

```
p = [1 2 3 4 5];
```

```
t = [1 3 5 7 9];
```

The code below designs a linear layer to solve this problem.

```
net = newlind(p, t)
```

You can test the network on your original inputs with `sim`.

`y = sim(net, p)`

The results show the network has solved the problem.

`y =`

1.0000

3.0000 5.0000

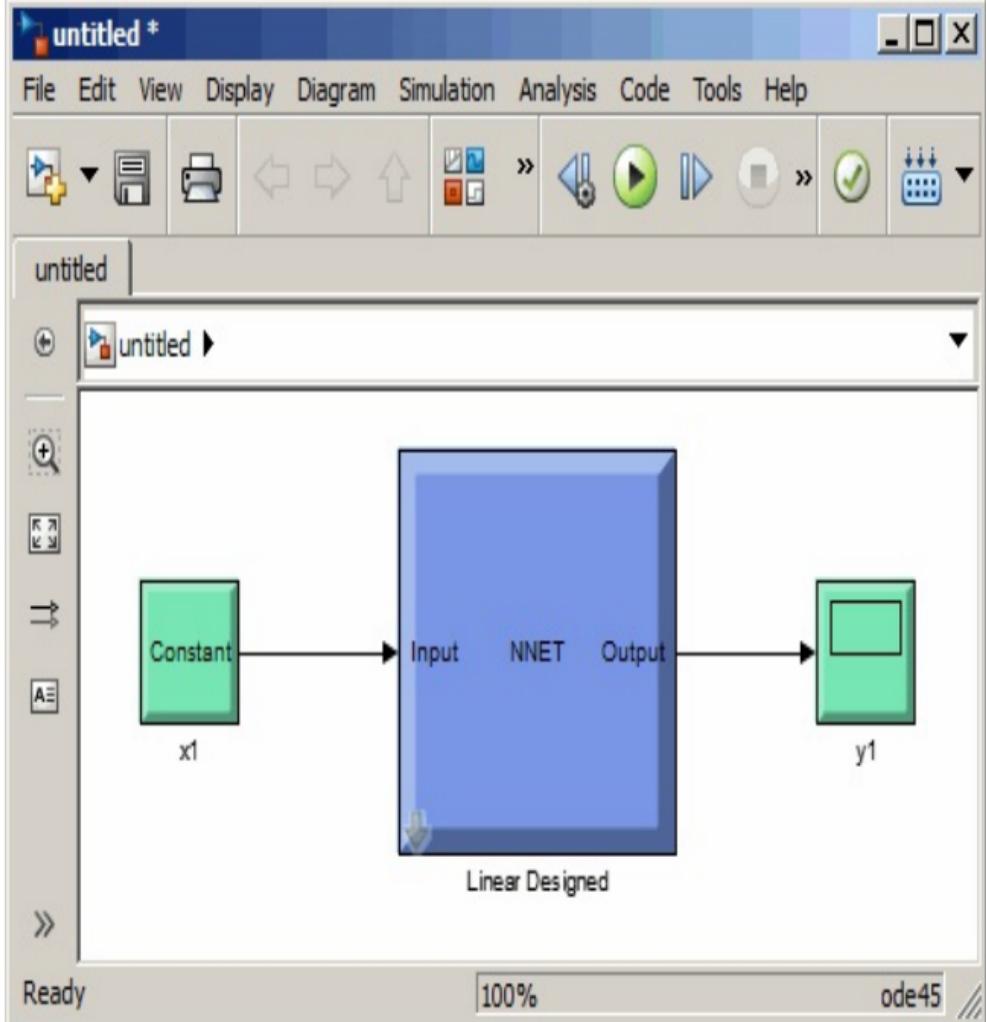
7.0000 9.0000

Call `gensim` as follows to generate a Simulink version of the network.

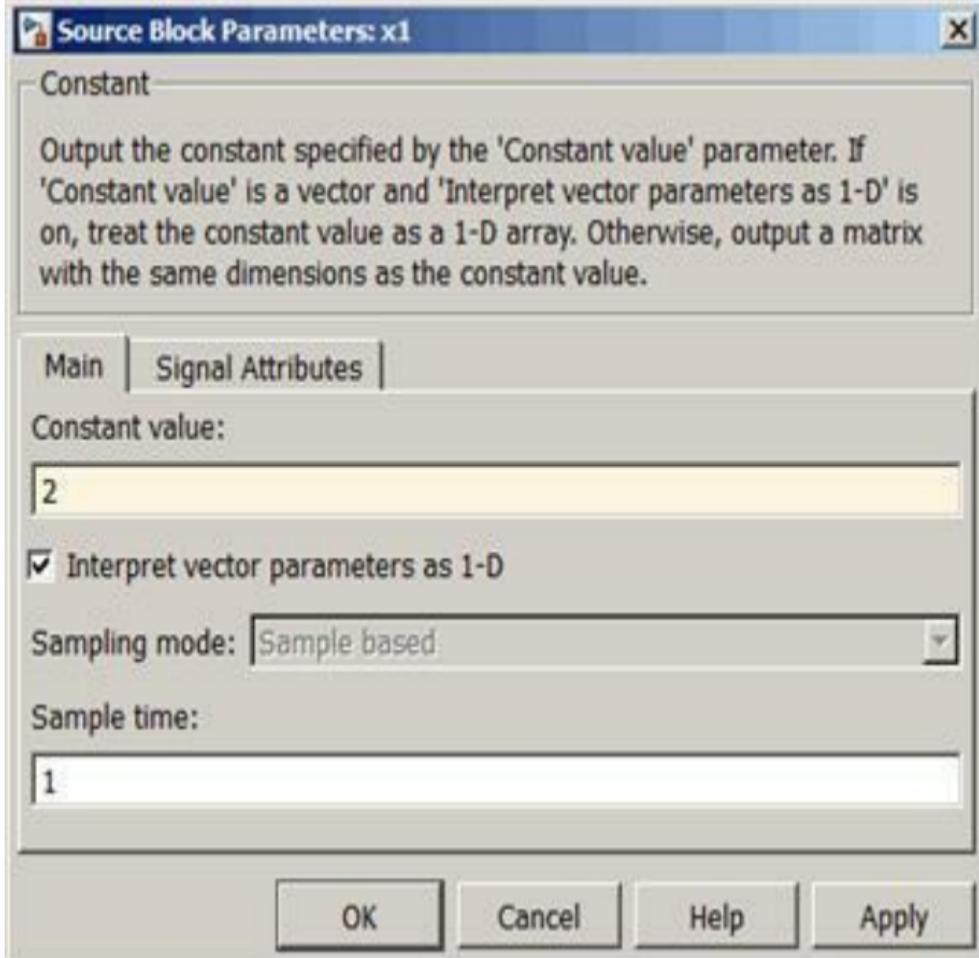
```
gensim(net, -1)
```

The second argument is -1, so the resulting network block samples continuously.

The call to `gensim` opens the following Simulink Editor, showing a system consisting of the linear network connected to a sample input and a scope.



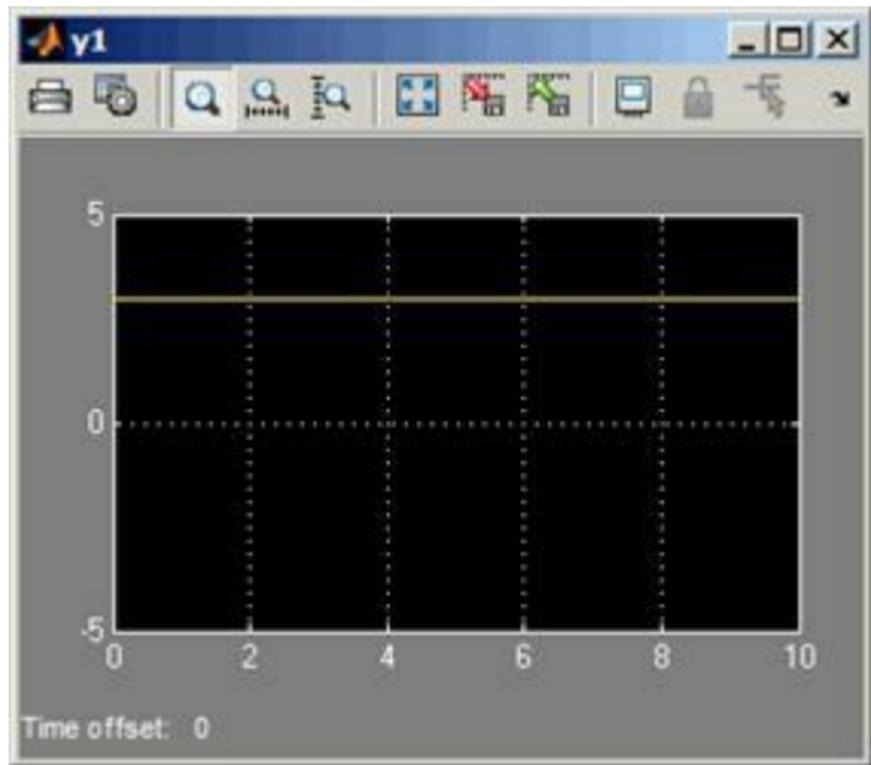
To test the network, double-click the input Constant  $x_1$  block on the left.



The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to 2, and then click **OK**.

Select the menu option **Simulation > Run**. Simulink takes a moment to simulate the system.

When the simulation is complete, double-click the output  $y_1$  block on the right to see the following display of the network's response.



Note that the output is 3, which is the correct output for an input of 2.

## **11.5.2 Suggested Exercises**

Here are a couple exercises you can try.

### **11.5.2.1.1 Change the Input Signal**

Replace the constant input block with a signal generator from the standard Simulink Sources blockset. Simulate the system and view the network's response.

### **11.5.2.1.2 Use a Discrete Sample Time**

Recreate the network, but with a discrete sample time of 0.5, instead of continuous sampling.

```
gensim(net, 0.5)
```

Again, replace the constant input with a signal generator. Simulate the system and view the network's response.

# 11.6 DEPLOY TRAINING OF NEURAL NETWORKS

Use MATLAB® Runtime to deploy functions that can train a model. You can deploy MATLAB code that trains neural networks as described in [Create Standalone Application from Command Line](#) and [Package Standalone Application with Application Compiler App](#).

The following methods and functions are NOT supported in deployed mode:

- Training progress

dialog, [nntraintool](#).

- [genFunction](#) and [gensim](#) to generate MATLAB code or Simulink® blocks
- `view` method
- 
- `nctool`, `nftool`, `nnstart`, `nprtool`,
- Plot functions (such as [plotperform](#), [plottrainstate](#), [p](#) and so on)
- `perceptron`, `newlind`, `elmannet`, and `newhop` functions

Here is an example of how you can deploy training of a network. Create a script to train a neural network, for

example, mynntraining.m:

```
% Create the predictor and response (target)  
  
x = [0.054 0.78 0.13  
0.47 0.34 0.79 0.53  
0.6 0.65 0.75 0.084  
0.91 0.83  
0.53 0.93 0.57  
0.012 0.16 0.31 0.17
```

```
0.26 0.69 0.45 0.23  
0.15 0.54] ;  
  
t = [0.46 0.079 0.42  
0.48 0.95 0.63 0.48  
0.51 0.16 0.51 1  
0.28 0.3] ;  
  
% Create and display  
the network  
  
net = fitnet();
```

```
disp('Training  
fitnet')  
  
% Train the network  
using the data in x  
and t  
  
net =  
train(net,x,t);  
  
% Predict the  
responses using the  
trained network
```

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

```
% Measure the  
performance
```

```
perf =  
perform(net, y, t)
```

Compile the script `mynntraining.m`, either by using the MATLAB Compiler™ interface as described in [Package Standalone Application with Application Compiler App](#), or by using

the command line:

```
mcc -m
```

```
'mynntraining.m'
```

`mcc` invokes the MATLAB Compiler to compile code at the prompt. The flag `-m` compiles a MATLAB function and generates a standalone executable. The EXE file is now in your local computer in the working directory.

To run the compiled EXE application on computers that do not have MATLAB installed, you need to download and

install MATLAB Runtime.  
The `readme.txt` created in your working folder has more information about the deployment requirements.

# **Chapter 12**

## **TRAINING SCALABILITY AND EFFICIENCY**

---

# **12.1 NEURAL NETWORKS WITH PARALLEL AND GPU COMPUTING**

## 12.1.1 Modes of Parallelism

Neural networks are inherently parallel algorithms. Multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs can take advantage of this parallelism.

Parallel Computing Toolbox™, when used in conjunction with Neural Network Toolbox™, enables neural network training and simulation to take advantage of each mode of parallelism.

For example, the following shows a standard single-threaded training and simulation session:

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

The two steps you can parallelize in this session are the call to [train](#) and the implicit call to [sim](#) (where the network net2 is called as a function).

In Neural Network Toolbox you can divide any data, such as x and t in the previous example code, across samples. If x and t contain only one sample each, there is no parallelism. But if x and t contain hundreds or thousands of samples, parallelism can provide both speed and problem size benefits.

# 1    2    .1    .2 Distributed Computing

Parallel Computing Toolbox allows neural network training and simulation to run across multiple CPU cores on a single PC, or across multiple CPUs on multiple computers on a network using MATLAB® Distributed Computing Server™.

Using multiple cores can speed calculations. Using multiple computers can allow you to solve problems using data sets too big to fit in the RAM of a single computer. The only limit to problem size is the total quantity of

RAM available across all computers.

To manage cluster configurations, use the Cluster Profile Manager from the **MATLAB Home tab Environment** menu **Cluster Profiles**.

To open a pool of MATLAB workers using the default cluster profile, which is usually the local CPU cores, use this command:

```
pool = parpool
```

Starting parallel pool (parpool) using  
the 'local' profile ... connected to 4  
workers.

When parpool runs, it displays the  
number of workers available in the pool.

Another way to determine the number of workers is to query the pool:

pool.NumWorkers

4

Now you can train and simulate the neural network with data split by sample across all the workers. To do this, set the train and sim parameter 'useParallel' to 'yes'.

```
net2 = train(net1,x,t,'useParallel','yes')  
y = net2(x,'useParallel','yes')
```

Use the 'showResources' argument to verify that the calculations ran across multiple workers.

net2

=

```
train(net1,x,t,'useParallel','yes','showRes  
y' =
```

```
net2(x,'useParallel','yes','showResources'
```

MATLAB indicates which resources were used. For example:

Computing Resources:

Parallel Workers

Worker 1 on MyComputer, MEX on PCWIN64

Worker 2 on MyComputer, MEX on PCWIN64

Worker 3 on MyComputer, MEX on PCWIN64

Worker 4 on MyComputer, MEX on PCWIN64

When train and sim are called, they divide the input matrix or cell array data into distributed Composite values before training and simulation. When sim has calculated a Composite, this output is converted back to the same matrix or cell array form before it is returned.

However, you might want to perform this data division manually if:

- The problem size is too large for the host computer. Manually defining the elements of Composite values sequentially allows much bigger problems to be defined.
- It is known that some workers are on computers that are faster or have

more memory than others. You can distribute the data with differing numbers of samples per worker. This is called load balancing.

The following code sequentially creates a series of random datasets and saves them to separate files:

```
pool = gcp;
for i=1:pool.NumWorkers
    x = rand(2,1000);
    save(['inputs' num2str(i)],'x');
    t = x(1,:).*.x(2,:)+2*(x(1,:)+x(2,:));
    save(['targets' num2str(i)],'t');
    clear x t
end
```

Because the data was defined sequentially, you can define a total dataset larger than can fit in the host PC memory. PC memory must accommodate only a sub-dataset at a time.

Now you can load the datasets sequentially across parallel workers, and train and simulate a network on the Composite data. When train or sim is called with Composite data, the 'useParallel' argument is automatically set to 'yes'. When using Composite data, configure the network's input and outputs to match one of the datasets manually using the configure function before training.

`xc = Composite;`

```
tc = Composite;  
for i=1:pool.NumWorkers  
    data = load(['inputs' num2str(i)],'x');  
    xc{i} = data.x;  
    data = load(['targets' num2str(i)],'t');  
    tc{i} = data.t;  
    clear data  
end  
net2 = configure(net1,xc{1},tc{1});  
net2 = train(net2,xc,tc);  
yc = net2(xc);
```

To convert the Composite output returned by sim, you can access each of its elements, separately if concerned about memory limitations.

```
for i=1:pool.NumWorkers  
    yi = yc{i}
```

end

Combined the Composite value into one local value if you are not concerned about memory limitations.

```
y = {yc{:}};
```

When load balancing, the same process happens, but, instead of each dataset having the same number of samples (1000 in the previous example), the numbers of samples can be adjusted to best take advantage of the memory and speed differences of the worker host computers.

It is not required that each worker have data. If element  $i$  of a Composite value is undefined, worker  $i$  will not be used

in the computation.

# 1 2 .1 .3 Single Computing

GPU

The number of cores, size of memory, and speed efficiencies of GPU cards are growing rapidly with each new generation. Where video games have long benefited from improved GPU performance, these cards are now flexible enough to perform general numerical computing tasks like training neural networks.

For the latest GPU requirements, see the web page for Parallel Computing Toolbox; or query MATLAB to determine whether your PC has a

supported GPU. This function returns the number of GPUs in your system:

```
count = gpuDeviceCount
```

```
count =
```

```
1
```

If the result is one or more, you can query each GPU by index for its characteristics. This includes its name, number of multiprocessors, SIMDWidth of each multiprocessor, and total memory.

```
gpu1 = gpuDevice(1)
```

```
gpu1 =
```

CUDADevice with properties:

Name: 'GeForce GTX

470'

Index: 1

ComputeCapability: '2.0'

SupportsDouble: 1

DriverVersion: 4.1000

MaxThreadsPerBlock: 1024

MaxShmemPerBlock: 49152

MaxThreadBlockSize: [1024 1024

64]

MaxGridSize: [65535 65535

1]

SIMDWidth: 32

TotalMemory: 1.3422e+09

AvailableMemory: 1.1056e+09

```
MultiprocessorCount: 14
ClockRateKHz: 1215000
ComputeMode: 'Default'
GPUOverlapsTransfers: 1
KernelExecutionTimeout: 1
CanMapHostMemory: 1
DeviceSupported: 1
DeviceSelected: 1
```

The simplest way to take advantage of the GPU is to specify call train and sim with the parameter argument 'useGPU' set to 'yes' ('no' is the default).

```
net2 = train(net1,x,t,'useGPU','yes')
y = net2(x,'useGPU','yes')
```

If net1 has the default training

function trainlm, you see a warning that GPU calculations do not support Jacobian training, only gradient training. So the training function is automatically changed to the gradient training function trainscg. To avoid the notice, you can specify the function before training:

```
net1.trainFcn = 'trainscg';
```

To verify that the training and simulation occur on the GPU device, request that the computer resources be shown:

```
net2 =  
train(net1,x,t,'useGPU','yes','showResour  
y =  
net2(x,'useGPU','yes','showResources','y
```

Each of the above lines of code outputs the following resources summary:

Computing Resources:

GPU device #1, GeForce GTX 470

Many MATLAB functions automatically execute on a GPU when any of the input arguments is a gpuArray. Normally you move arrays to and from the GPU with the functions gpuArray and gather. However, for neural network calculations on a GPU to be efficient, matrices need to be transposed and the columns padded so that the first element in each column aligns properly in the GPU memory. Neural Network Toolbox provides a special function called [ndata2gpu](#) to move an array to a

GPU and properly organize it:

```
xg = nnData2gpu(x);
```

```
tg = nnData2gpu(t);
```

Now you can train and simulate the network using the converted data already on the GPU, without having to specify the 'useGPU' argument. Then convert and return the resulting GPU array back to MATLAB with the complementary function [gpu2nnData](#).

Before training with gpuArray data, the network's input and outputs must be manually configured with regular MATLAB matrices using the `configure` function:

```
net2 = configure(net1,x,t); % Configure
```

with MATLAB arrays

```
net2 = train(net2,xg,tg); % Execute on  
GPU with NNET formatted gpuArrays  
yg = net2(xg); % Execute on  
GPU  
y = gpu2nndata(yg); % Transfer  
array to local workspace
```

On GPUs and other hardware where you might want to deploy your neural networks, it is often the case that the exponential function `exp` is not implemented with hardware, but with a software library. This can slow down neural networks that use the `tansig` sigmoid transfer function. An alternative function is the Elliot sigmoid function whose expression does not

include a call to any higher order functions:

$$\text{(equation)} \quad a = n / (1 + \text{abs}(n))$$

Before training, the network's tansig layers can be converted to elliotsig layers as follows:

for i=1:net.numLayers

if

strcmp(net.layers{i}.transferFcn,'tansig')

    net.layers{i}.transferFcn = 'elliotsig';

end

end

Now training and simulation might be faster on the GPU and simpler deployment hardware.

# 1 2 .1 .4 Distributed GPU Computing

Distributed and GPU computing can be combined to run calculations across multiple CPUs and/or GPUs on a single computer, or on a cluster with MATLAB Distributed Computing Server.

The simplest way to do this is to specify train and sim to do so, using the parallel pool determined by the cluster profile you use. The 'showResources' option is especially recommended in this case, to verify that the expected hardware is being employed:

```
net2 =  
train(net1,x,t,'useParallel','yes','useGPU',  
y =  
net2(x,'useParallel','yes','useGPU','yes','s
```

These lines of code use all available workers in the parallel pool. One worker for each unique GPU employs that GPU, while other workers operate as CPUs. In some cases, it might be faster to use only GPUs. For instance, if a single computer has three GPUs and four workers each, the three workers that are accelerated by the three GPUs might be speed limited by the fourth CPU worker. In these cases, you can specify that train and sim use only workers with unique GPUs.

```
net2 =  
train(net1,x,t,'useParallel','yes','useGPU',  
y =  
net2(x,'useParallel','yes','useGPU','only','
```

As with simple distributed computing, distributed GPU computing can benefit from manually created Composite values. Defining the Composite values yourself lets you indicate which workers to use, how many samples to assign to each worker, and which workers use GPUs.

For instance, if you have four workers and only three GPUs, you can define larger datasets for the GPU workers. Here, a random dataset is created with different sample loads per Composite

element:

```
numSamples = [1000 1000 1000 300];
xc = Composite;
tc = Composite;
for i=1:4
    xi = rand(2,numSamples(i));
    ti = xi(1,:).^2 + 3*xi(2,:);
    xc{i} = xi;
    tc{i} = ti;
end
```

You can now specify that train and sim use the three GPUs available:

```
net2 = configure(net1,xc{1},tc{1});
net2 =
train(net2,xc,tc,'useGPU','yes','showReso
```

```
yc = net2(xc,'showResources','yes');
```

To ensure that the GPUs get used by the first three workers, manually converting each worker's Composite elements to gpuArrays. Each worker performs this transformation within a parallel executing spmd block.

```
spmd
```

```
    if labindex <= 3
```

```
        xc = nnData2GPU(xc);
```

```
        tc = nnData2GPU(tc);
```

```
    end
```

```
end
```

Now the data specifies when to use GPUs, so you do not need to tell train and sim to do so.

```
net2 = configure(net1,xc{1},tc{1});  
net2  
=  
train(net2,xc,tc,'showResources','yes');  
yc = net2(xc,'showResources','yes');
```

Ensure that each GPU is used by only one worker, so that the computations are most efficient. If multiple workers assign gpuArray data on the same GPU, the computation will still work but will be slower, because the GPU will operate on the multiple workers' data sequentially.

## 12.1.5 Deep Learning

Training a convolutional neural network (CNN, ConvNet) requires the Parallel Computing Toolbox and a CUDA®-enabled NVIDIA® GPU with compute capability 3.0 or higher. You have the option to choose the execution environment (CPU or GPU) for extracting features, predicting responses, or classifying observations (see [activations](#), [predict](#), and [classify](#)).

## 12.1.6 Parallel Time Series

For time series networks, simply use cell array values for x and t, and optionally include initial input delay states xi and initial layer delay states ai, as required.

```
net2 = train(net1,x,t,xi,ai,'useGPU','yes')  
y = net2(x,xi,ai,'useParallel','yes','useGPU','y')
```

```
net2 = train(net1,x,t,xi,ai,'useParallel','yes')  
y = net2(x,xi,ai,'useParallel','yes','useGPU','c')
```

```
net2 =
```

```
train(net1,x,t,xi,ai,'useParallel','yes','useC  
y  
=
```

```
net2(x,xi,ai,'useParallel','yes','useGPU','c
```

Note that parallelism happens across samples, or in the case of time series across different series. However, if the network has only input delays, with no layer delays, the delayed inputs can be precalculated so that for the purposes of computation, the time steps become different samples and can be parallelized. This is the case for networks such as timedelaynet and open-loop versions of narxnet and narnet. If a network has layer delays, then time cannot be "flattened" for purposes of computation, and so single series data

cannot be parallelized. This is the case for networks such as layrecnet and closed-loop versions of narxnet and narnet. However, if the data consists of multiple sequences, it can be parallelized across the separate sequences.

## 12.1.7 Parallel Availability, Fallbacks, and Feedback

As mentioned previously, you can query MATLAB to discover the current parallel resources that are available.

To see what GPUs are available on the host computer:

```
gpuCount = gpuDeviceCount  
for i=1:gpuCount  
    gpuDevice(i)  
end
```

To see how many workers are running in the current parallel pool:

```
poolSize = pool.NumWorkers
```

To see the GPUs available across a parallel pool running on a PC cluster using MATLAB Distributed Computing Server:

```
spmd
    worker.index = labindex;
    worker.name = system('hostname');
    worker.gpuCount = gpuDeviceCount;
    try
        worker.gpuInfo = gpuDevice;
    catch
        worker.gpuInfo = [];
    end
    worker
end
```

When 'useParallel' or 'useGPU' are set to 'yes', but parallel or GPU workers are

unavailable, the convention is that when resources are requested, they are used if available. The computation is performed without error even if they are not. This process of falling back from requested resources to actual resources happens as follows:

- If 'useParallel' is 'yes' but Parallel Computing Toolbox is unavailable, or a parallel pool is not open, then computation reverts to single-threaded MATLAB.
- If 'useGPU' is 'yes' but the gpuDevice for the current MATLAB session is unassigned or not supported, then computation reverts to the CPU.

If 'useParallel' and 'useGPU' are 'yes', then each worker with a unique GPU uses that GPU, and other workers revert to CPU.

If 'useParallel' is 'yes' and 'useGPU' is then workers with unique GPUs are used. Other workers are not used, unless no workers have GPUs. In the case with no GPUs, all workers use CPUs.

When unsure about what hardware is actually being employed, check `gpuDeviceCount`, `gpuDevice`, and `pool.NumWorkers` to ensure the desired hardware is available, and

call train and sim with 'showResources' set to 'yes' to verify what resources were actually used.

## 12.2 AUTOMATICALLY SAVE CHECKPOINTS DURING NEURAL NETWORK TRAINING

During neural network training, intermediate results can be periodically saved to a MAT file for recovery if the computer fails or you kill the training process. This helps protect the value of long training runs, which if interrupted would need to be completely restarted otherwise. This feature is especially useful for long parallel training sessions, which are more likely to be interrupted by computing resource failures.

Checkpoint saves are enabled with the optional 'CheckpointFile' training argument followed by the checkpoint file name or path. If you specify only a file name, the file is placed in the working directory by default. The file must have the .mat file extension, but if this is not specified it is automatically appended. In this example, checkpoint saves are made to the file called MyCheckpoint.mat in the current working directory.

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

```
net =
feedforwardnet(10);

net2 =
train(net,x,t,'Check']
```

22-Mar-2013 04:49:05  
First Checkpoint #1:  
/WorkingDir/MyCheckp

22-Mar-2013 04:49:06  
Final Checkpoint #2:  
/WorkingDir/MyCheckp

By default, checkpoint saves occur at most once every 60 seconds. For the previous short training example, this results in only two checkpoint saves: one at the beginning and one at the end of training.

The optional training argument '`'CheckpointDelay'`' can change the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds for a time-series problem where a neural network is trained to model a levitated magnet.

[ $x, t$ ] =

```
maglev_dataset;
```

```
net =  
narxnet(1:2,1:2,10);
```

```
[X,Xi,Ai,T] =  
preparets(net,x,  
{ },t);
```

```
net2 =  
train(net,X,T,Xi,Ai,
```

22-Mar-2013 04:59:28

First Checkpoint #1:  
/WorkingDir/MyCheckp

22-Mar-2013 04:59:38  
Write Checkpoint #2:  
/WorkingDir/MyCheckp

22-Mar-2013 04:59:48  
Write Checkpoint #3:  
/WorkingDir/MyCheckp

22-Mar-2013 04:59:58  
Write Checkpoint #4:

/WorkingDir/MyCheckp

22-Mar-2013 05:00:08

Write Checkpoint #5:

/WorkingDir/MyCheckp

22-Mar-2013 05:00:09

Final Checkpoint #6:

/WorkingDir/MyCheckp

After a computer failure or training interruption, you can reload the checkpoint structure containing the best neural network obtained before the

interruption, and the training record. In this case, the stage field value is 'Final', indicating the last save was at the final epoch because training completed successfully. The first epoch checkpoint is indicated by 'First', and intermediate checkpoints by 'write'.

```
load('MyCheckpoint.m');
checkpoint =
```

file:

'/WorkingDir/MyCheck'

time: [2013 3  
22 5 0 9.0712]

number: 6

stage: 'Final'

net: [1x1  
network]

```
tr: [1x1  
struct]
```

You can resume training from the last checkpoint by reloading the dataset (if necessary), then calling train with the recovered network.

```
net =  
checkpoint.net;
```

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

```
load('MyCheckpoint.m');
[X,Xi,Ai,T] =
preparets(net,x,
{ },t);
net2 =
train(net,X,T,Xi,Ai,
```

## **12.3 OPTIMIZE NEURAL NETWORK TRAINING SPEED AND MEMORY**

## 12.3.1 Memory Reduction

Depending on the particular neural network, simulation and gradient calculations can occur in MATLAB® or MEX. MEX is more memory efficient, but MATLAB can be made more memory efficient in exchange for time.

To determine whether MATLAB or MEX is being used, use the 'showResources' option, as shown in this general form of the syntax:

```
net2 =  
train(net1,x,t,'show'
```

If MATLAB is being used and memory limitations are a problem, the amount of temporary storage needed can be reduced by a factor of  $N$ , in exchange for performing the computations  $N$  times sequentially on each of  $N$  subsets of the data.

```
net2 =  
train(net1,x,t,'redu')  
This is called memory reduction.
```

## 12.3.2 Fast Elliot Sigmoid

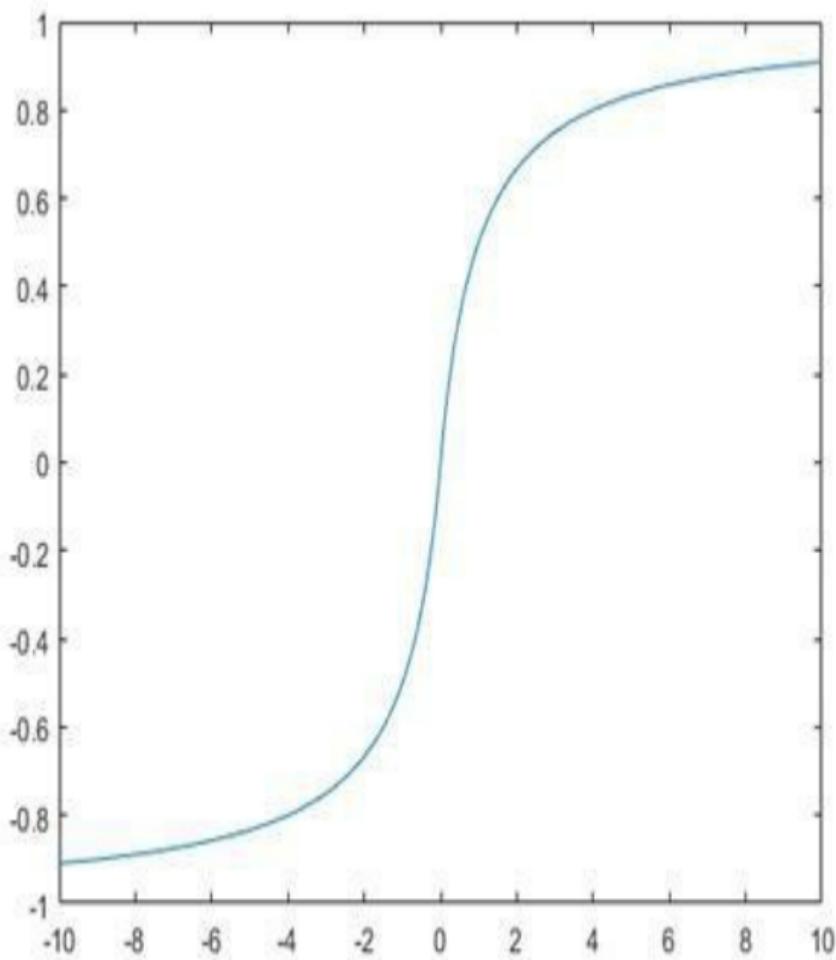
Some simple computing hardware might not support the exponential function directly, and software implementations can be slow. The Elliot sigmoid `elliotsig` function performs the same role as the symmetric sigmoid `tansig` function, but avoids the exponential function.

Here is a plot of the Elliot sigmoid:

```
n = -10:0.01:10;
```

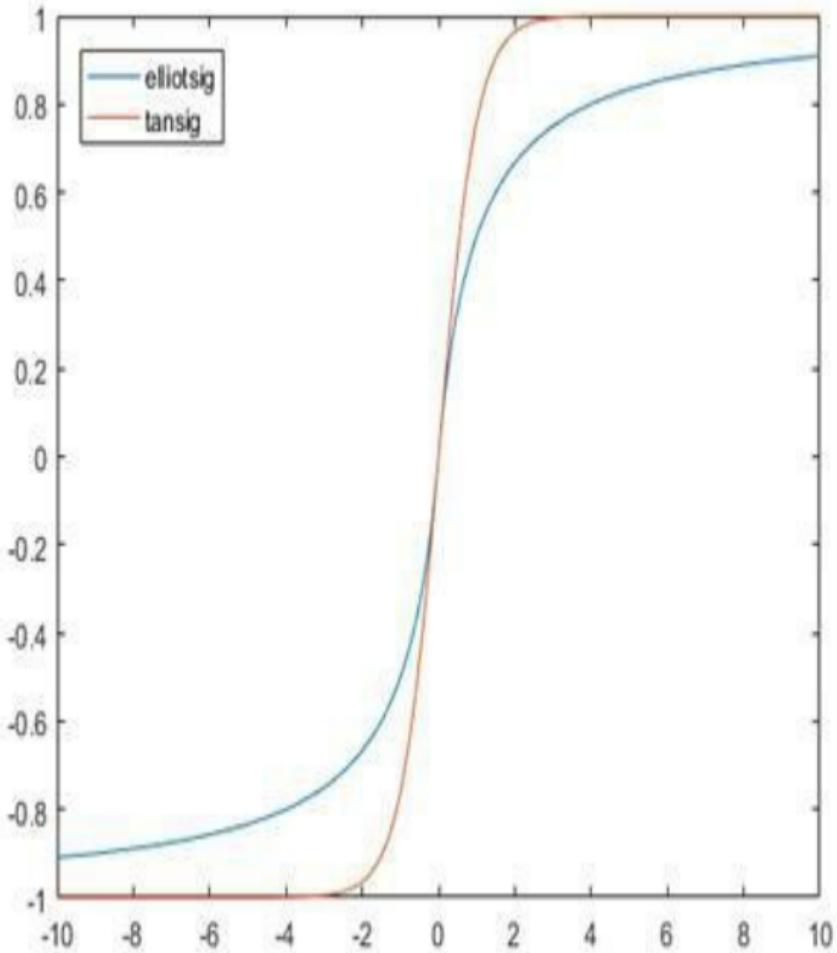
```
a = elliotsig(n);
```

plot(n, a)



Next, elliot sig is compared  
with tansig.

```
a2 = tansig(n);  
  
h = plot(n,a,n,a2);  
  
legend(h, 'elliot sig')
```



To train a neural network using ellotsig instead of tansig,

transform the network's transfer functions:

```
[x, t] =
```

```
house_dataset;
```

```
net =
```

```
feedforwardnet;
```

```
view(net)
```

```
net.layers{1}.transf  
= 'elliotsig';
```

```
view(net)
```

```
net =
```

```
train(net,x,t);
```

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

Here, the times to execute elliotsig and tansig are compared. elliotsig is approximately four times faster on the test system.

```
n = rand(1000,1000);
```

```
tic,for  
i=1:100,a=tansig(n);  
end, tansigTime =  
toc;
```

```
tic,for  
i=1:100,a=elliotsig(:  
end, elliotTime =  
toc;
```

```
speedup = tansigTime  
/ elliotTime
```

speedup =

4.1406

However, while simulation is faster with elliotSig, training is not guaranteed to be faster, due to the different shapes of the two transfer functions. Here, 10 networks are each trained for tansig and elliotSig, but training times vary significantly even on the same problem with the same network.

[x, t] =  
house\_dataset;

```
tansigNet =  
feedforwardnet;
```

```
tansigNet.trainParam  
= false;
```

```
elliotNet =  
tansigNet;
```

```
elliotNet.layers{1}.  
= 'elliotsig';
```

```
for i=1:10, tic, net  
=  
train(tansigNet,x,t)  
tansigTime = toc,  
end
```

```
for i=1:10, tic, net  
=  
train(elliotNet,x,t)  
elliotTime = toc,  
end
```

# **Chapter 13**

## **OPTIMAL SOLUTIONS**

---

# **13.1 REPRESENTING UNKNOWN OR DON'T-CARE TARGETS**

# 1 3 .1 Choose Neural Network Input-Output Processing Functions

This topic presents part of a typical multilayer network workflow.

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

coming into the network is preprocessed in the same way.)

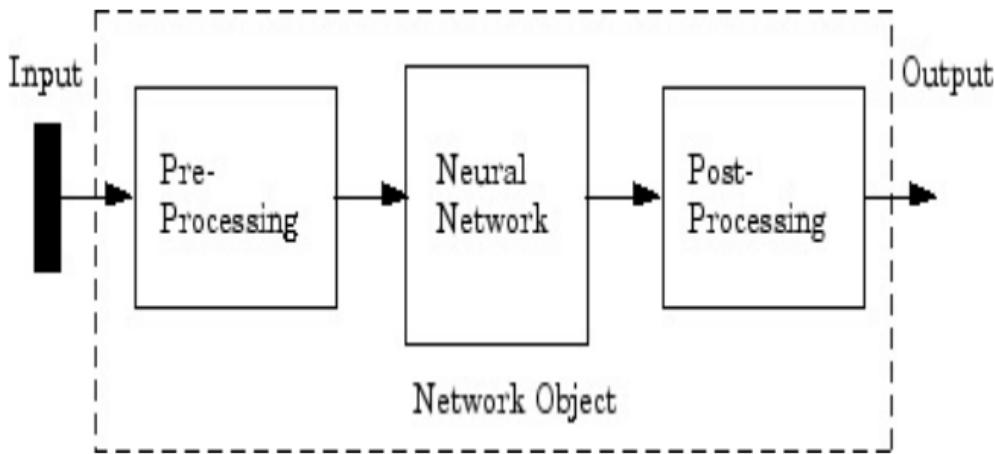
For example, in multilayer networks, sigmoid transfer functions are generally used in the hidden layers. These functions become essentially saturated when the net input is greater than three ( $\exp(-3) \approx 0.05$ ). If this happens at the beginning of the training process, the gradients will be very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

practice to normalize the inputs before applying them to the network.

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

It is easiest to think of the neural network as having a preprocessing block that appears between the input and the first layer of the network and a postprocessing block that appears between the last layer of the network and

the output, as shown in the following figure.



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

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

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

`net.inputs{1}.process`, 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}.process`, where the index 2 refers to the output vector coming from the second layer. (For the feedforward network, there is only one output vector, and it comes from the final layer.) You can use these properties to change the processing functions that you want your network to apply to the inputs and outputs. However, the defaults usually provide excellent performance.

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

function for the network input as follows:

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

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

```
net.outputs{2}.process
```

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 [mapminn](#)

The following table lists the most common preprocessing and postprocessing functions. In most cases, you will not need to use them directly, since the preprocessing steps become part of the network object. When you simulate or train the network, the preprocessing and postprocessing will be done automatically.

Function	Algorithm
<a href="#"><u>mapminmax</u></a>	Normalize inputs/targets to fall in the range [-1, 1]
<a href="#"><u>mapstd</u></a>	Normalize inputs/targets to have zero mean and unit variance
<a href="#"><u>processpca</u></a>	Extract principal components from the inputs
<a href="#"><u>fixunknowns</u></a>	Process unknown inputs
<a href="#"><u>removeconstantrows</u></a>	Remove inputs/targets that are constant

# 1 3 .1 .2 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.

## 13.2 CONFIGURE NEURAL NETWORK INPUTS AND OUTPUTS

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

previously to approximate a sine function, issue the following commands:

```
p = -2:.1:2;
```

```
t = sin(pi*p/2);
```

```
net1 = configure(net,p,t);
```

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

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

target for this network is a scalar.

net1.layerWeights{2,1}

## Neural Network Weight

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

In addition to setting the appropriate

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

```
net1.inputs{1}
```

## Neural Network Input

```
name: 'Input'
```

```
feedbackOutput: []
```

```
processFcns:
```

```
{'removeconstantrows',  
mapminmax}
```

```
processParams: {1x2 cell array  
of 2 params}
```

```
processSettings: {1x2 cell array of
```

2 settings}

processedRange: [1x2 double]

processedSize: 1

range: [1x2 double]

size: 1

userdata: (your custom info)

Before the input is applied to the network, it will be processed by two functions: [removeconstantrows](#) and [mapmi](#). These are discussed fully in [Multilayer Neural 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.

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.

## 13.3 DIVIDE DATA FOR OPTIMAL NEURAL NETWORK TRAINING

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.

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
<a href="#"><u>dividerand</u></a>	Divide the data randomly (default)
<a href="#"><u>divideblock</u></a>	Divide the data into contiguous blocks
<a href="#"><u>divideint</u></a>	Divide the data using an interleaved selection
<a href="#"><u>divideind</u></a>	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, n and net.divideParam.testRatio. The fraction of data that is placed in the training set is  $\text{trainRatio}/(\text{trainRatio}+\text{valRatio}+\text{testRatio})$ , 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.

## **13.4 CHOOSE A MULTILAYER NEURAL NETWORK TRAINING FUNCTION**

It is very difficult to know which training algorithm will be the fastest for a given problem. It depends on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). This section compares the

various training algorithms. Feedforward networks are trained on six different problems. Three of the problems fall in the pattern recognition category and the three others fall in the function approximation category. Two of the problems are simple "toy" problems, while the other four are "real world" problems. Networks with a variety of different architectures and complexities are used, and the networks are trained to a variety of different accuracy levels.

The following table lists the algorithms that are tested and the acronyms used to identify them.

Acronym	Algorithm	Description
LM	<a href="#">trainlm</a>	Levenberg-Marquardt
BFG	<a href="#">trainbfg</a>	BFGS Quasi-Newton

RP	<a href="#"><u>trainrp</u></a>	Resilient Backpropagation
SCG	<a href="#"><u>trainscg</u></a>	Scaled Conjugate Gradient
CGB	<a href="#"><u>traincgb</u></a>	Conjugate Gradient with Powell/Beale
CGF	<a href="#"><u>traincgf</u></a>	Fletcher-Powell Conjugate Gradient
CGP	<a href="#"><u>traincgp</u></a>	Polak-Ribi�re Conjugate Gradient
OSS	<a href="#"><u>trainoss</u></a>	One Step Secant
GDX	<a href="#"><u>trainqdx</u></a>	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



## 13.4.1 SIN Data Set

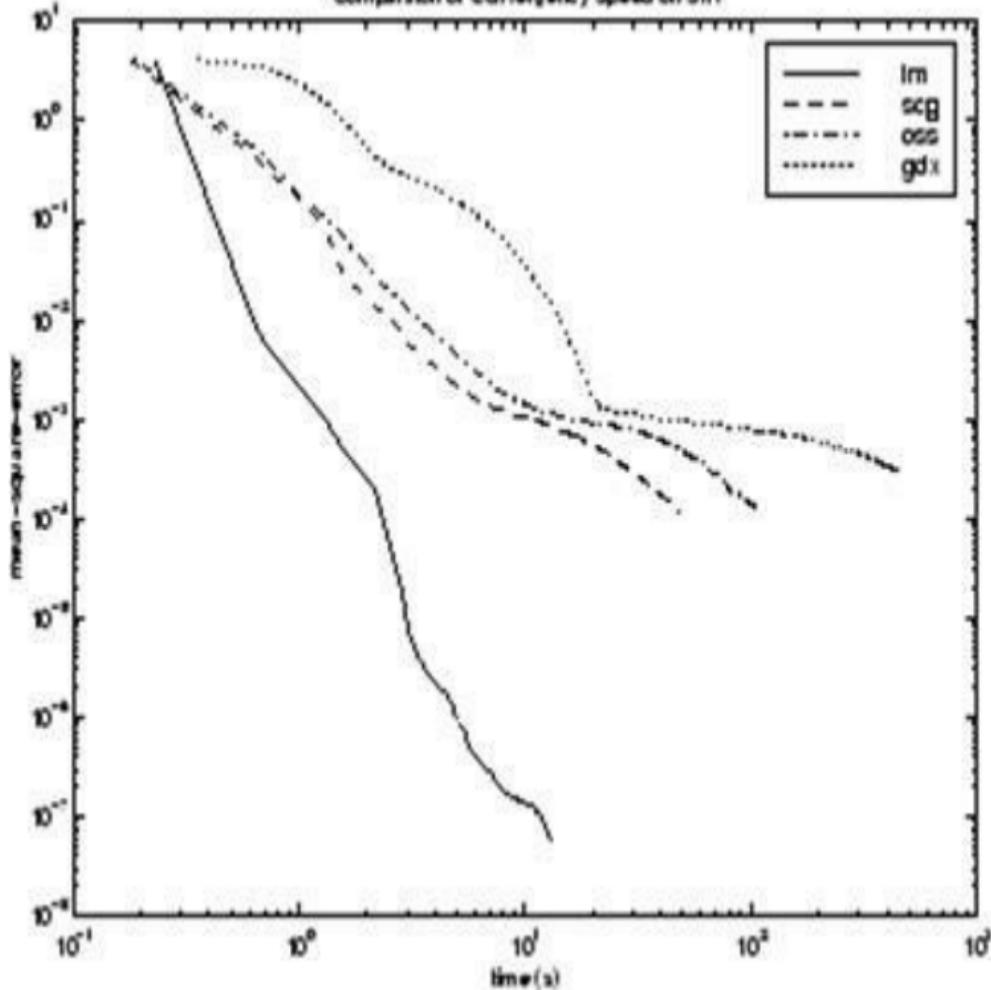
The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with tansig transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training 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)
LM	1.14	1.00	0.65
BFG	5.22	4.58	3.17
RP	5.67	4.97	2.66
SCG	6.09	5.34	3.18
CGB	6.61	5.80	2.99
CGF	7.86	6.89	3.57
CGP	8.24	7.23	4.07
OSS	9.64	8.46	3.97
GDX	27.69	24.29	17.21

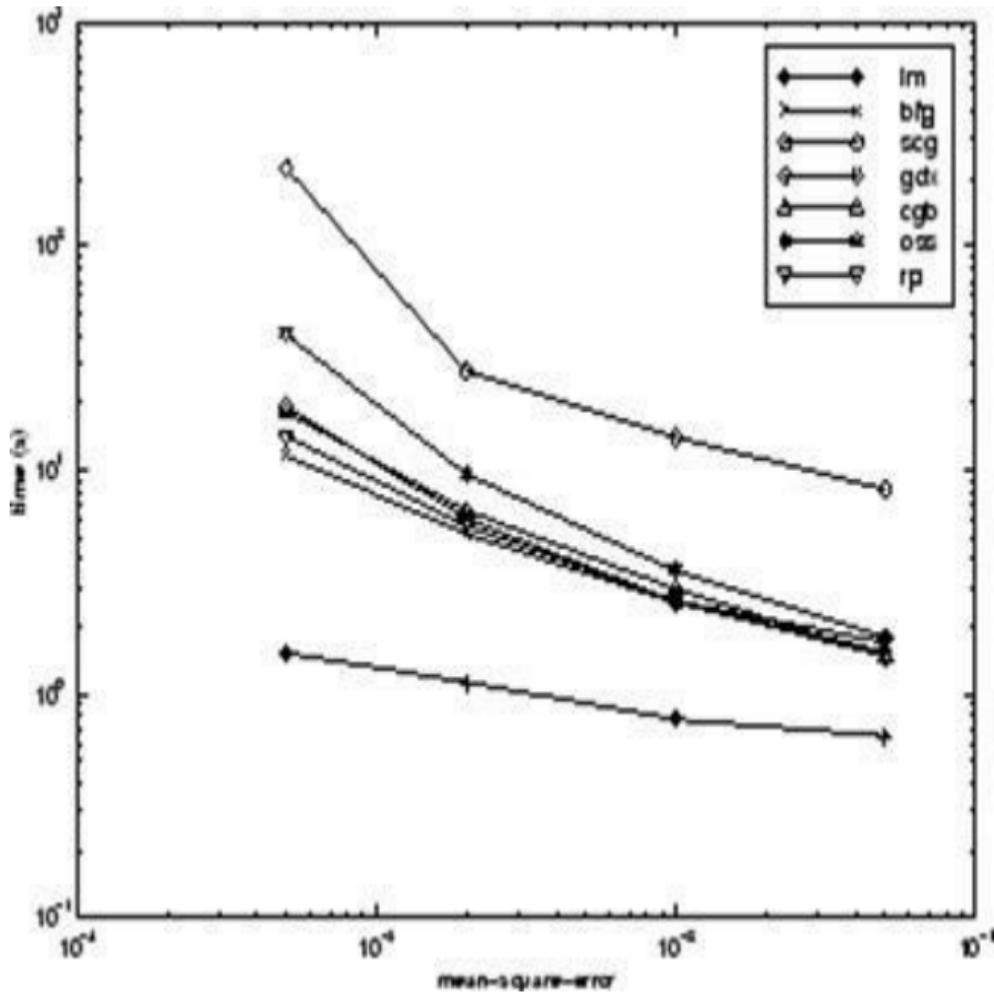
The performance of the various algorithms can be affected by the accuracy required of the approximation. This is shown in the following figure, which plots the mean square error versus execution time (averaged over the 30 trials) for several representative algorithms. Here you can see that the error in the LM algorithm decreases much more rapidly with time than the other algorithms shown.

Comparison of Convergency Speed on SII



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).



## 13.4.2 PARITY Data Set

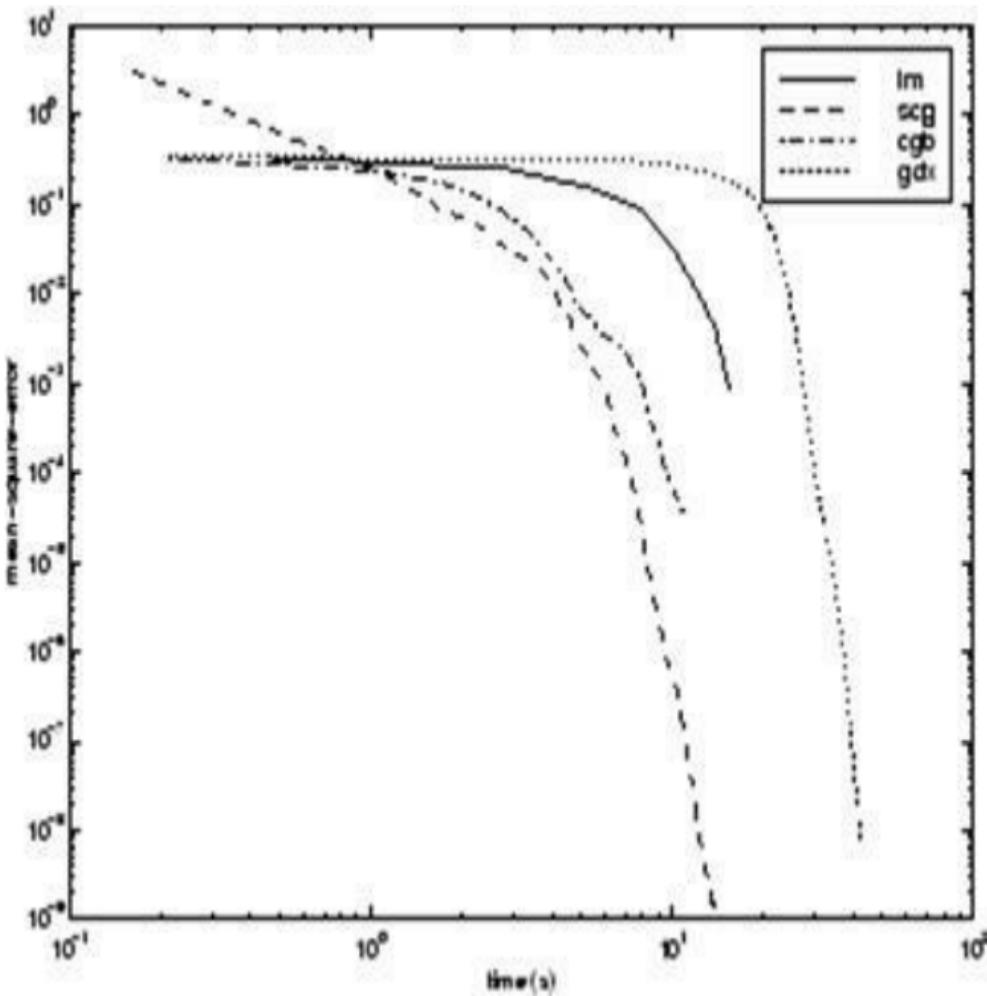
The second benchmark problem is a simple pattern recognition problem—detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a 1; otherwise, it should output a -1. The network used for this problem is a 3-10-10-1 network with tansig neurons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

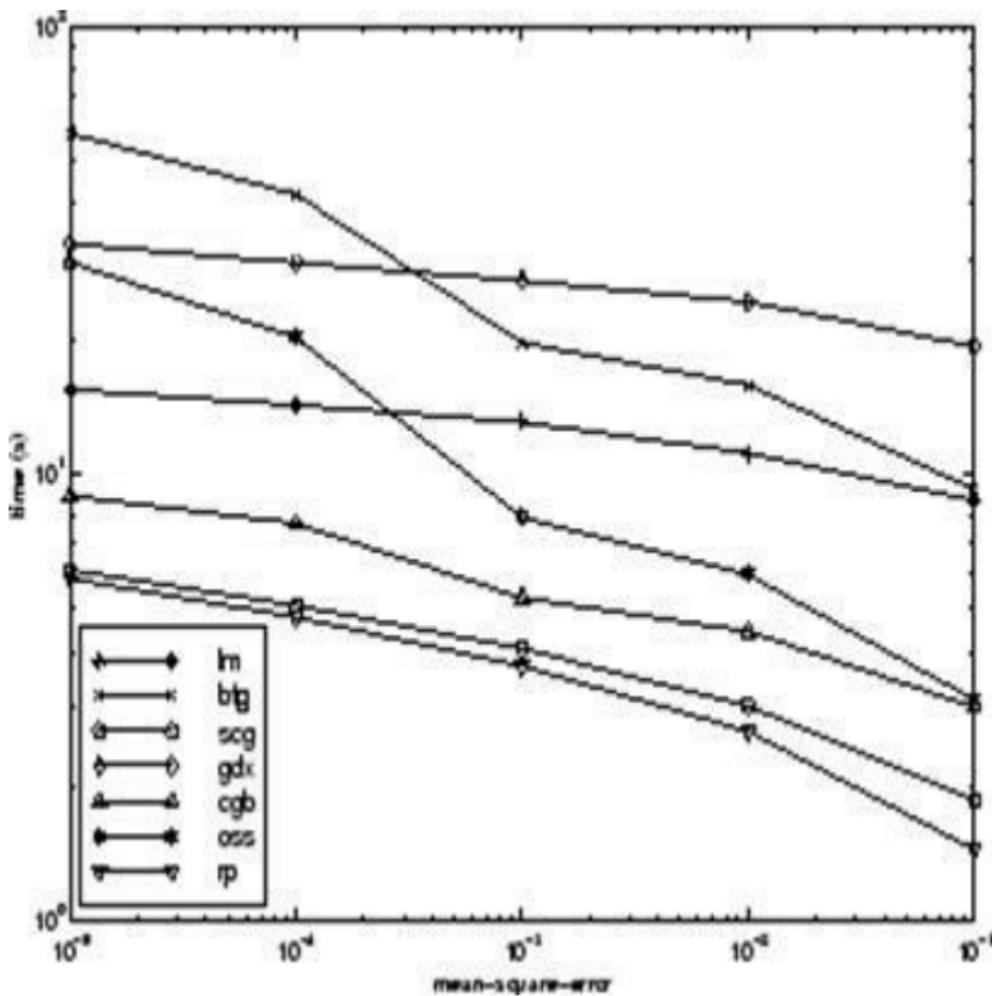
Algorithm	Mean Time (s)	Ratio	Min. Time (s)
RP	3.73	1.00	2.35
SCG	4.09	1.10	2.36
CGP	5.13	1.38	3.50
CGB	5.30	1.42	3.91
CGF	6.62	1.77	3.96
OSS	8.00	2.14	5.06
LM	13.07	3.50	6.48
BFG	19.68	5.28	14.19
GDX	27.07	7.26	25.21

As with function approximation problems, the performance of the various algorithms can be affected by the accuracy required of the network. This is shown in the following figure, which plots the mean square error versus execution time for some typical algorithms. The LM algorithm converges rapidly after some point, but only after the other algorithms have already converged.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to

converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG).



### 13.4.3 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

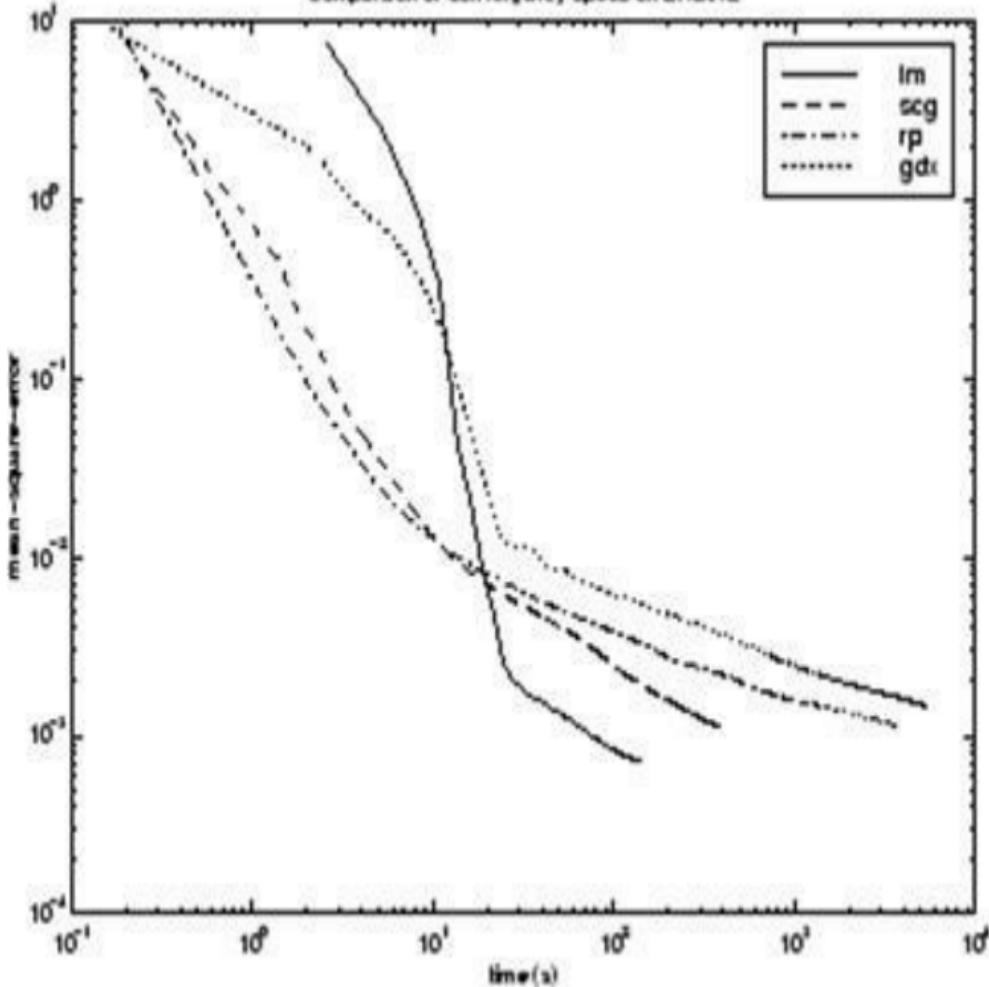
trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, although the BFGS quasi-Newton algorithm and 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)
LM	18.45	1.00	12.01
BFG	27.12	1.47	16.42
SCG	36.02	1.95	19.39
CGF	37.93	2.06	18.89
CGB	39.93	2.16	23.33
CGP	44.30	2.40	24.99
OSS	48.71	2.64	23.51
RP	65.91	3.57	31.83
GDX	188.50	10.22	81.59

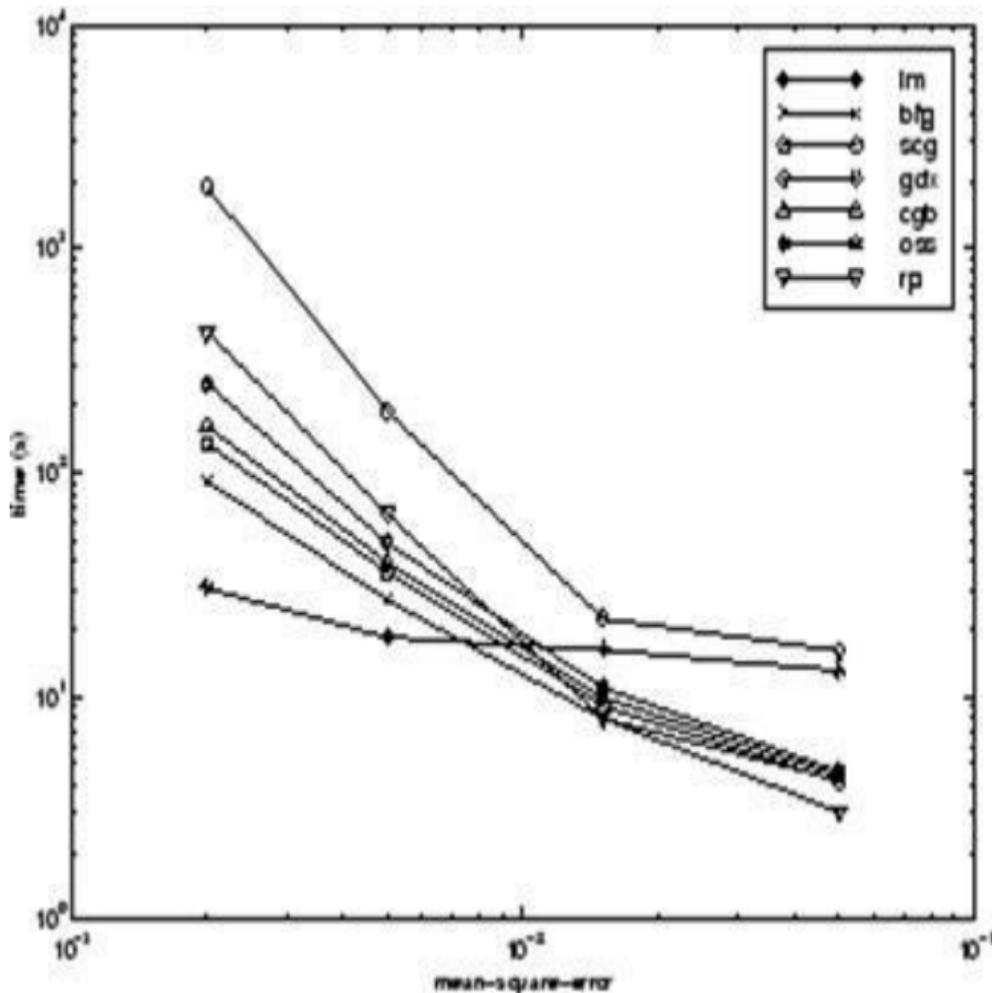
The following figure plots the mean square error versus execution time for some typical algorithms. The performance of the LM algorithm improves over time relative to the other algorithms.

Comparison of Convergancy Speed on ENIGMIE



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.



## 13.4.4 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

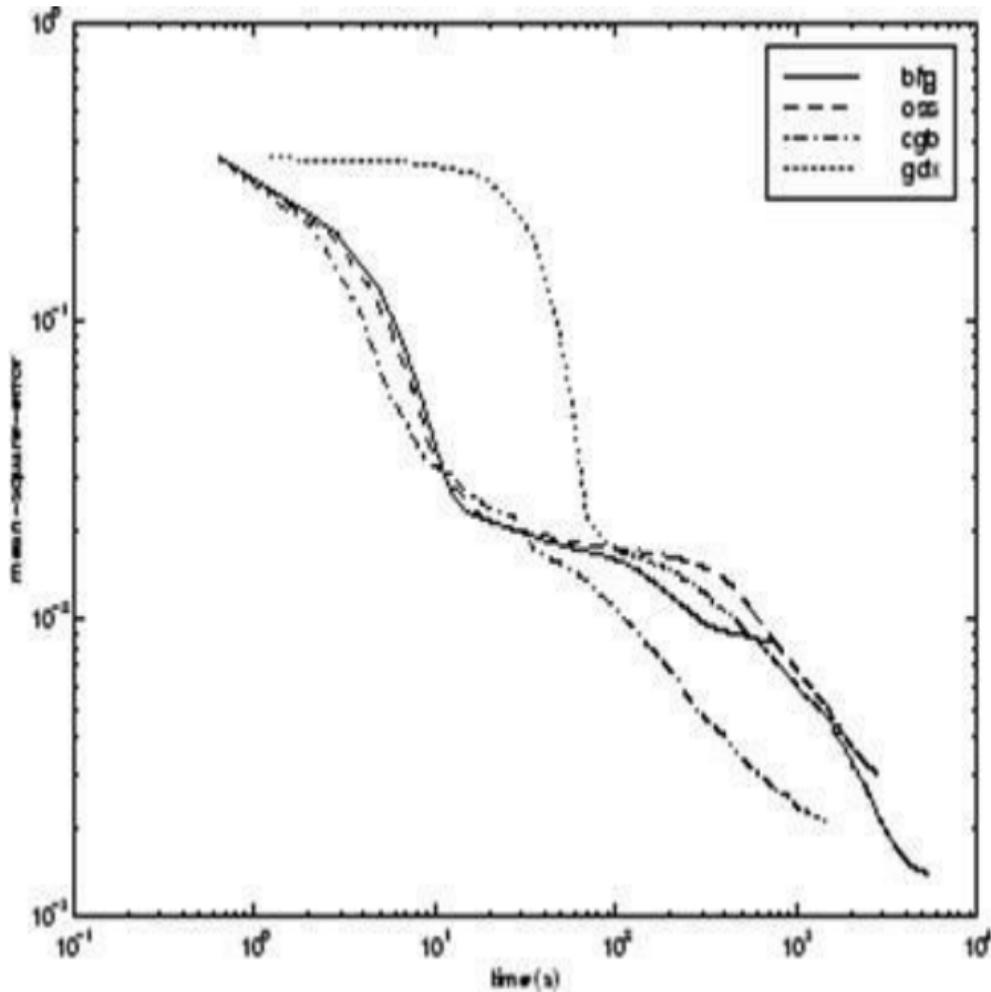
with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until 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.

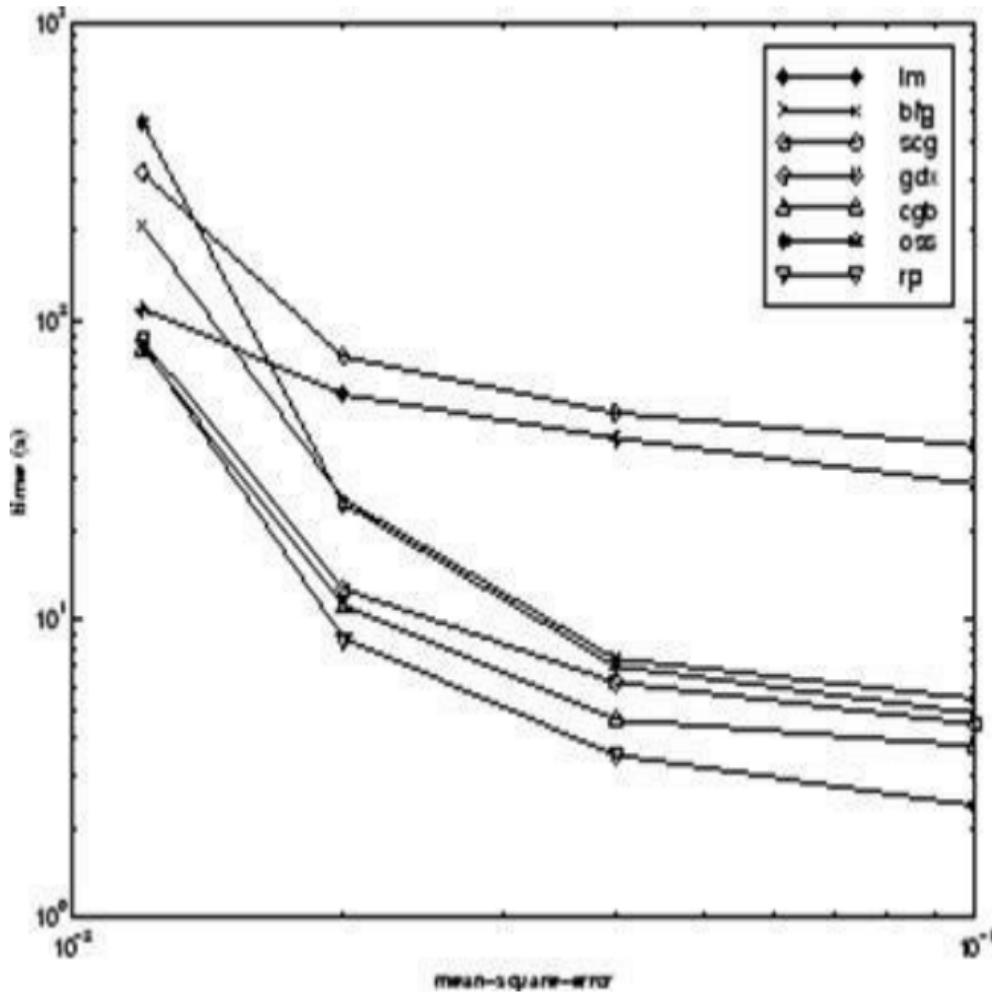
Algorithm	Mean Time (s)	Ratio	Min. Time (s)
CGB	80.27	1.00	55.07
RP	83.41	1.04	59.51
SCG	86.58	1.08	41.21
CGP	87.70	1.09	56.35
CGF	110.05	1.37	63.33
LM	110.33	1.37	58.94
BFG	209.60	2.61	118.92
GDX	313.22	3.90	166.48
OSS	463.87	5.78	250.62

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.



## 13.4.5 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.

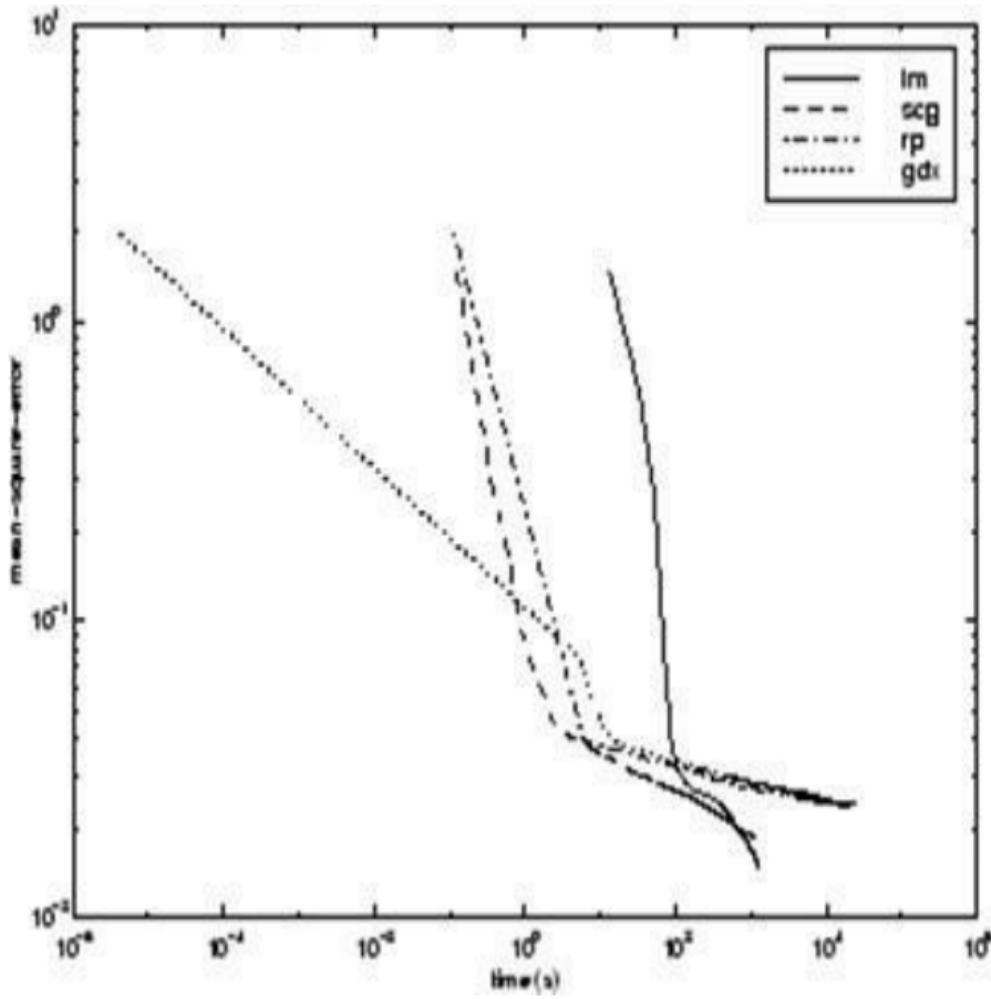
The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases 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)
SCG	99.73	1.00	83.10
CGP	121.54	1.22	101.76
CGB	124.06	1.2	107.64
CGF	136.04	1.36	106.46
LM	261.50	2.62	103.52
OSS	268.55	2.69	197.84
BFG	550.92	5.52	471.61
RP	1519.00	15.23	581.17
GDX	3169.50	31.78	2514.90

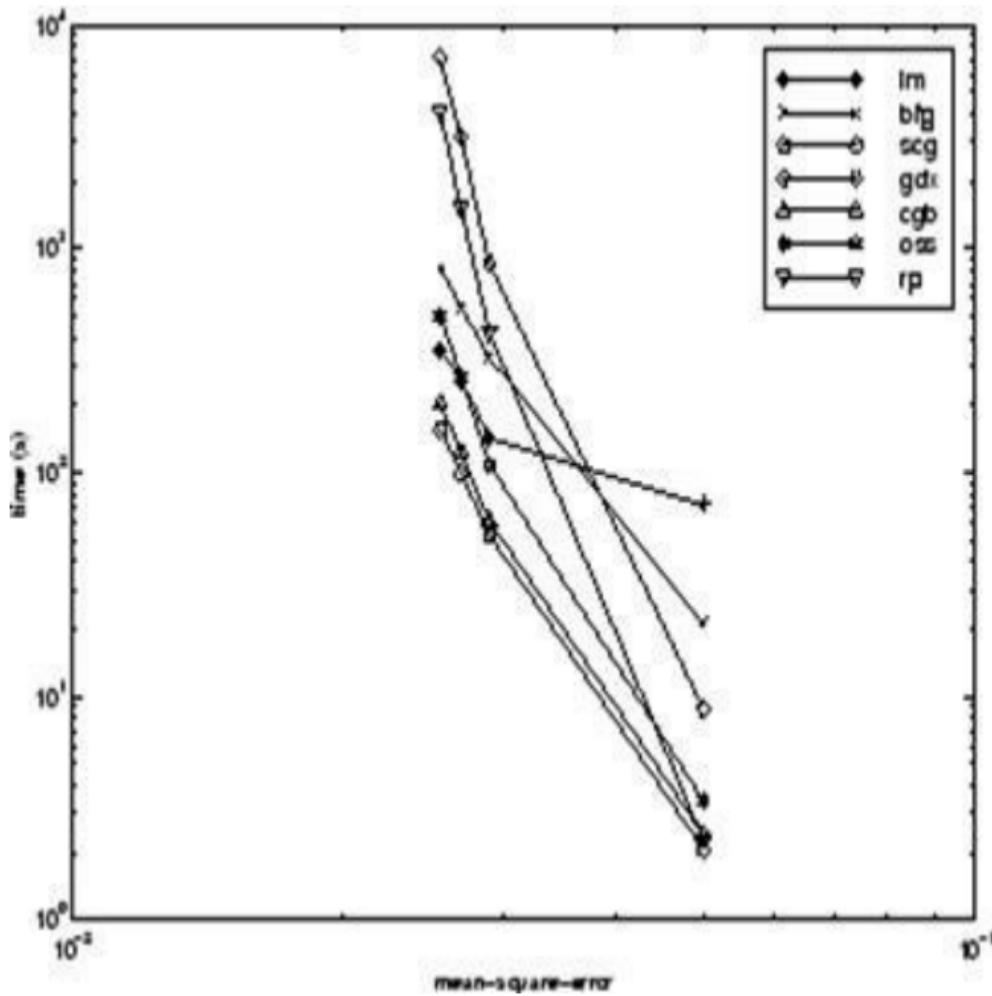
The following figure plots the mean square error versus execution time for some typical algorithms. For this problem, you can see that the LM

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.



## 13.4.6 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.

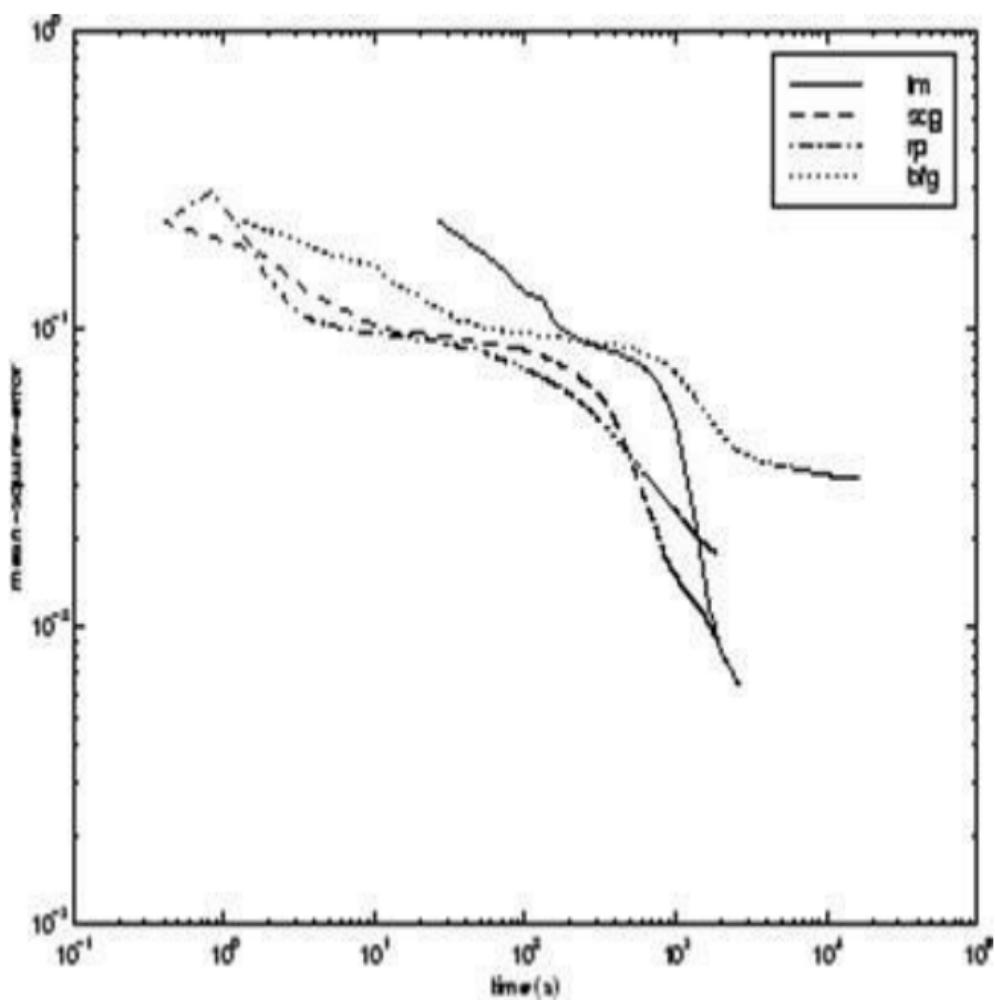
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems considered. The RP algorithm works well on all the pattern recognition problems. This is reasonable, because that algorithm 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.

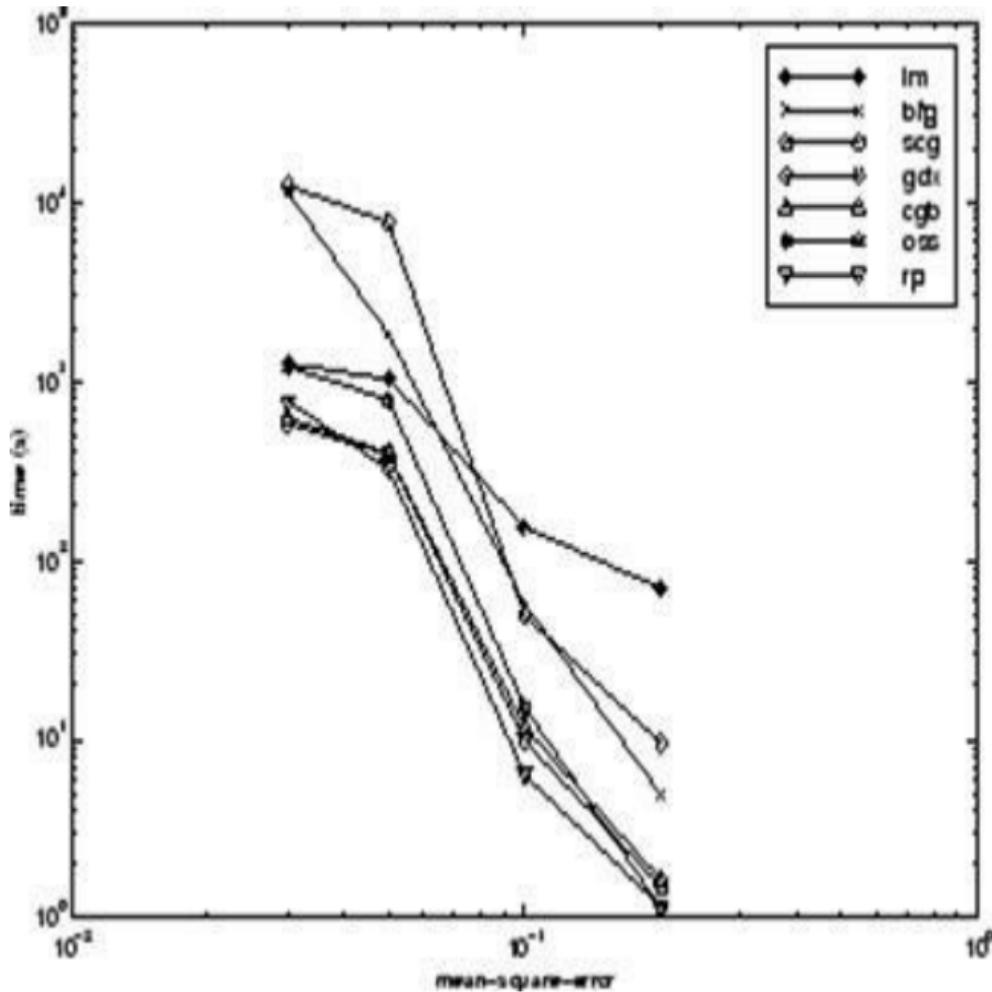
Algorithm	Mean Time (s)	Ratio	Min. Time (s)
RP	323.90	1.00	187.43
SCG	390.53	1.21	267.99
CGB	394.67	1.22	312.25
CGP	415.90	1.28	320.62
OSS	784.00	2.42	706.89
CGF	784.50	2.42	629.42
LM	1028.10	3.17	802.01
BFG	1821.00	5.62	1415.80
GDX	7687.00	23.73	5169.20

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.



## 13.4.7 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.

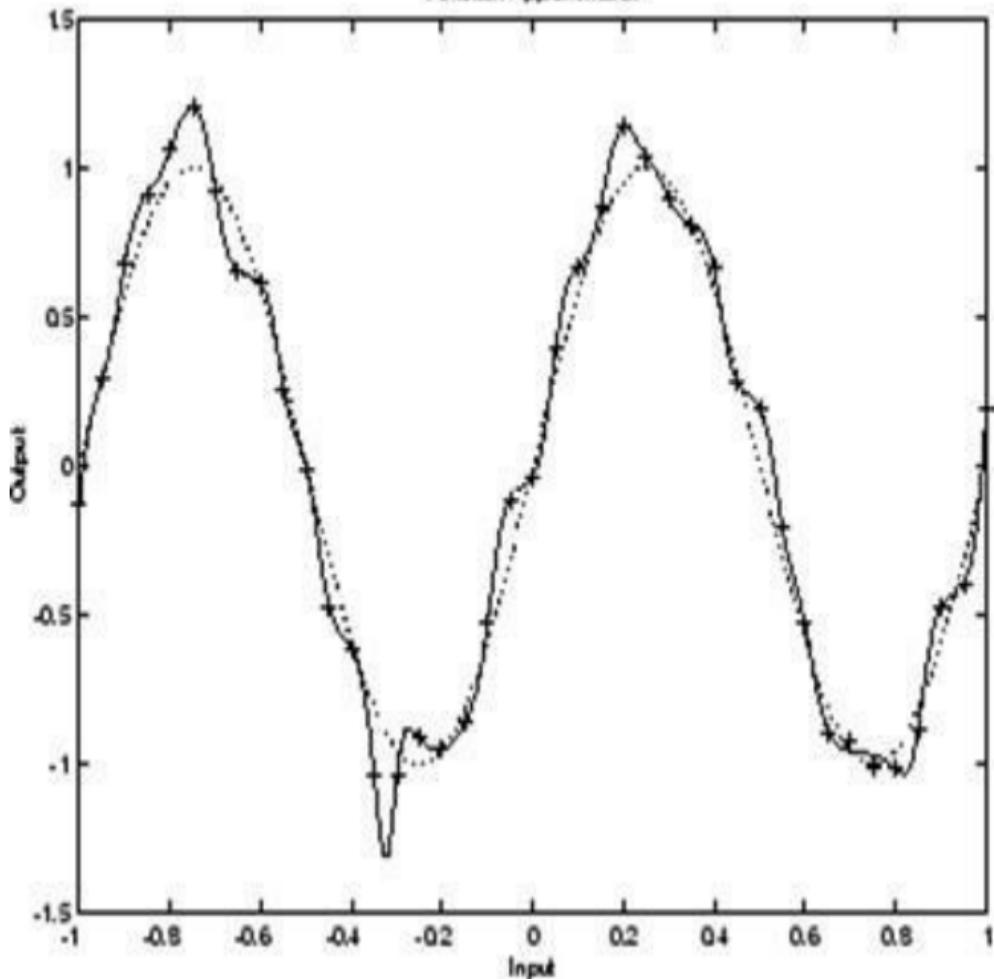
# **13.5 IMPROVE NEURAL NETWORK GENERALIZATION AND AVOID OVERFITTING**

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.

### Function Approximation



One method for improving network generalization is to use a network that is just large enough to provide an adequate

fit. The larger network you use, the more complex the functions the network can create. If you use a small enough network, it will not have enough power to overfit the data. Run the *Neural Network*

*Design* example `nndl1gn` [[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.

# 1 3 .5 .1 Retraining Neural Networks

Typically each backpropagation training session starts with different initial weights and biases, and different divisions of data into training, validation, and test sets. These different conditions can lead to very different solutions for the same problem.

It is a good idea to train several networks to ensure that a network with good generalization is found.

Here a dataset is loaded and divided into two parts: 90% for designing networks and 10% for testing them all.

```
[x, t] =  
house_dataset;  
  
Q = size(x, 2);  
  
Q1 = floor(Q*0.90);  
  
Q2 = Q-Q1;  
  
ind = randperm(Q);  
  
ind1 = ind(1:Q1);
```

```
ind2 = ind(Q1+  
(1:Q2)) ;
```

```
x1 = x(:,ind1) ;
```

```
t1 = t(:,ind1) ;
```

```
x2 = x(:,ind2) ;
```

```
t2 = t(:,ind2) ;
```

Next a network architecture is chosen and trained ten times on the first part of

the dataset, with each network's mean square error on the second part of the dataset.

```
net =
feedforwardnet(10);

numNN = 10;

NN = cell(1,numNN);

perfs =
zeros(1,numNN);
```

```
for i=1:numNN  
    disp(['Training ' num2str(i) '/' num2str(numNN)])  
  
    NN{i} = train(net,x1,t1);  
  
    y2 = NN{i}(x2);  
  
    perfs(i) =
```

```
mse(net,t2,y2);  
end
```

Each network will be trained starting from different initial weights and biases, and with a different division of the first dataset into training, validation, and test sets. Note that the test sets are a good measure of generalization for each respective network, but not for all the networks, because data that is a test set for one network will likely be used for training or validation by other neural networks. This is why the original dataset was divided into two parts, to

ensure that a completely independent test set is preserved.

The neural network with the lowest performance is the one that generalized best to the second part of the dataset.

# 1 3 .5 .2 Multiple Neural Networks

Another simple way to improve generalization, especially when caused by noisy data or a small dataset, is to train multiple neural networks and average their outputs.

For instance, here 10 neural networks are trained on a small problem and their mean squared errors compared to the means squared error of their average.

First, the dataset is loaded and divided into a design and test set.

```
[x, t] =  
house_dataset;  
  
Q = size(x, 2);  
  
Q1 = floor(Q*0.90);  
  
Q2 = Q-Q1;  
  
ind = randperm(Q);  
  
ind1 = ind(1:Q1);
```

```
ind2 = ind(Q1+  
(1:Q2)) ;
```

```
x1 = x(:,ind1) ;
```

```
t1 = t(:,ind1) ;
```

```
x2 = x(:,ind2) ;
```

```
t2 = t(:,ind2) ;
```

Then, ten neural networks are trained.

```
net =
```

```
feedforwardnet(10);  
  
numNN = 10;  
  
nets =  
cell(1, numNN);  
  
for i=1:numNN  
  
    disp(['Training ']  
num2str(i) '/'  
num2str(numNN)])
```

```
nets{i} =  
train(net,x1,t1);  
  
end
```

Next, each network is tested on the second dataset with both individual performances and the performance for the average output calculated.

```
perfs =  
zeros(1,numNN);  
  
y2Total = 0;
```

```
for i=1:numNN  
    neti = nets{i};  
  
    y2 = neti(x2);  
  
    perfs(i) =  
    mse(neti,t2,y2);  
  
    y2Total = y2Total  
+ y2;
```

end

perfs

y2AverageOutput =  
y2Total / numNN;

perfAveragedOutputs  
= mse(nets{1}, t2, y2Ave);

The mean squared error for the average output is likely to be lower than most of the individual performances, perhaps not

all. It is likely to generalize better to additional new data.

For some very difficult problems, a hundred networks can be trained and the average of their outputs taken for any input. This is especially helpful for a small, noisy dataset in conjunction with the Bayesian Regularization training function [trainbr](#), described below.

### 13.5.3 Early Stopping

The default method for improving generalization is called *early stopping*. This technique is automatically provided for all of the supervised network creation functions, including the backpropagation network creation functions such as [feedforwardnet](#).

In this technique the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The

validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. When the validation error increases for a specified number of iterations (`net.trainParam.max_fail`), the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set

reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are [dividerand](#) (the default), [divideblock](#), [divideint](#), and [divideind](#). You can access or change the division function for your network with this property:

`net.divideFcn`

Each of these functions takes parameters that customize its behavior. These values are stored and can be changed with the

following network property:

```
net.divideParam
```

## 13.5.4 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*rand;
```

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)
```

```
[trainT, valT, testT]
```

=

divideind(t, trainInd)

## 13.5.5 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, t:  
= 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)
```

## 13.5.6 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, t:  
= divideblock(p);  
Divide the target data accordingly  
using divideind:
```

```
[trainT, valT, testT]  
=  
divideind(t, trainInd)
```

# 1 3 .5 .7 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, t:  
= divideint(p);  
Divide the target data accordingly
```

using [divideind](#).

```
[trainT, valT, testT]  
=  
divideind(t, trainInd)
```

## 13.5.8 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.

## 13.5.9 Modified Performance Function

The typical performance function used for training feedforward neural networks is the mean sum of squares of the network errors.

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

It is possible to improve generalization if you modify the performance function by adding a term that consists of the mean of the sum of squares of the

network weights and biases

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

where  $\gamma$  is the performance ratio, and

$$msw = \frac{1}{n} \sum_{j=1}^n w_j^2$$

Using this performance function causes the network to have smaller weights and biases, and this forces the network response to be smoother and less likely to overfit.

The following code reinitializes the previous network and retrains it using the BFGS algorithm with the regularized performance function. Here the performance ratio is set to 0.5, which gives equal weight to the mean square

errors and the mean square weights.  
(Data division is cancelled by  
setting `net.divideFcn` so that the  
effects of `msereg` are isolated from early  
stopping.)

```
[x, t] =  
simplefit_dataset;  
  
net =  
feedforwardnet(10, 't:  
net.divideFcn = '';
```

```
net.trainParam.epoch  
= 300;
```

```
net.trainParam.goal  
= 1e-5;
```

```
net.performParam.reg  
= 0.5;
```

```
net =  
train(net, x, t);
```

The problem with regularization is that it is difficult to determine the optimum

value for the performance ratio parameter. If you make this parameter too large, you might get overfitting. If the ratio is too small, the network does not adequately fit the training data. The next section describes a routine that automatically sets the regularization parameters.

# 1 3 .5 .1 0 Automated Regularization (trainbr)

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [[MacK92](#)]. In this framework, the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. You can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this user guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [[FoHa97](#)].

Bayesian regularization has been implemented in the function [trainbr](#). The following code shows how you can train a 1-20-1 network using this function to approximate the noisy sine wave shown in the figure in [Improve Neural Network Generalization and Avoid Overfitting](#). (Data division is cancelled by setting `net.divideFcn` so that the effects of [trainbr](#) are isolated from early stopping.)

```
x = -1:0.05:1;
```

```
t = sin(2*pi*x) +  
0.1*randn(size(x));
```

```
net =  
feedforwardnet(20,'t');
```

```
net =  
train(net,x,t);
```

One feature of this algorithm is that it provides a measure of how many

network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by `#Par` in the printout) out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain approximately the same, no matter how large the number of parameters in the network becomes. (This assumes that the network has been trained for a sufficient number of iterations to ensure convergence.)

The [trainbr](#) algorithm generally works best when the network inputs and targets are scaled so that they fall

approximately in the range  $[-1,1]$ . That is the case for the test problem here. If your inputs and targets do not fall in this range, you can use the function `mapminmax` or `mapstd` to perform the scaling, as described in [Choose Neural Network Input-Output Processing Functions](#). Networks created with `feedforwardnet` include `mapminmax` as an input and output processing function by default.

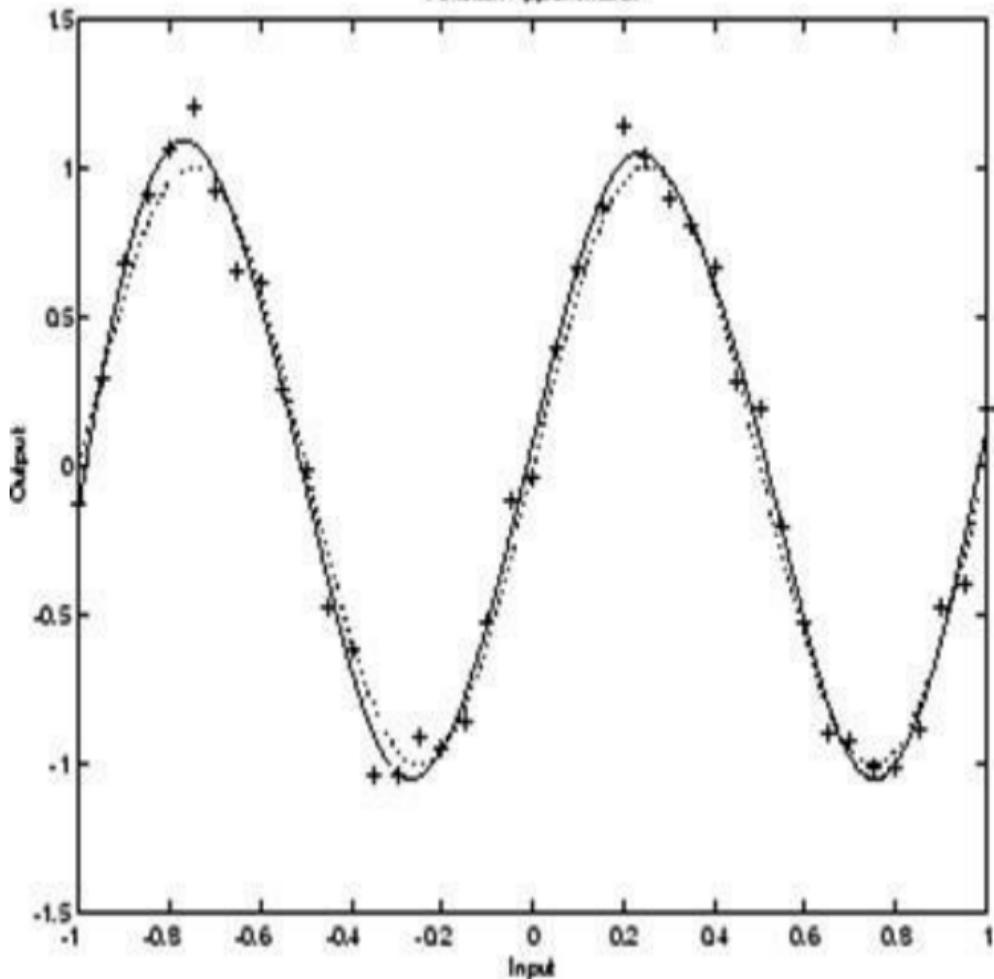
The following figure shows the response of the trained network. In contrast to the previous figure, in which a 1-20-1 network overfits the data, here you see that the network response is very close to the underlying sine function (dotted

line), and, therefore, the network will generalize well to new inputs. You could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.

When using [`trainbr`](#), it is important to let the algorithm run until the effective number of parameters has converged. The training might stop with the message "Maximum MU reached." This is typical, and is a good indication that the algorithm has truly converged. You can also tell that the algorithm has converged if the sum squared error (SSE) and sum squared weights (SSW) are relatively

constant over several iterations. When this occurs you might want to click the **Stop Training** button in the training window.

### Function Approximation



## 13.5.11 Summary and Discussion of Early Stopping and Regularization

Early stopping and regularization can ensure network generalization when you apply them properly.

For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are using a fast algorithm (like `trainlm`), set the training parameters so that the convergence is relatively slow. For example, set `mu` to a relatively large value, such as 1, and

set `mu_dec` and `mu_inc` to values close to 1, such as 0.8 and 1.5, respectively. The training functions [`trainscg`](#) and [`trainbr`](#) usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

When you use Bayesian regularization, it is important to train the network until it reaches convergence. The sum-squared error, the sum-squared weights, and the effective number of parameters should reach constant values when the network has converged.

With both early stopping and regularization, it is a good idea to train the network starting from several different initial conditions. It is possible for either method to fail in certain circumstances. By testing several different initial conditions, you can verify robust network performance.

When the data set is small and you are training function approximation networks, Bayesian regularization provides better generalization performance than early stopping. This is because Bayesian regularization does not require that a validation data set be separate from the training data set; it uses all the data.

To provide some insight into the performance of the algorithms, both early stopping and Bayesian regularization were tested on several benchmark data sets, which are listed in the following table.

Data Set Title	Number of Points	Network	Description
BALL	67	2-10-1	Dual-sensor calibra
SINE (5% N)	41	1-15-1	Single-cycle sine w
SINE (2% N)	41	1-15-1	Single-cycle sine w
ENGINE (ALL)	1199	2-30-2	Engine sensor—ful
ENGINE (1/4)	300	2-30-2	Engine sensor—1/4
CHOLEST (ALL)	264	5-15-3	Cholesterol measu
CHOLEST (1/2)	132	5-15-3	Cholesterol measu

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)	Ch (1/4)
ES	1.2e-1	1.3e-2	1.9e-2	1.2e-1	1.4
BR	1.3e-3	2.6e-3	4.7e-3	1.2e-1	9.3
ES/BR	92	5	4	1	1.5

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.

## 13.5.12 Posttraining Analysis (regression)

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

The following commands illustrate how to perform a regression analysis on a network trained.

```
x = [-1:.05:1];
```

```
t =
```

```
sin(2*pi*x)+0.1*rand;
```

```
net =
```

```
feedforwardnet(10);
```

```
net =
```

```
train(net,x,t);
```

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

[ r , m , b ] =  
regression ( t , y )

r =

0 . 9935

m =

0 . 9874

b =

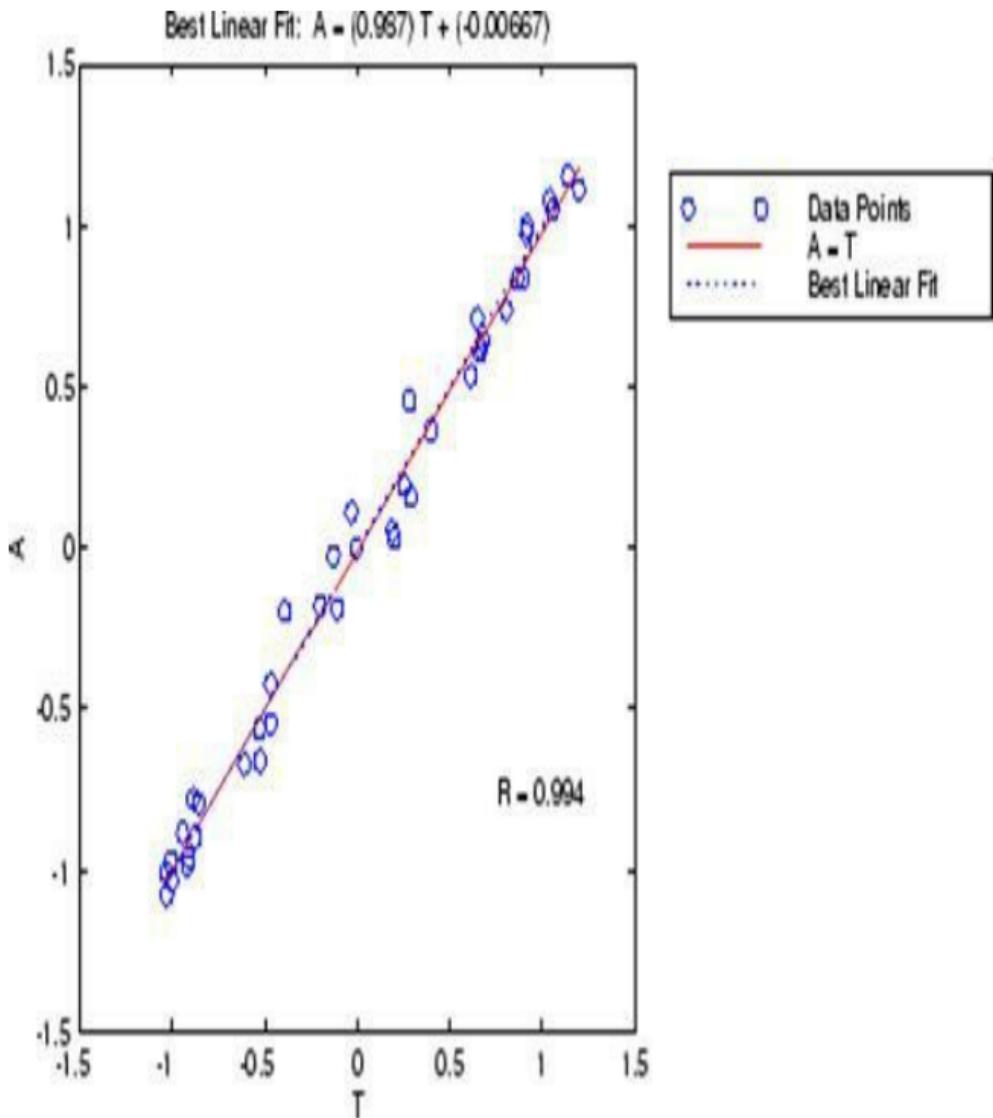
-0.0067

The network output and the corresponding targets are passed to regression. It returns three parameters. The first two,  $m$  and  $b$ , correspond to the slope and the  $y$ -intercept of the best linear regression relating targets to network outputs. If there were a perfect fit (outputs exactly equal to targets), the slope would be 1, and the  $y$ -intercept would be 0. In this example, you can see that the numbers are very close. The third variable returned by regression is the correlation coefficient (R-value)

between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In the example, the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by regression. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In this example, it is difficult to distinguish the best linear fit line from

the perfect fit line because the fit is so good.



## 13.6 TRAIN NEURAL NETWORKS WITH ERROR WEIGHTS

In the default mean square error performance function (see [Train and Apply Multilayer Neural Networks](#)), each squared error contributes the same amount to the performance function as follows:

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

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

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

The error weighting object needs to have the same dimensions as the target data. In this way, errors can be weighted according to time step, sample number, signal number or element number. The following is an example of weighting the errors at the end of a time sequence more heavily than errors at the beginning of a time sequence. The error weighting object is passed as the last argument in the call to [train](#).

```
y = laser_dataset;
```

```
y = y(1:600);  
  
ind = 1:600;  
  
ew = 0.99.^ (600-  
ind);  
  
figure  
  
plot(ew)  
  
ew = con2seq(ew);
```

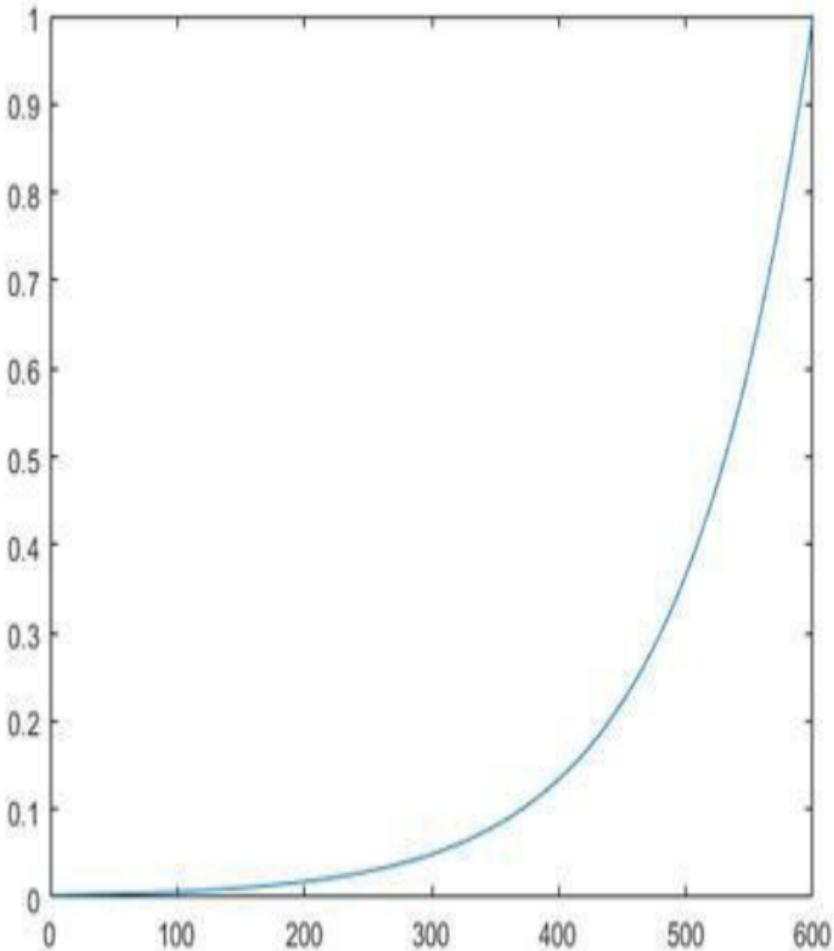
```
ftdnn_net =
timedelaynet([1:8],1)

ftdnn_net.trainParam
= 1000;

ftdnn_net.divideFcn
= ' ';

[p,Pi,Ai,t,ew1] =
preparets(ftdnn_net,
{},ew);
```

```
[ftdnn_net1,tr] =  
train(ftdnn_net,p,t,:)
```

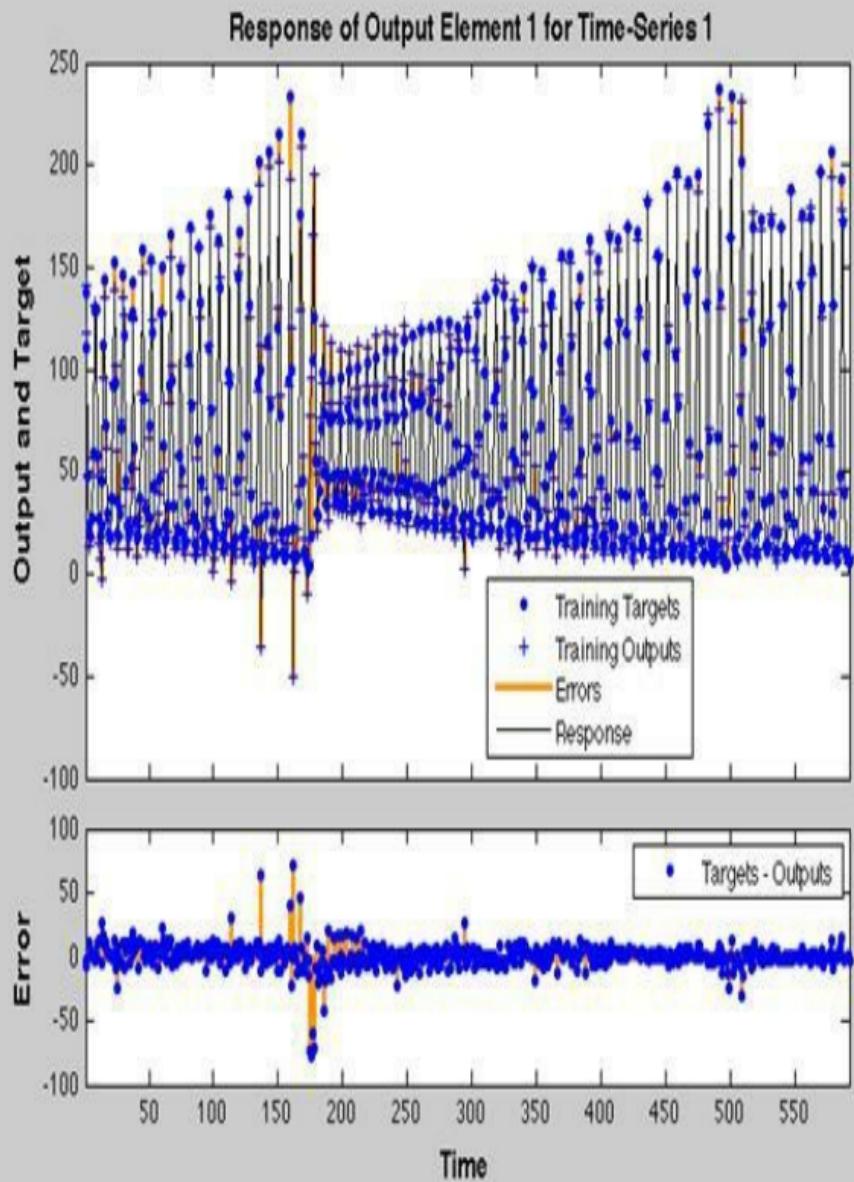


The figure illustrates the error weighting for this example. There are 600 time

steps in the training data, and the errors are weighted exponentially, with the last squared error having a weight of 1, and the squared error at the first time step having a weighting of 0.0024.

The response of the trained network is shown in the following figure. If you compare this response to the response of the network that was trained without exponential weighting on the squared errors, as shown in [Design Time Series Time-Delay Neural Networks](#), 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.





## 13.7 NORMALIZE ERRORS OF MULTIPLE OUTPUTS

The most common performance function used to train neural networks is mean squared error (mse). However, with multiple outputs that have different ranges of values, training with mean squared error tends to optimize accuracy on the output element with the wider range of values relative to the output element with a smaller range.

For instance, here two target elements have very different ranges:

$$x = -1:0.01:1;$$

```
t1 = 100*sin(x);  
t2 = 0.01*cos(x);  
t = [t1; t2];
```

The range of t1 is 200 (from a minimum of -100 to a maximum of 100), while the range of t2 is only 0.02 (from -0.01 to 0.01). The range of t1 is 10,000 times greater than the range of t2.

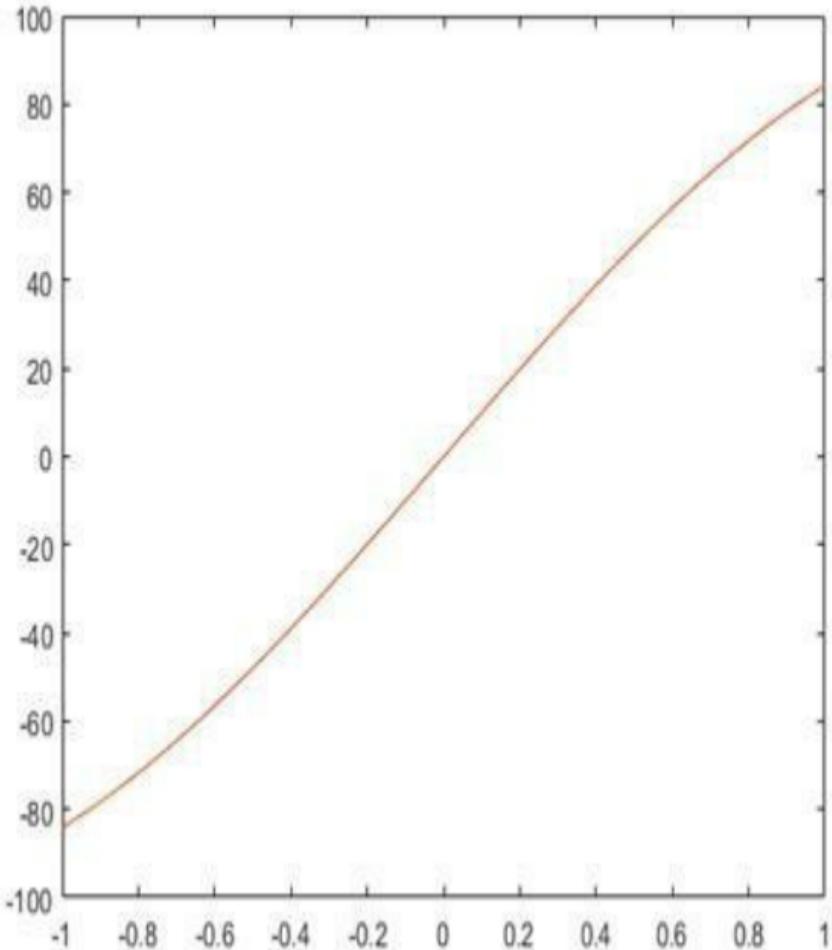
If you create and train a neural network on this to minimize mean squared error, training favors the relative accuracy of the first output element over the second.

```
net = feedforwardnet(5);  
net1 = train(net,x,t);  
y = net1(x);
```

Here you can see that the network has

learned to fit the first output element very well.

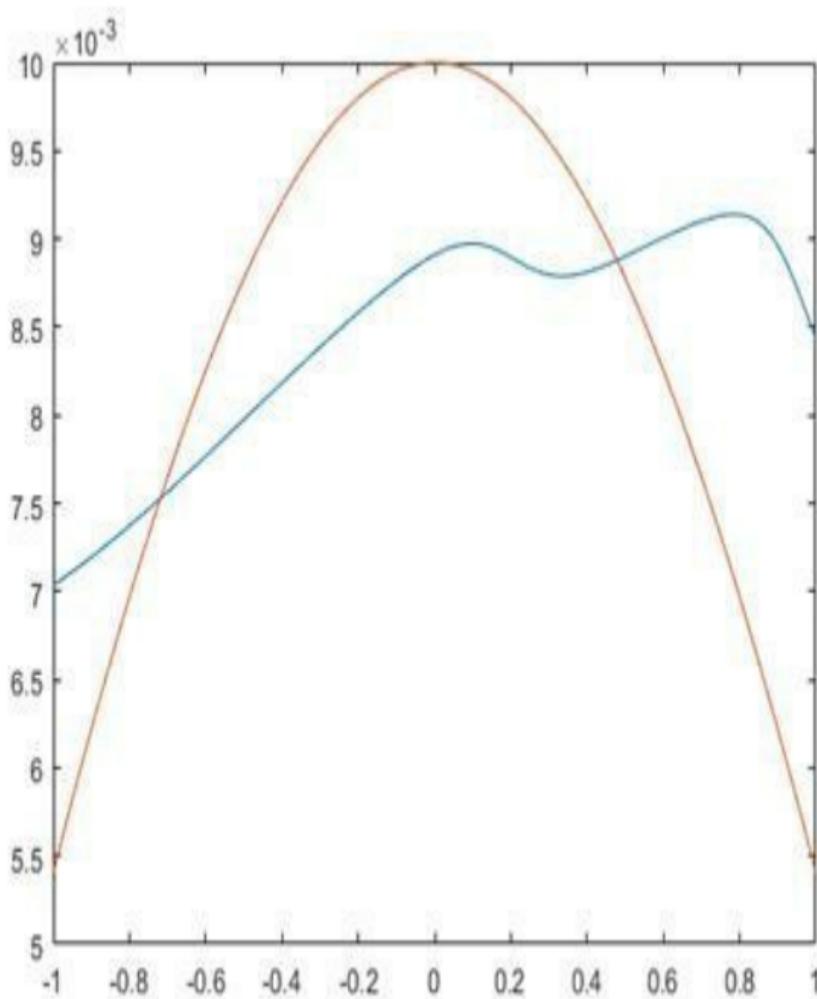
```
figure(1)  
plot(x,y(1,:),x,t(1,:))
```



However, the second element's function is not fit nearly as well.

figure(2)

plot(x,y(2,:),x,t(2,:))



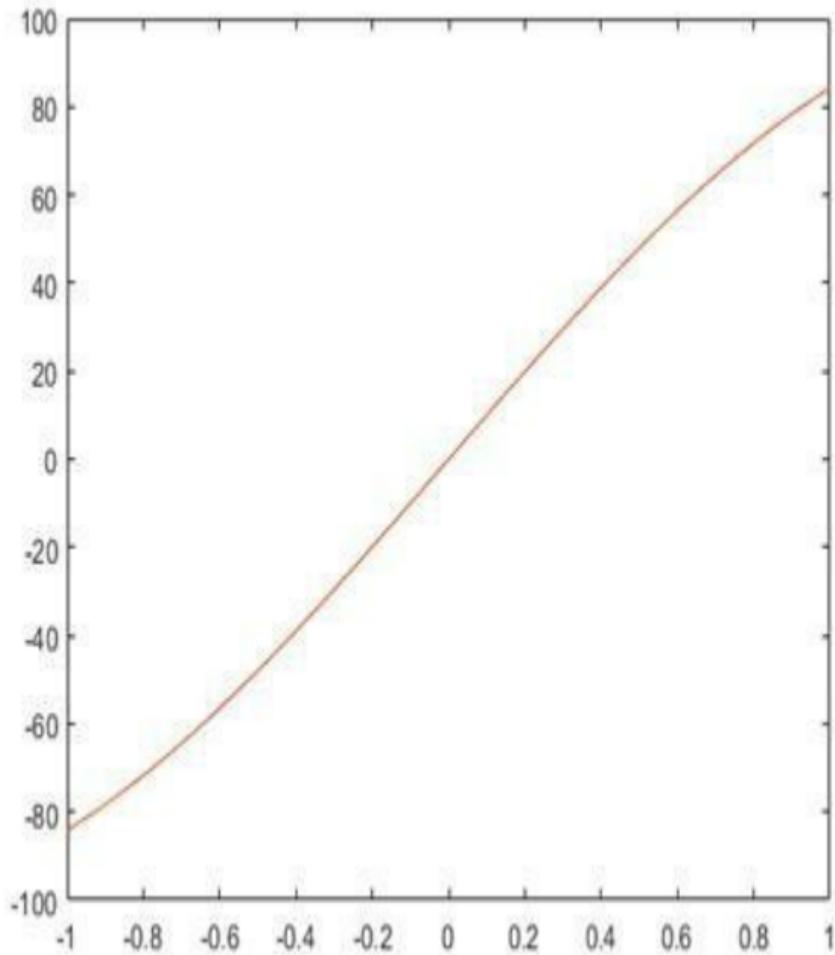
To fit both output elements equally well in a relative sense, set the normalization performance parameter to 'standard'. This then calculates errors for performance measures as if each output element has a range of 2 (i.e., as if each output element's values range from -1 to 1, instead of their differing ranges).

```
net.performParam.normalization =  
'standard';  
net2 = train(net,x,t);  
y = net2(x);
```

Now the two output elements both fit well.

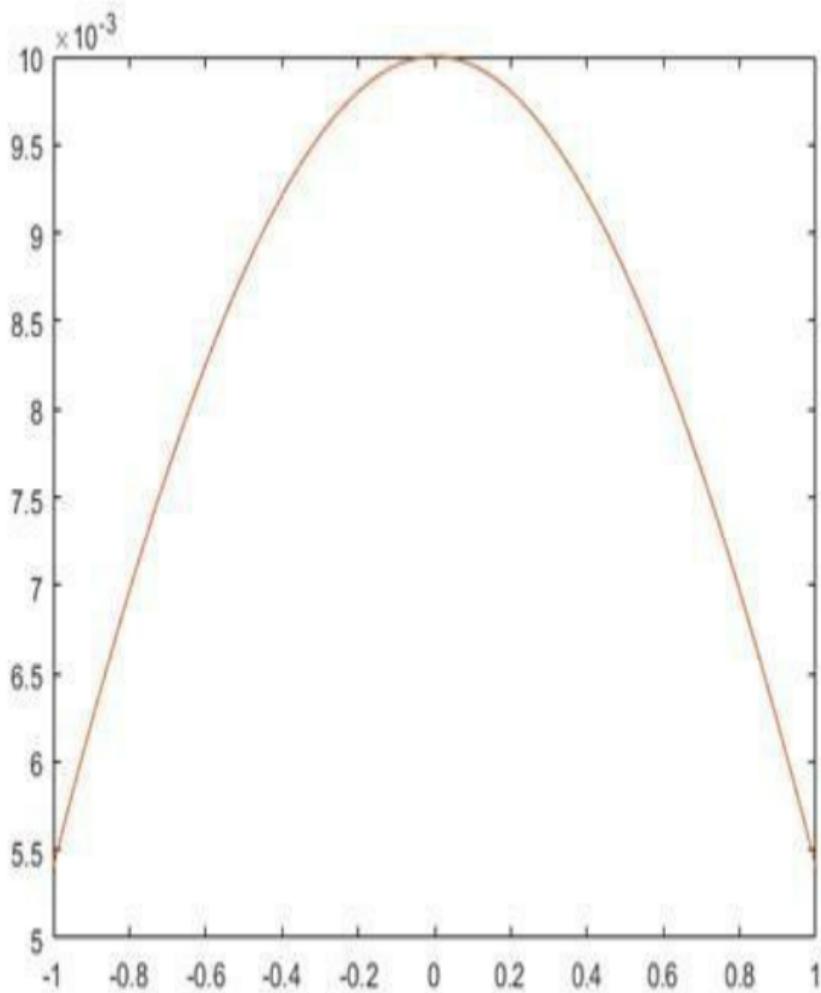
figure(3)

```
plot(x,y(1,:),x,t(1,:))
```



figure(4)

```
plot(x,y(2,:),x,t(2,:))
```



# **Chapter 14**

## **CLASSIFICATION WITH NEURAL NETWORKS. EXAMPLES**

---

# 14.1 CRAB CLASSIFICATION

This example illustrates using a neural network as a classifier to identify the sex of crabs from physical dimensions of the crab.

In this example we attempt to build a classifier that can identify the sex of a crab from its physical measurements. Six physical characteristics of a crab are considered: species, frontallip, rearwidth, length, width and depth. The problem on hand is to identify the sex of a crab given the observed values for each of these 6 physical characteristics.

## 14.1.1 Why Neural Networks?

Neural networks have proven themselves as proficient classifiers and are particularly well suited for addressing non-linear problems. Given the non-linear nature of real world phenomena, like crab classification, neural networks is certainly a good candidate for solving the problem.

The six physical characterstics will act as inputs to a neural network and the sex of the crab will be target. Given an input, which constitutes the six observed values for the physical characterstics of

a crab, the neural network is expected to identify if the crab is male or female.

This is achieved by presenting previously recorded inputs to a neural network and then tuning it to produce the desired target outputs. This process is called neural network training.

## 14.1.2 Preparing the Data

Data for classification problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have six elements representing a crabs species, fontallip, rearwidth, length, width and depth.

Each corresponding column of the target matrix will have two elements. Female crabs are reprented with a one in the first element, male crabs with a one in the second element. (All other elements are zero).

Here such the dataset is loaded.

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

```
size(x)
```

```
size(t)
```

```
ans =
```

6 200

ans =

2 200

## 14.1.3 Building the Neural Network Classifier

The next step is to create a neural network that will learn to identify the sex of the crabs.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

setdemorandstream(49)

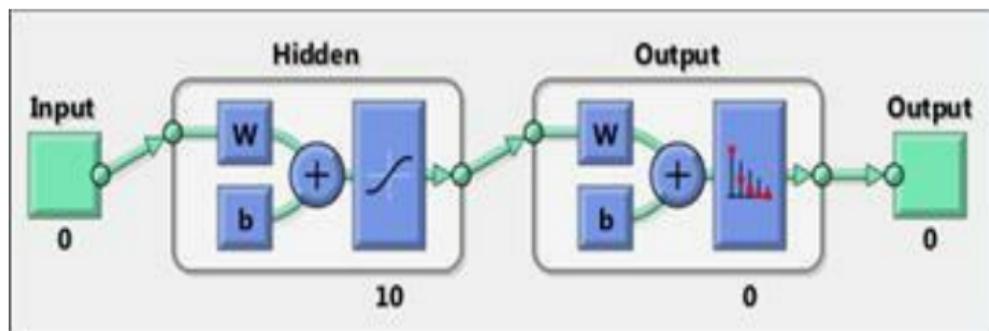
Two-layer (i.e. one-hidden-layer) feed forward neural networks can learn any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target

data. This will happen when the network is trained.

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



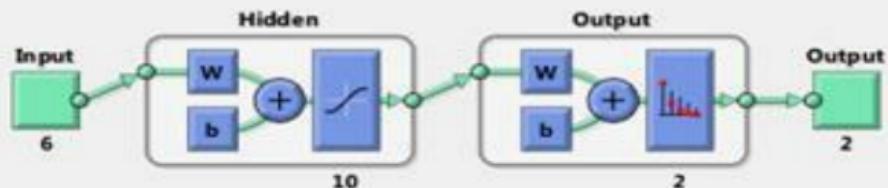
Now the network is ready to be trained.

The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

```
[net,tr] =  
train(net,x,t);
```

Nntraintool

## Neural Network



## Algorithms

Data Division: Random (dividerand)  
Training: Scaled Conjugate Gradient (trainscg)  
Performance: Cross-Entropy (crossentropy)  
Calculations: MEX

## Progress

Epoch:	0	45 iterations	1000
Time:		0:00:00	
Performance:	0.537	0.0170	0.00
Gradient:	0.440	0.0110	1.00e-06
Validation Checks:	0	6	6

## Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Confusion (plotconfusion)

Receiver Operating Characteristic (plotroc)

Plot Interval:



Validation stop.

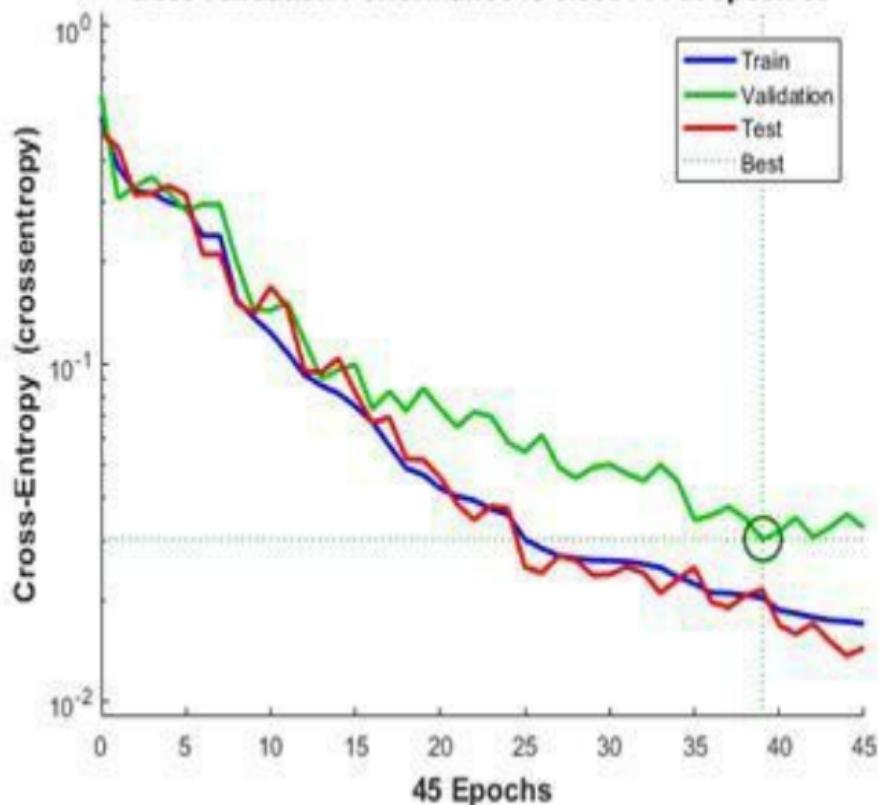
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

```
plotperform(tr)
```

**Best Validation Performance is 0.030144 at epoch 39**



## 14.1.4 Testing the Classifier

The trained neural network can now be tested with the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

The network outputs will be in the range 0 to 1, so we can use **vec2ind** function to get the class indices as the position of the highest element in each output vector.

```
testX =  
x(:, tr.testInd);
```

```
testT =  
t(:,tr.testInd);
```

```
testY = net(testX);
```

```
testIndices =  
vec2ind(testY)
```

```
testIndices =
```

Columns 1 through 13

2	2	1	2	2	2	2	1	2	2	2	1
---	---	---	---	---	---	---	---	---	---	---	---

Columns 14 through 26

2	1	1	2	2	2	1	2	1	1	1	2	1
---	---	---	---	---	---	---	---	---	---	---	---	---

Columns 27 through 30

1            2            2            1

One measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices

diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

plotconfusion(testT, · · ·)

Confusion Matrix			
Output Class	Target Class		
	1	2	
1	12 40.0%	0 0.0%	100% 0.0%
2	0 0.0%	18 60.0%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

Here are the overall percentages of

correct and incorrect classification.

```
[c, cm] =  
confusion(testT, testC)
```

```
fprintf('Percentage  
Correct  
Classification :  
%f%%\n', 100*(1-c));
```

```
fprintf('Percentage
```

Incorrect  
Classification :  
%f%%\n', 100\*c);

c =

0.0333

cm =

12              1

0              17

Percentage Correct  
Classification :

96.66667%

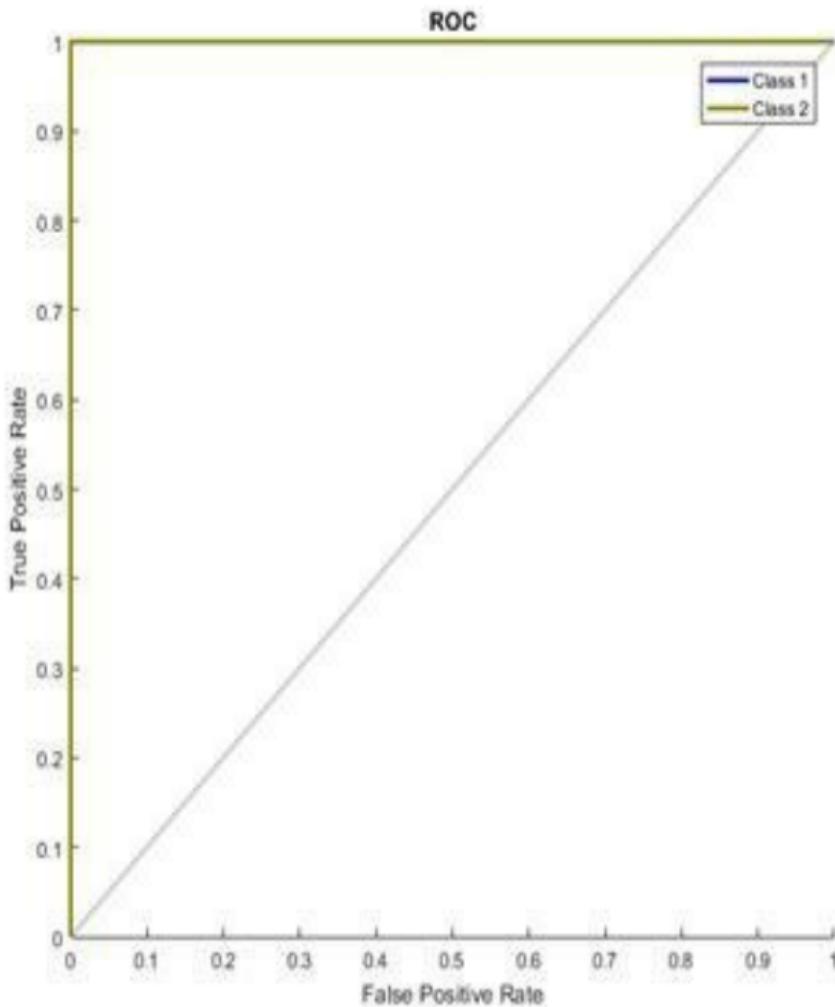
Percentage Incorrect Classification :  
3.333333%

Another measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true

positive rate. The best classifiers will have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

```
plotroc(testT, testY)
```



This example illustrated using a neural

network to classify crabs.

## 14.2 WINE CLASSIFICATION

This example illustrates how a pattern recognition neural network can classify wines by winery based on its chemical characteristics.

## 14.2.1 The Problem: Classify Wines

In this example we attempt to build a neural network that can classify wines from three wineries by thirteen attributes:

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols

- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

This is an example of a pattern recognition problem, where inputs are associated with different classes, and we would like to create a neural network that not only classifies the known wines properly, but can generalize to accurately classify wines that were not used to design the solution.

## 14.2.2 Why Neural Networks?

Neural networks are very good at pattern recognition problems. A neural network with enough elements (called neurons) can classify any data with arbitrary accuracy. They are particularly well suited for complex decision boundary problems over many variables. Therefore neural networks are a good candidate for solving the wine classification problem.

The thirteen neighborhood attributes will act as inputs to a neural network, and the respective target for each will be

a 3-element class vector with a 1 in the position of the associated winery, #1, #2 or #3.

The network will be designed by using the attributes of neighborhoods to train the network to produce the correct target classes.

## 14.2.3 Preparing the Data

Data for classification problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have thirteen elements representing a wine whose winery is already known.

Each corresponding column of the target matrix will have three elements, consisting of two zeros and a 1 in the location of the associated winery.

Here such a dataset is loaded.

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

We can view the sizes of inputs X and targets T.

Note that both X and T have 178 columns. These represent 178 wine sample attributes (inputs) and associated winery class vectors (targets).

Input matrix X has thirteen rows, for the thirteen attributes. Target matrix T has three rows, as for each example we have three possible wineries.

`size(x)`

`size(t)`

`ans =`

13 178

ans =

3 178

## 14.2.4 Pattern Recognition with a Neural Network

The next step is to create a neural network that will learn to classify the wines.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

```
setdemorandstream(391418381)
```

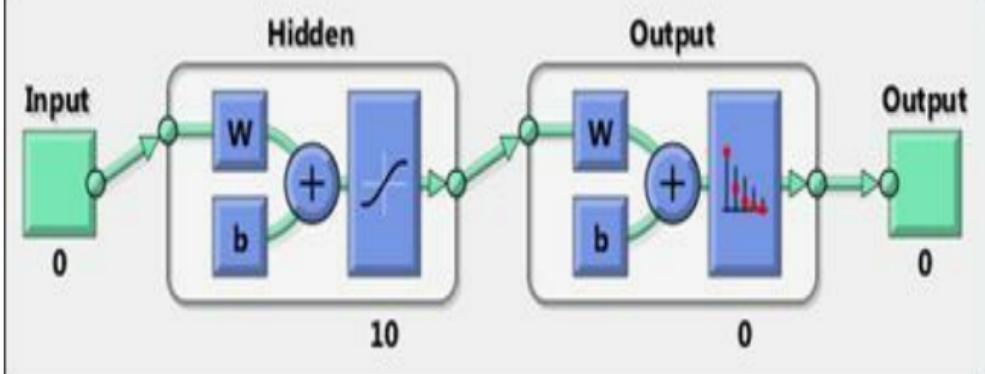
Two-layer (i.e. one-hidden-layer) feed forward neural networks can learn any input-output relationship given enough

neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

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



Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

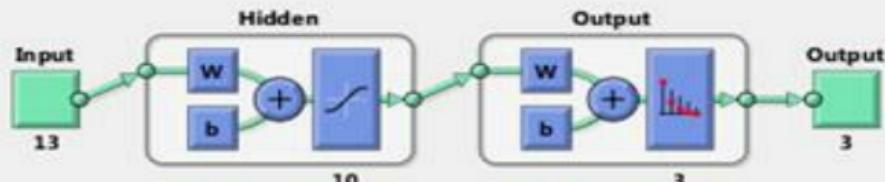
The NN Training Tool shows the network being trained and the algorithms

used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] = train(net,x,t);  
nntraintool
```

## Neural Network



## Algorithms

Data Division: Random (dividerand)  
Training: Scaled Conjugate Gradient (trainscg)  
Performance: Cross-Entropy (crossentropy)  
Calculations: MEX

## Progress

Epoch:	0	37 iterations	1000
Time:		0:00:00	
Performance:	0.635	2.71e-07	0.00
Gradient:	0.383	5.84e-07	1.00e-06
Validation Checks:	0	1	6

## Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Confusion (plotconfusion)

Receiver Operating Characteristic (plotroc)

Plot Interval:  1 epochs



Minimum gradient reached.



Stop Training



Cancel

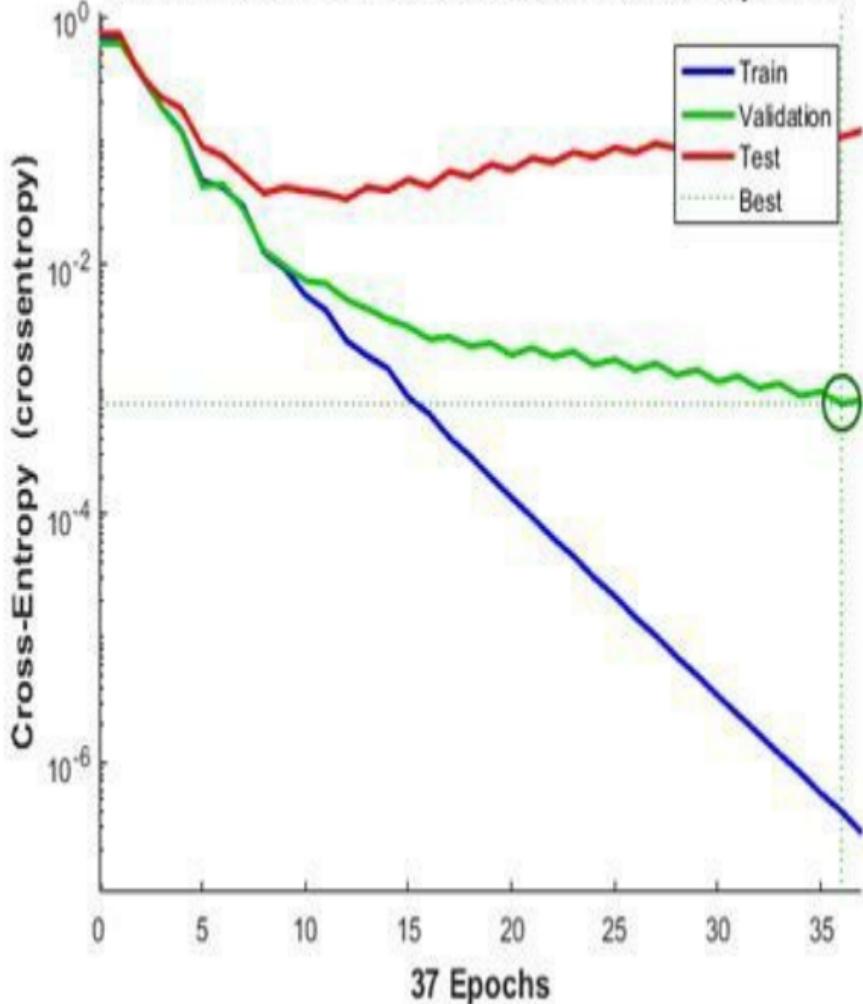
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is after training.

```
plotperform(tr)
```

**Best Validation Performance is 0.00076278 at epoch 36**



## 14.2.5 Testing the Neural Network

The mean squared error of the trained neural network can now be measured with respect to the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

The network outputs will be in the range 0 to 1, so we can use **vec2ind** function to get the class indices as the position of the highest element in each output vector.

```
testX = x(:,tr.testInd);
```

```
testT = t(:,tr.testInd);
```

```
testY = net(testX);  
testIndices = vec2ind(testY)  
  
testIndices =
```

Columns 1 through 13

	1	1	1	2	1	1	1	1	1	1	1
1	2	2									

Columns 14 through 26

	2	2	2	2	2	2	3	2	3	3
3	3	3								

Column 27

3

Another measure of how well the neural

network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

```
plotconfusion(testT,testY)
```

### Confusion Matrix

		Target Class			
		1	2	3	4
Output Class	1	10 37.0%	0 0.0%	0 0.0%	100% 0.0%
	2	1 3.7%	8 29.6%	1 3.7%	80.0% 20.0%
3	0 0.0%	0 0.0%	7 25.9%	100% 0.0%	
4	90.9% 9.1%	100% 0.0%	87.5% 12.5%	92.6% 7.4%	

Here are the overall percentages of correct and incorrect classification.

[c,cm] = confusion(testT,testY)

```
fprintf('Percentage Correct  
Classification : %f%%\n', 100*(1-c));  
fprintf('Percentage Incorrect  
Classification : %f%%\n', 100*c);
```

c =

0.0741

cm =

10 1 0

0	8	0
0	1	7

Percentage Correct Classification :  
92.592593%

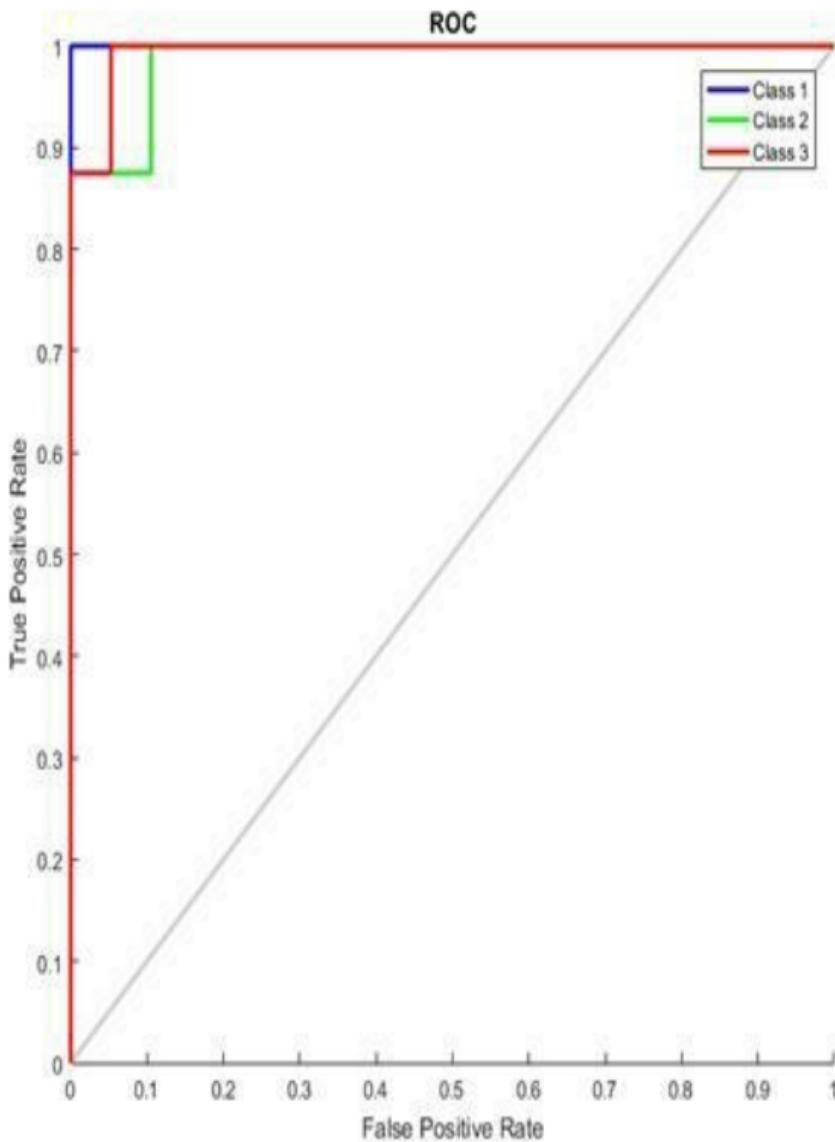
Percentage Incorrect Classification :  
7.407407%

A third measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true positive rate. The best classifiers will

have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

plotroc(testT,testY)



## 14.3 CANCER DETECTION

This example demonstrates using a neural network to detect cancer from mass spectrometry data on protein profiles.

Serum proteomic pattern diagnostics can be used to differentiate samples from patients with and without disease. Profile patterns are generated using surface-enhanced laser desorption and ionization (SELDI) protein mass spectrometry. This technology has the potential to improve clinical diagnostics tests for cancer pathologies.

The goal is to build a classifier that can

distinguish between cancer and control patients from the mass spectrometry data.

The methodology followed in this example is to select a reduced set of measurements or "features" that can be used to distinguish between cancer and control patients using a classifier.

These features will be ion intensity levels at specific mass/charge values.

## 14.3.1 Formatting the Data

The data in this example is from the FDA-NCI Clinical Proteomics Program Databank: <http://home.ccr.cancer.gov/nci>

To recreate the data in **ovarian\_dataset.mat** used in this example, download and uncompress the raw mass-spectrometry data from the FDA-NCI web site. Create the data file **OvarianCancerQAQCdataset.mat** either running script **msseqprocessing** in Bioinformatics Toolbox (TM) or by following the steps in the example **biodistcompdemo** (Batch processing with parallel computing). The new file contains

variables `Y`, `MZ` and `grp`.

Each column in `Y` represents measurements taken from a patient. There are 216 columns in `Y` representing 216 patients, out of which 121 are ovarian cancer patients and 95 are normal patients.

Each row in `Y` represents the ion intensity level at a specific mass-charge value indicated in `MZ`. There are 15000 mass-charge values in `MZ` and each row in `Y` represents the ion-intesity levels of the patients at that particular mass-charge value.

The variable `grp` holds the index information as to which of these samples

represent cancer patients and which ones represent normal patients.

An extensive description of this data set and excellent introduction to this promising technology can be found in [1] and [2].

## 14.3.2 Ranking Key Features

This is a typical classification problem in which the number of features is much larger than the number of observations, but in which no single feature achieves a correct classification, therefore we need to find a classifier which appropriately learns how to weight multiple features and at the same time produce a generalized mapping which is not over-fitted.

A simple approach for finding significant features is to assume that each M/Z value is independent and

compute a two-way t-test. **rankfeatures** returns an index to the most significant M/Z values, for instance 100 indices ranked by the absolute value of the test statistic.

To finish recreating the data from **ovarian\_dataset.mat**, load the **OvarianCancerQAQCdataset.mat** and Bioinformatics Toolbox to choose 100 highest ranked measurements as inputs **x**.

```
ind =  
rankfeatures(Y,grp,''  
  
x = Y(ind,:);
```

Define the targets  $t$  for the two classes as follows:

```
t =  
double(strcmp('Cancel',
```

```
t = [t; 1-t];
```

The preprocessing steps from the script and example listed above are intended to demonstrate a representative set of possible pre-processing and feature selection procedures. Using different steps or parameters may lead to different and possibly improved results of this example.

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

```
whos
```

Name

Size

Bytes Class

Attributes

t

2x216

3456

double

x

100x216

172800

double

Each column in x represents one of 216 different patients.

Each row in  $\mathbf{x}$  represents the ion intensity level at one of the 100 specific mass-charge values for each patient.

The variable  $\mathbf{t}$  has 2 rows of 216 values each of which are either [1;0], indicating a cancer patient, or [0;1] for a normal patient.

### 14.3.3 Classification Using a Feed Forward Neural Network

Now that you have identified some significant features, you can use this information to classify the cancer and normal samples.

Since the neural network is initialized with random initial weights, the results after training the network vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. However this is not necessary for your own applications.

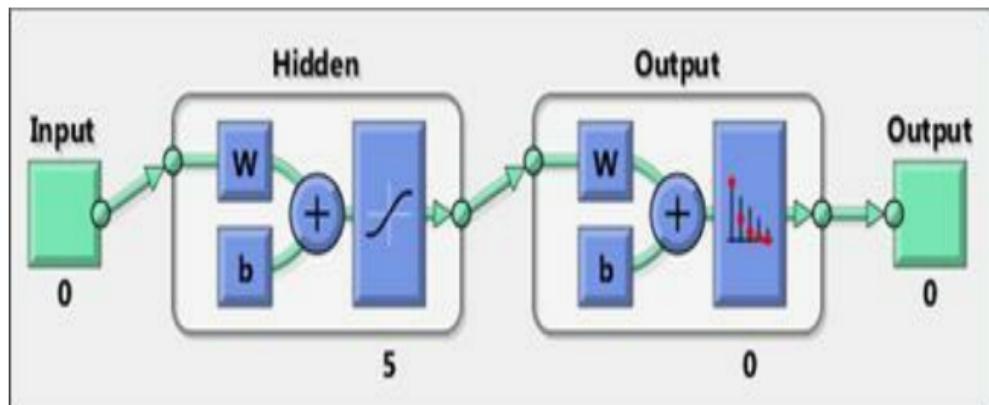
```
setdemorandstream(67);
```

A 1-hidden layer feed forward neural network with 5 hidden layer neurons is created and trained. The input and target samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target

data. This will happen when the network is trained.

```
net = patternnet(5);  
view(net)
```



Now the network is ready to be trained.

The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm

names and plot buttons open documentation on those subjects.

```
[net,tr] =  
train(net,x,t);
```

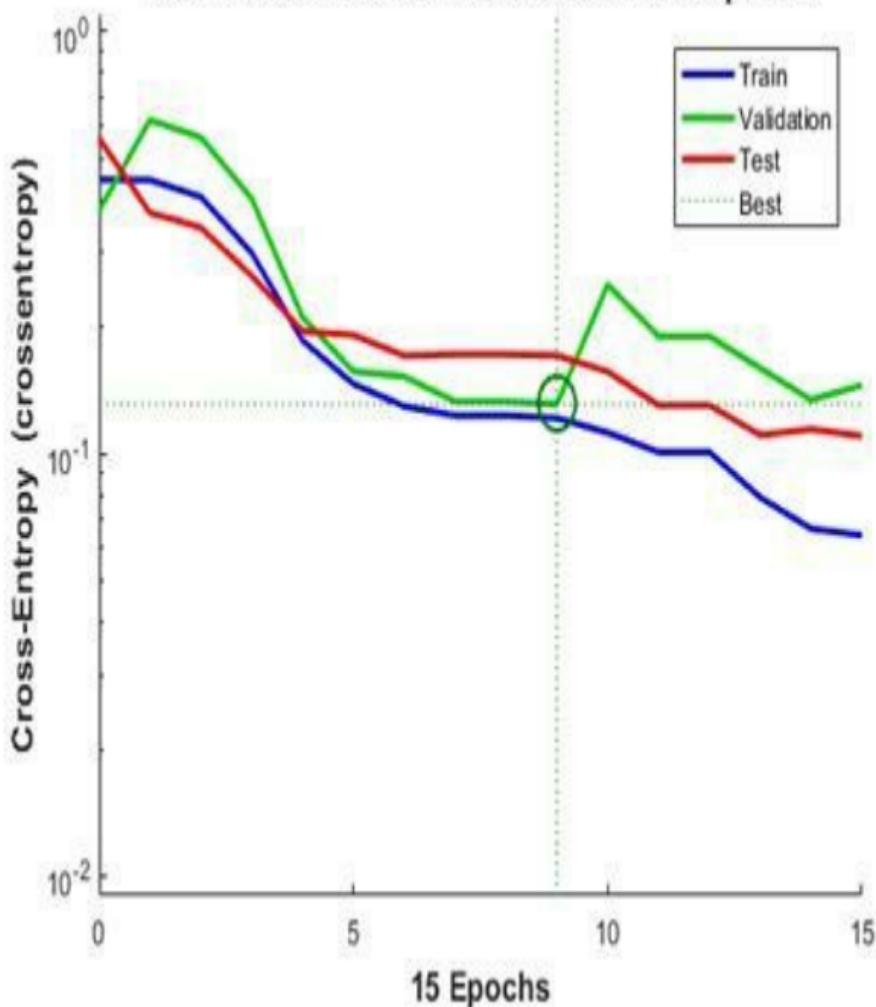
To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets. The version of the network that did best on the validation set is was after training.

```
plotperform(tr)
```

Best Validation Performance is 0.13105 at epoch 9



The trained neural network can now be tested with the testing samples we

partitioned from the main dataset. The testing data was not used in training in any way and hence provides an "out-of-sample" dataset to test the network on. This will give us a sense of how well the network will do when tested with data from the real world.

The network outputs will be in the range 0 to 1, so we threshold them to get 1's and 0's indicating cancer or normal patients respectively.

```
testX =  
x(:, tr.testInd);
```

```
testT =  
t(:,tr.testInd);
```

```
testY = net(testX);
```

```
testClasses = testY  
> 0.5
```

```
testClasses =
```

2×32 logical array

Columns 1 through 19

Columns 20 through 32

0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0								
1	1	1	1	0	1	1	1	1	1	1	1
1	1	1	1								

One measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications

are the green squares on the matrices diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

plotconfusion(testT, · · ·)

### Confusion Matrix

		Target Class
Output Class	1	1
		2
1	13 40.6%	5 15.6%
2	0 0.0%	14 43.8%
	100% 0.0%	73.7% 26.3%
		84.4% 15.6%

Here are the overall percentages of correct and incorrect classification.

```
[c, cm] =  
confusion(testT, test)  
  
fprintf('Percentage  
Correct  
Classification :  
%f%%\n', 100*(1-c));
```

```
fprintf ('Percentage  
Incorrect  
Classification :  
%f%%\n', 100*c);
```

c =

0.0938

cm =

16                  2

1                  13

Percentage Correct  
Classification :  
90.625000%

Percentage Incorrect Classification :  
9.375000%

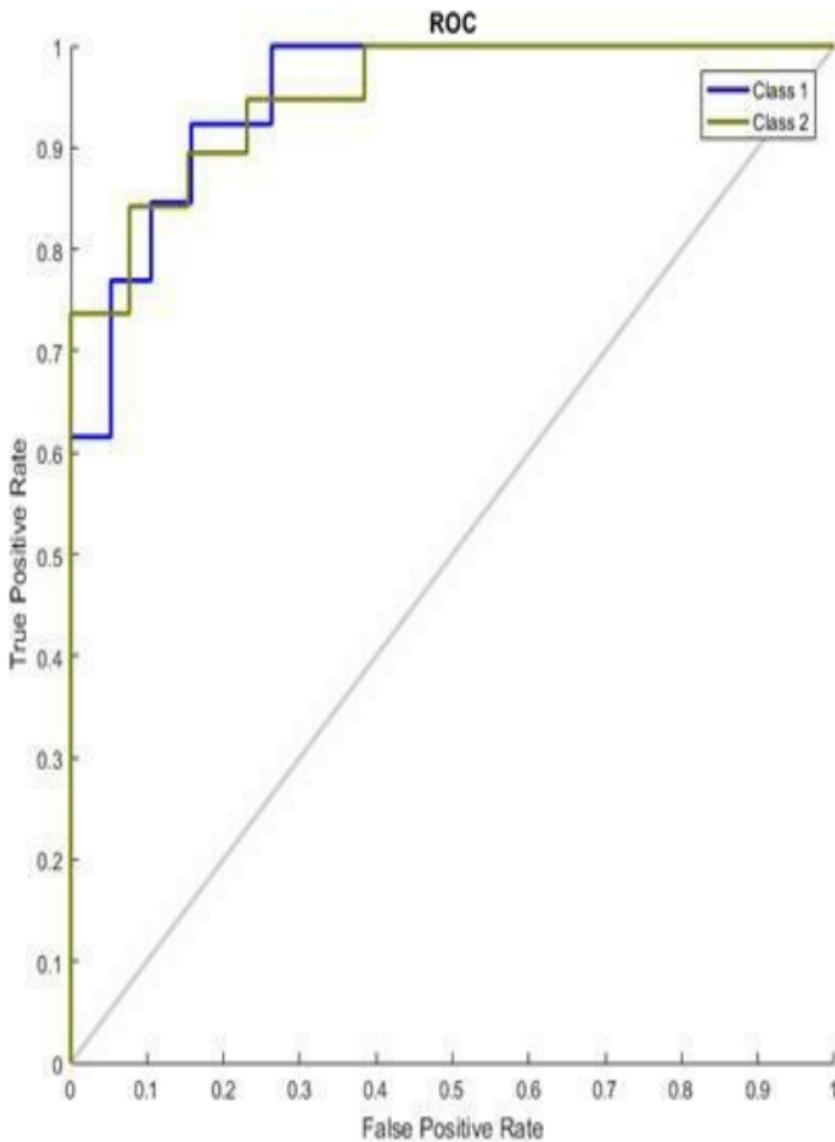
Another measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true

positive rate. The best classifiers will have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

Class 1 indicate cancer patients, class 2 normal patients.

```
plotroc(testT, testY)
```



This example illustrated how neural networks can be used as classifiers for cancer detection. One can also experiment using techniques like principal component analysis to reduce the dimensionality of the data to be used for building neural networks to improve classifier performance.

## 14.4 CHARACTER RECOGNITION

This example illustrates how to train a neural network to perform simple character recognition.

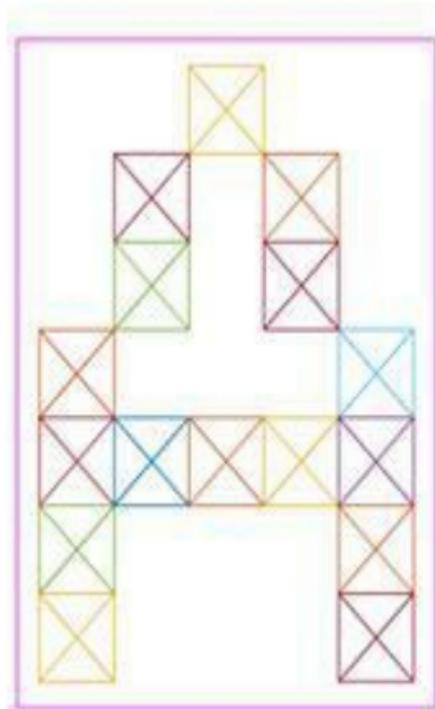
The script **prprob** defines a matrix X with 26 columns, one for each letter of the alphabet. Each column has 35 values which can either be 1 or 0. Each column of 35 values defines a 5x7 bitmap of a letter.

The matrix T is a 26x26 identity matrix which maps the 26 input vectors to the 26 classes.

[X,T] = prprob;

Here A, the first letter, is plotted as a bit map.

`plotchar(X(:,1))`



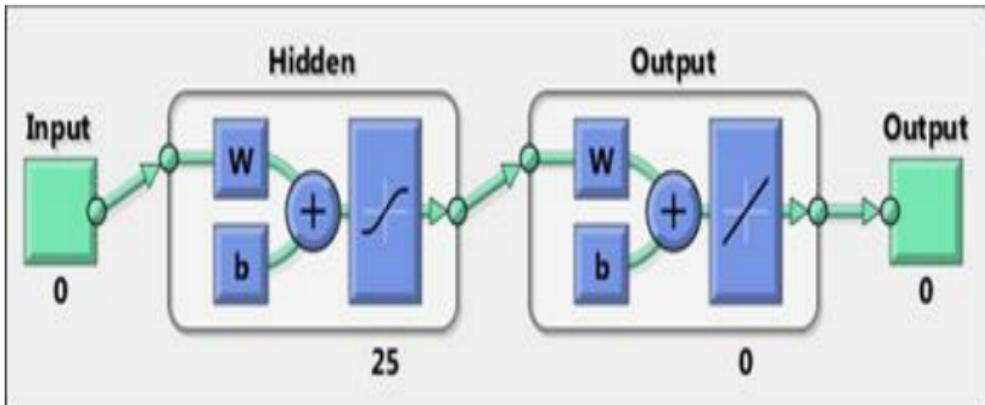
## 14.4.1 Creating the First Neural Network

To solve this problem we will use a feedforward neural network set up for pattern recognition with 25 hidden neurons.

Since the neural network is initialized with random initial weights, the results after training vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. This is not necessary for your own applications.

```
setdemorandstream(pi);
```

```
net1 = feedforwardnet(25);  
view(net1)
```



## 14.4.2 Training the first Neural Network

The function **train** divides up the data into training, validation and test sets. The training set is used to update the network, the validation set is used to stop the network before it overfits the training data, thus preserving good generalization. The test set acts as a completely independent measure of how well the network can be expected to do on new samples.

Training stops when the network is no longer likely to improve on the training or validation sets.

```
net1.divideFcn = "';  
net1 = train(net1,X,T,nnMATLAB);
```

## Computing Resources: MATLAB on GLNXA64

### 14.4.3 Training the Second Neural Network

We would like the network to not only recognize perfectly formed letters, but also noisy versions of the letters. So we will try training a second network on noisy data and compare its ability to generalize with the first network.

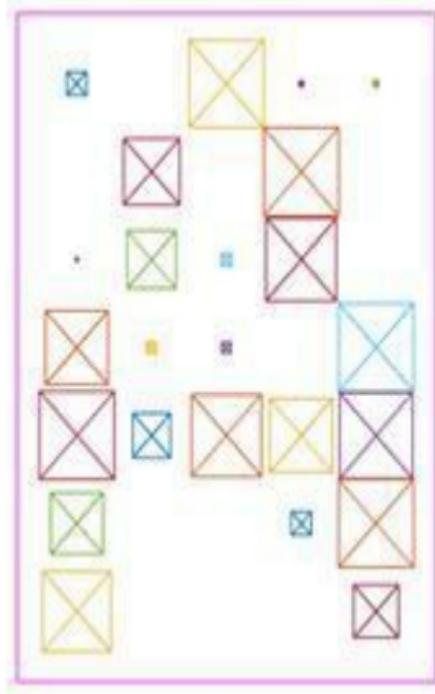
Here 30 noisy copies of each letter  $X_n$  are created. Values are limited by **min** and **max** to fall between 0 and 1. The corresponding targets  $T_n$  are also defined.

```
numNoise = 30;  
Xn =
```

```
min(max(repmat(X,1,numNoise)+randn(:));
Tn = repmat(T,1,numNoise);
```

Here is a noise version of A.

```
figure  
plotchar(Xn(:,1))
```



Here the second network is created and trained.

```
net2 = feedforwardnet(25);
```

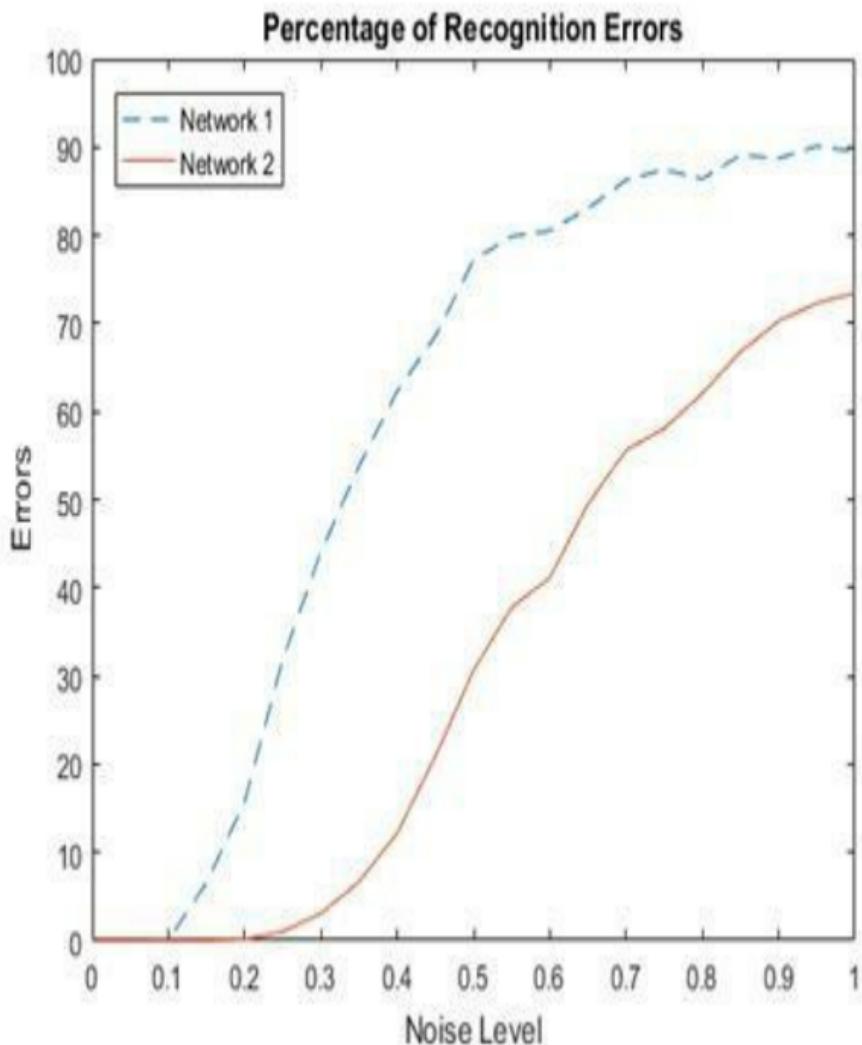
```
net2 = train(net2,Xn,Tn,nnMATLAB);
```

# Computing Resources: MATLAB on GLNXA64

## 14.4.4 Testing Both Neural Networks

```
noiseLevels = 0:.05:1;
numLevels = length(noiseLevels);
percError1 = zeros(1,numLevels);
percError2 = zeros(1,numLevels);
for i = 1:numLevels
    Xtest =
        min(max(repmat(X,1,numNoise)+randn(:,1)*noiseLevel));
    Y1 = net1(Xtest);
    percError1(i) = sum(sum(abs(Tn-compet(Y1))))/(26*numNoise^2);
    Y2 = net2(Xtest);
    percError2(i) = sum(sum(abs(Tn-compet(Y2))))/(26*numNoise^2);
end
```

```
figure  
plot(noiseLevels,percError1*100,'--  
' ,noiseLevels,percError2*100);  
title('Percentage of Recognition Errors');  
xlabel('Noise Level');  
ylabel('Errors');  
legend('Network 1','Network  
2','Location','NorthWest')
```



Network 1, trained without noise, has more errors due to noise than does

Network 2, which was trained with noise.

# **Chapter 15**

## **AUTOENCODERS AND CLUSTERING WITH NEURAL NETWORKS. EXAMPLES**



# **15.1 TRAIN STACKED AUTOENCODERS FOR IMAGE CLASSIFICATION**

This example shows how to use Neural Network Toolbox autoencoders functionality for training a deep neural network to classify images of digits.

Neural networks with multiple hidden layers can be useful for solving classification problems with complex data, such as images. Each layer can learn features at a different level of abstraction. However, training neural networks with multiple hidden layers

can be difficult in practice.

One way to effectively train a neural network with multiple layers is by training one layer at a time. You can achieve this by training a special type of network known as an autoencoder for each desired hidden layer.

This example shows you how to train a neural network with two hidden layers to classify digits in images. First you train the hidden layers individually in an unsupervised fashion using autoencoders. Then you train a final softmax layer, and join the layers together to form a deep network, which you train one final time in a supervised fashion.

## 15.1.1 Data set

This example uses synthetic data throughout, for training and testing. The synthetic images have been generated by applying random affine transformations to digit images created using different fonts.

Each digit image is 28-by-28 pixels, and there are 5,000 training examples. You can load the training data, and view some of the images.

```
% Load the training data into memory  
[xTrainImages,tTrain] =  
digitTrainCellArrayData;
```

```
% Display some of the training images
clf
for i = 1:20
    subplot(4,5,i);
    imshow(xTrainImages{i});
end
```



The labels for the images are stored in a 10-by-5000 matrix, where in every

column a single element will be 1 to indicate the class that the digit belongs to, and all other elements in the column will be 0. It should be noted that if the tenth element is 1, then the digit image is a zero.

## 15.1.2 Training the first autoencoder

Begin by training a sparse autoencoder on the training data without using the labels.

An autoencoder is a neural network which attempts to replicate its input at its output. Thus, the size of its input will be the same as the size of its output. When the number of neurons in the hidden layer is less than the size of the input, the autoencoder learns a compressed representation of the input.

Neural networks have weights randomly initialized before training. Therefore the

results from training are different each time. To avoid this behavior, explicitly set the random number generator seed.

```
rng('default')
```

Set the size of the hidden layer for the autoencoder. For the autoencoder that you are going to train, it is a good idea to make this smaller than the input size.

```
hiddenSize1 = 100;
```

The type of autoencoder that you will train is a sparse autoencoder. This autoencoder uses regularizers to learn a sparse representation in the first layer. You can control the influence of these regularizers by setting various parameters:

- L2WeightRegularization controls the impact of an L2 regularizer for the weights of the network (and not the biases). This should typically be quite small.

- SparsityRegularization controls the impact of a sparsity regularizer, which attempts to enforce a constraint on the sparsity of the output from the hidden layer. Note that this is different from applying a sparsity regularizer to the weights.

- SparsityProportion is a parameter of the sparsity regularizer. It controls the sparsity of the output from the hidden layer. A low value

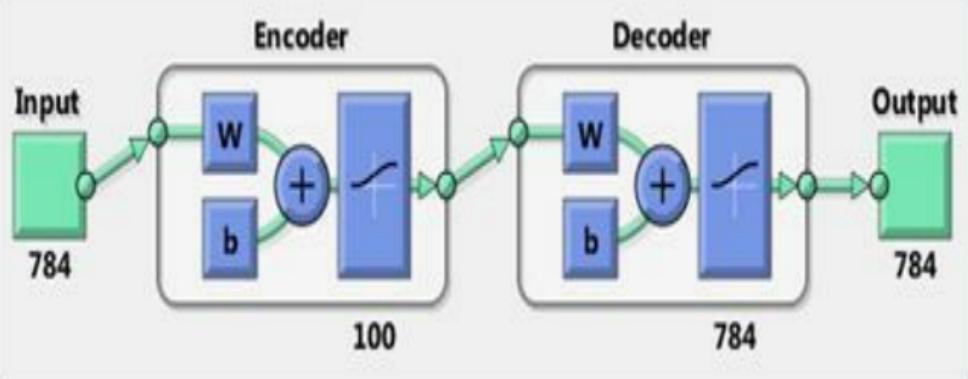
for SparsityProportion usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples. For example, if SparsityProportion is set to 0.1, this is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples. This value must be between 0 and 1. The ideal value varies depending on the nature of the problem.

Now train the autoencoder, specifying the values for the regularizers that are described above.

```
autoenc1 =  
trainAutoencoder(xTrainImages,hiddenSi  
...  
'MaxEpochs',400, ...  
'L2WeightRegularization',0.004, ...  
'SparsityRegularization',4, ...  
'SparsityProportion',0.15, ...  
'ScaleData', false);
```

You can view a diagram of the autoencoder. The autoencoder is comprised of an encoder followed by a decoder. The encoder maps an input to a hidden representation, and the decoder attempts to reverse this mapping to reconstruct the original input.

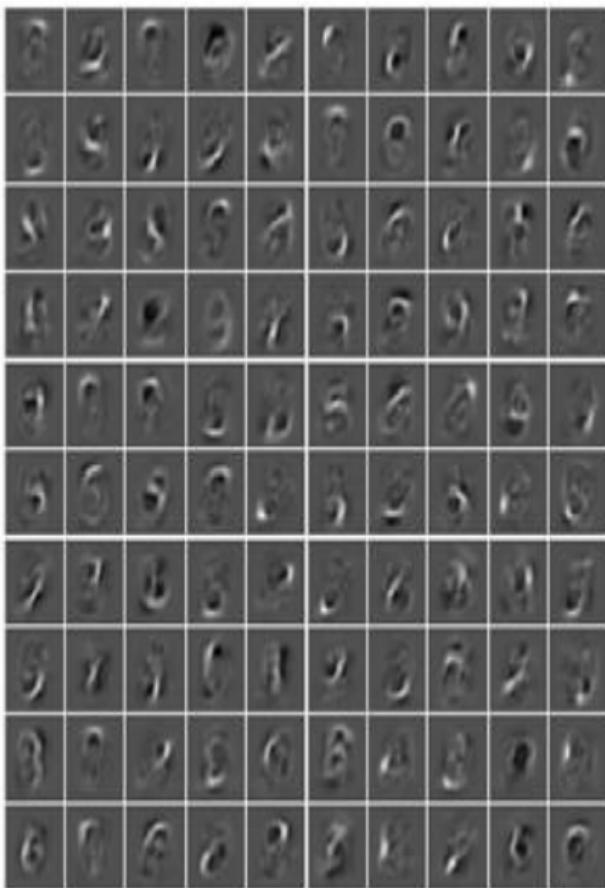
```
view(autoenc1)
```



## 15.1.3 Visualizing the weights of the first autoencoder

The mapping learned by the encoder part of an autoencoder can be useful for extracting features from data. Each neuron in the encoder has a vector of weights associated with it which will be tuned to respond to a particular visual feature. You can view a representation of these features.

```
figure()  
plotWeights(autoenc1);
```



You can see that the features learned by the autoencoder represent curls and

stroke patterns from the digit images.

The 100-dimensional output from the hidden layer of the autoencoder is a compressed version of the input, which summarizes its response to the features visualized above. Train the next autoencoder on a set of these vectors extracted from the training data. First, you must use the encoder from the trained autoencoder to generate the features.

```
feat1 = encode(autoenc1,xTrainImages);
```

## 15.1.4 Training the second autoencoder

After training the first autoencoder, you train the second autoencoder in a similar way. The main difference is that you use the features that were generated from the first autoencoder as the training data in the second autoencoder. Also, you decrease the size of the hidden representation to 50, so that the encoder in the second autoencoder learns an even smaller representation of the input data.

```
hiddenSize2 = 50;
```

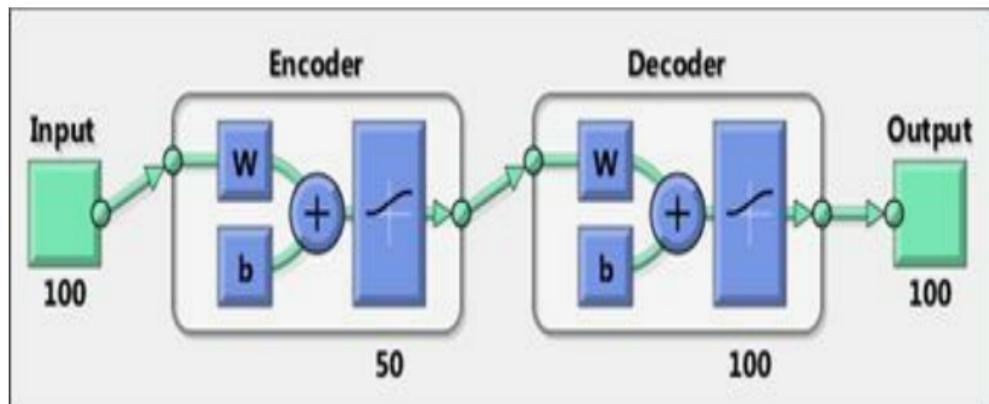
```
autoenc2 =
```

```
trainAutoencoder(feat1,hiddenSize2, ...)
```

```
'MaxEpochs',100, ...
'L2WeightRegularization',0.002, ...
'SparsityRegularization',4, ...
'SparsityProportion',0.1, ...
'ScaleData', false);
```

Once again, you can view a diagram of the autoencoder with the view function.

```
view(autoenc2)
```



You can extract a second set of features by passing the previous set through the

encoder from the second autoencoder.

```
feat2 = encode(autoenc2,feat1);
```

The original vectors in the training data had 784 dimensions. After passing them through the first encoder, this was reduced to 100 dimensions. After using the second encoder, this was reduced again to 50 dimensions. You can now train a final layer to classify these 50-dimensional vectors into different digit classes.

## 15.1.5 Training the final softmax layer

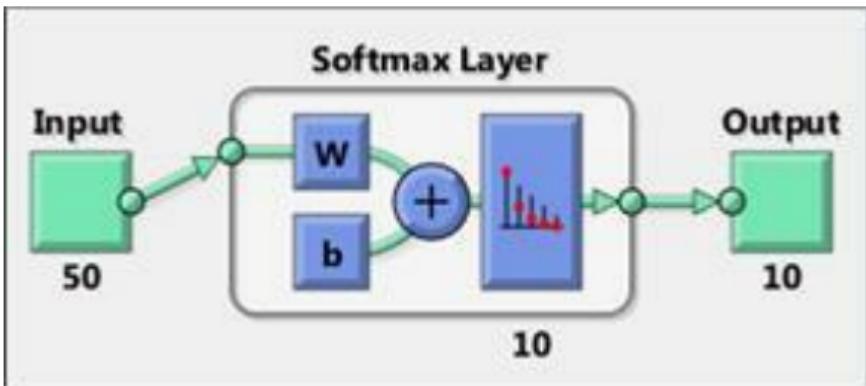
Train a softmax layer to classify the 50-dimensional feature vectors. Unlike the autoencoders, you train the softmax layer in a supervised fashion using labels for the training data.

softnet =

```
trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',10)
```

You can view a diagram of the softmax layer with the `view` function.

```
view(softnet)
```



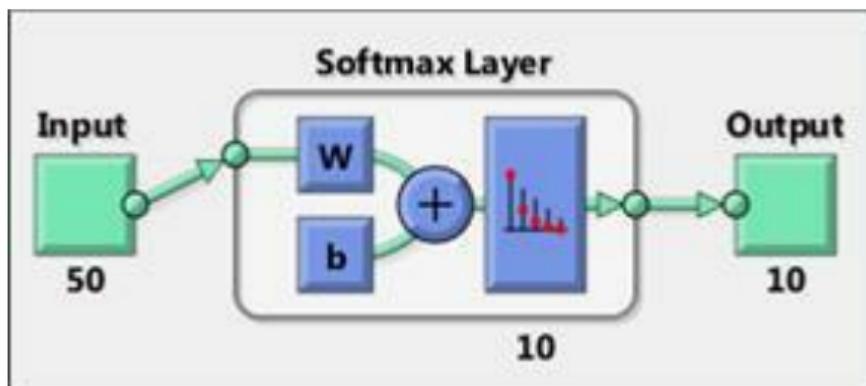
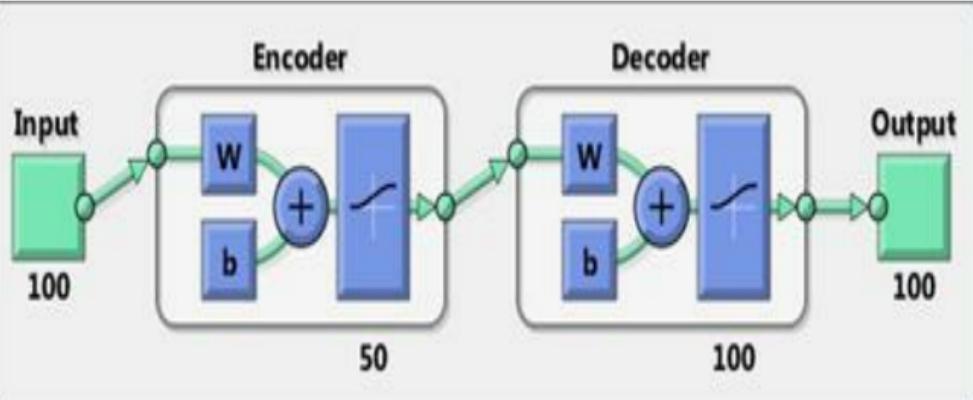
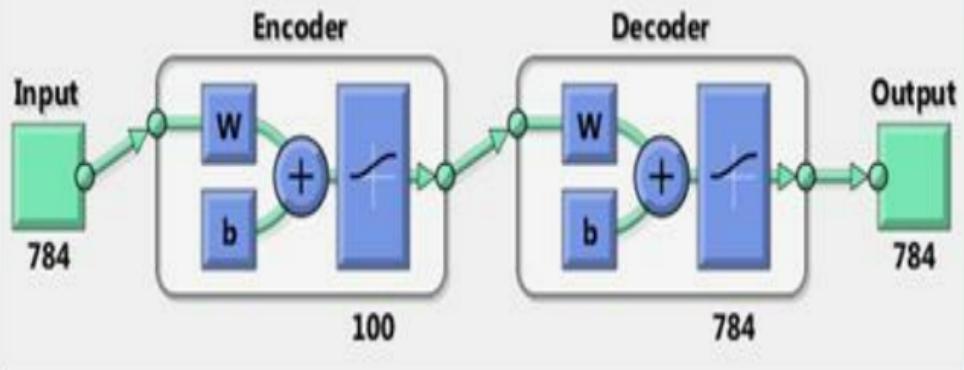
## 15.1.6 Forming a stacked neural network

You have trained three separate components of a deep neural network in isolation. At this point, it might be useful to view the three neural networks that you have trained. They are autoenc1, autoenc2, and softnet.

`view(autoenc1)`

`view(autoenc2)`

`view(softnet)`



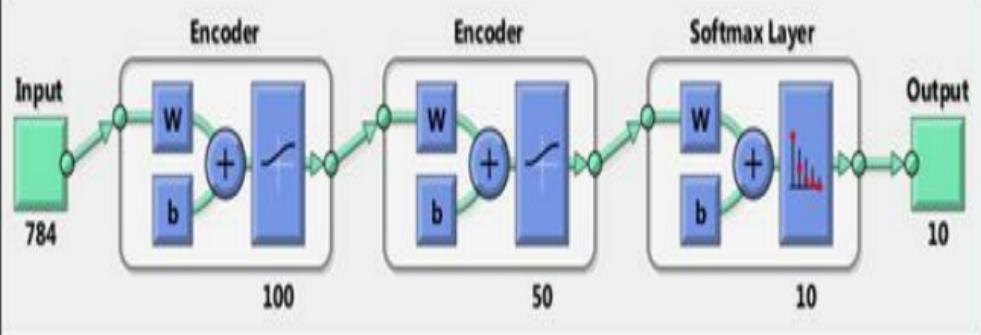
As was explained, the encoders from the autoencoders have been used to extract features. You can stack the encoders from the autoencoders together with the softmax layer to form a deep network.

```
deepnet =
```

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

You can view a diagram of the stacked network with the `view` function. The network is formed by the encoders from the autoencoders and the softmax layer.

```
view(deepnet)
```



With the full deep network formed, you can compute the results on the test set. To use images with the stacked network, you have to reshape the test images into a matrix. You can do this by stacking the columns of an image to form a vector, and then forming a matrix from these vectors.

```
% Get the number of pixels in each  
image  
imageWidth = 28;  
imageHeight = 28;
```

```
inputSize = imageSize*imageHeight;  
  
% Load the test images  
[xTestImages,tTest] =  
digitTestCellArrayData;  
  
% Turn the test images into vectors and  
put them in a matrix  
xTest =  
zeros(inputSize,numel(xTestImages));  
for i = 1:numel(xTestImages)  
    xTest(:,i) = xTestImages{i}(:);  
end
```

You can visualize the results with a confusion matrix. The numbers in the bottom right-hand square of the matrix give the overall accuracy.

```
y = deepnet(xTest);  
plotconfusion(tTest,y);
```

Confusion Matrix

Output Class	Target Class										Overall Accuracy (%)
	1	2	3	4	5	6	7	8	9	10	
1	448 9.0%	9 0.2%	0 0.0%	1 0.0%	3 0.1%	8 0.2%	14 0.3%	11 0.2%	0 0.0%	3 0.1%	90.1% 9.9%
2	3 0.1%	447 8.9%	14 0.3%	4 0.1%	0 0.0%	0 0.0%	14 0.3%	20 0.4%	6 0.1%	17 0.3%	85.1% 14.9%
3	6 0.1%	21 0.4%	338 6.8%	1 0.0%	49 1.0%	2 0.0%	7 0.1%	41 0.8%	3 0.1%	4 0.1%	71.6% 28.4%
4	7 0.1%	1 0.0%	5 0.1%	472 9.4%	1 0.0%	8 0.2%	0 0.0%	1 0.0%	5 0.1%	1 0.0%	94.2% 5.8%
5	0 0.0%	2 0.0%	61 1.2%	2 0.0%	411 8.2%	26 0.5%	0 0.0%	60 1.2%	1 0.0%	2 0.0%	72.7% 27.3%
6	19 0.4%	0 0.0%	5 0.1%	5 0.1%	6 0.1%	409 8.2%	3 0.1%	34 0.7%	9 0.2%	16 0.3%	80.8% 19.2%
7	30 0.6%	10 0.2%	6 0.1%	2 0.0%	0 0.0%	0 0.0%	450 9.0%	5 0.1%	1 0.0%	2 0.0%	88.9% 11.1%
8	0 0.0%	0 0.0%	42 0.8%	2 0.0%	20 0.4%	19 0.4%	7 0.1%	303 6.1%	5 0.1%	27 0.5%	71.3% 28.7%
9	0 0.0%	1 0.0%	16 0.3%	6 0.1%	6 0.1%	9 0.2%	2 0.0%	16 0.3%	461 9.2%	7 0.1%	88.0% 12.0%
10	0 0.0%	10 0.2%	9 0.2%	0 0.0%	12 0.2%	15 0.3%	2 0.0%	10 0.2%	3 0.1%	415 8.3%	87.2% 12.8%
	87.3% 12.7%	89.2% 10.8%	88.1% 31.9%	85.4% 4.6%	80.9% 19.1%	82.5% 17.5%	80.2% 9.8%	80.5% 39.5%	83.3% 6.7%	84.0% 16.0%	83.1% 16.9%

## 15.1.7 Fine tuning the deep neural network

The results for the deep neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine tuning.

You fine tune the network by retraining it on the training data in a supervised fashion. Before you can do this, you have to reshape the training images into a matrix, as was done for the test images.

```
% Turn the training images into vectors  
and put them in a matrix
```

```
xTrain =  
zeros(inputSize,numel(xTrainImages));  
for i = 1:numel(xTrainImages)  
    xTrain(:,i) = xTrainImages{i}(:);  
end
```

```
% Perform fine tuning  
deepnet = train(deepnet,xTrain,tTrain);
```

You then view the results again using a confusion matrix.

```
y = deepnet(xTest);  
plotconfusion(tTest,y);
```

## Confusion Matrix

	Target Class										
	1	2	3	4	5	6	7	8	9	10	
1	511 10.2%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	1 0.0%	0 0.0%	0 0.0%	99.6% 0.4%
2	0 0.0%	501 10.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	99.6% 0.4%
3	0 0.0%	0 0.0%	496 9.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	494 9.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
5	0 0.0%	0 0.0%	0 0.0%	508 10.2%	1 0.0%	0 0.0%	3 0.1%	0 0.0%	0 0.0%	0 0.0%	99.2% 0.8%
6	0 0.0%	0 0.0%	0 0.0%	0 0.0%	493 9.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
7	2 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	498 10.0%	0 0.0%	0 0.0%	0 0.0%	99.4% 0.6%
8	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2 0.0%	0 0.0%	496 9.9%	0 0.0%	0 0.0%	99.6% 0.4%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	494 9.9%	0 0.0%	100% 0.0%
10	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	493 9.9%	100% 0.0%
	99.6% 0.4%	100% 0.0%	100% 0.0%	99.8% 0.2%	100% 0.0%	99.4% 0.6%	99.8% 0.2%	99.0% 1.0%	100% 0.0%	99.8% 0.2%	99.7% 0.3%

## 15.1.8 Summary

This example showed how to train a deep neural network to classify digits in images using Neural Network Toolbox™. The steps that have been outlined can be applied to other similar problems, such as classifying images of letters, or even small images of objects of a specific category.

## **15.2 TRANSFER LEARNING USING CONVOLUTIONAL NEURAL NETWORKS**

Fine-tune a convolutional neural network pretrained on digit images to learn the features of letter images. Transfer learning is considered as the transfer of knowledge from one learned task to a new task in machine learning [1]. In the context of neural networks, it is transferring learned features of a pretrained network to a new problem. Training a convolutional neural network from the beginning in each case usually

is not effective when there is not sufficient amount of training data. The common practice in deep learning for such cases is to use a network that is trained on a large data set for a new problem. While the initial layers of the pretrained network can be fixed, the last few layers must be fine-tuned to learn the specific features of the new data set. Transfer learning usually results in faster training times than training a new convolutional neural network because you do not need to estimate all the parameters in the new network.

**NOTE:** Training a convolutional neural network requires Parallel Computing Toolbox™ and a CUDA®-enabled

NVIDIA® GPU with compute capability 3.0 or higher.

Load the sample data as an ImageDatastore.

```
digitDatasetPath =  
fullfile(matlabroot,'toolbox','nnet','nnDEM  
'nnDatasets','DigitDataset');  
digitData =  
 imageDatastore(digitDatasetPath,...
```

```
'IncludeSubfolders',true,'LabelSource','fo
```

The data store contains 10000 synthetic images of digits 0–9. The images are generated by applying random transformations to digit images created using different fonts. Each digit image is

28-by-28 pixels.

Display some of the images in the datastore.

```
for i = 1:20
```

```
    subplot(4,5,i);
```

```
    imshow(digitData.Files{i});
```

```
end
```



Check the number of images in each digit category.

`digitData.countEachLabel`

`ans =`

Label	Count
-------	-------

0	988
1	1026
2	1003
3	993
4	991
5	1017
6	992
7	999
8	1003
9	988

The data contains an unequal number of images per category.

To balance the number of images for each digit in the training set, first find the minimum number of images in a category.

`minSetCount =`

`min(digitData.countEachLabel {:,2})`

`minSetCount =`

988

Divide the dataset so that each category in the training set has 494 images and the testing set has the remaining images from each label.

`trainingNumFiles =`

`round(minSetCount/2);`

```
rng(1) % For reproducibility  
[trainDigitData,testDigitData] =  
splitEachLabel(digitData,...  
  
trainingNumFiles,'randomize');  
  
splitEachLabel splits the image files  
in digitData into two new  
datastores, trainDigitData and testDigitL  
Create the layers for the convolutional  
neural network.  
  
layers = [imageInputLayer([28 28 1])  
    convolution2dLayer(5,20)  
    reluLayer()  
    maxPooling2dLayer(2,'Stride',2)  
    fullyConnectedLayer(10)]
```

```
softmaxLayer()  
classificationLayer());
```

Create the training options. Set the maximum number of epochs at 20, and start the training with an initial learning rate of 0.001.

```
options =  
trainingOptions('sgdm','MaxEpochs',20,..  
'InitialLearnRate',0.001);
```

Train the network using the training set and the options you defined in the previous step.

```
convnet =  
trainNetwork(trainDigitData,layers,options)
```

---

---

Epoch	Iteration	Time Elapsed	Mini-batch	Mini-
-------	-----------	--------------	------------	-------

batch	Base Learning				
		(seconds)	Loss	Accuracy	
Rate					
0.001000	2	50	0.71	0.2233	92.97%
0.001000	3	100	1.37	0.0182	99.22%
0.001000	4	150	2.02	0.0395	99.22%
0.001000	6	200	2.70	0.0105	99.22%
0.001000	7	250	3.35	0.0026	100.00%
0.001000	8	300	4.00	0.0004	100.00%
0.001000	10	350	4.67	0.0002	100.00%
0.001000	11	400	5.32	0.0001	100.00%
0.001000	12	450	5.95	0.0001	100.00%
0.001000	14	500	6.60	0.0002	100.00%
0.001000	15	550	7.23	0.0001	100.00%
0.001000	16	600	7.87	0.0001	100.00%

	18	650	8.52	0.0001	100.00%
0.001000					
	19	700	9.15	0.0001	100.00%
0.001000					
	20	750	9.79	0.0000	100.00%
0.001000					

Test the network using the testing set and compute the accuracy.

```
YTest = classify(convnet,testDigitData);
```

```
TTest = testDigitData.Labels;
```

```
accuracy = sum(YTest ==
```

```
TTest)/numel(YTest)
```

```
accuracy =
```

0.9976

Accuracy is the ratio of the number of true labels in the test data matching the

classifications from classify, to the number of images in the test data. In this case 99.78% of the digit estimations match the true digit values in the test set.

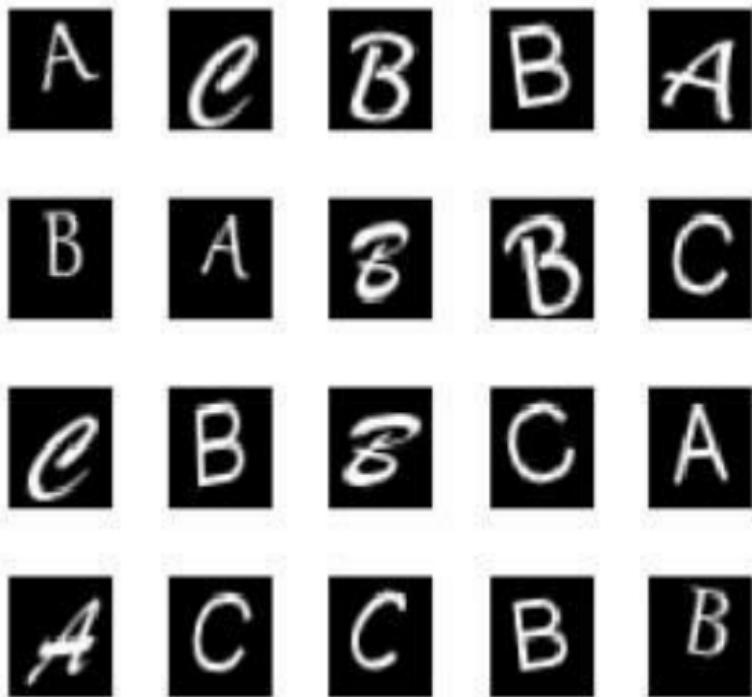
Now, suppose you would like to use the trained network net to predict classes on a new set of data. Load the letters training data.

```
load lettersTrainSet.mat
```

XTrain contains 1500 28-by-28 grayscale images of the letters A, B, and C in a 4-D array. TTrain contains the categorical array of the letter labels.

Display some of the letter images.

```
figure;
for j = 1:20
    subplot(4,5,j);
    selectImage =
datasample(XTrain,1,4);
    imshow(selectImage,[ ]);
end
```



The pixel values in XTrain are in the range [0 1]. The digit data used in training the network net were in [0 255]; scale the letters data between [0 255].

```
XTrain = XTrain*255;
```

The last three layers of the trained network net are tuned for the digit dataset, which has 10 classes. The properties of these layers depend on the classification task. Display the fully connected layer (fullyConnectedLayer).

```
convnet.Layers(end-2)
```

```
ans =
```

FullyConnectedLayer with properties:

Name: 'fc'

Hyperparameters

InputSize: 2880

OutputSize: 10

Learnable Parameters

Weights: [10×2880 single]

Bias: [10×1 single]

Use properties method to see a list of all properties.

Display the last layer  
(classificationLayer).

convnet.Layers(end)

ans =

ClassificationOutputLayer with  
properties:

Name: 'classoutput'  
classNames: {10×1 cell}  
OutputSize: 10

Hyperparameters

LossFunction: 'crossentropyex'

These three layers must be fine-tuned for the new classification problem. Extract all the layers but the last three from the trained network, net.

```
layersTransfer = convnet.Layers(1:end-3);
```

The letters data set has three classes. Add a new fully connected layer for three classes, and increase the learning

rate for this layer.

```
layersTransfer(end+1) =  
fullyConnectedLayer(3,...  
    'WeightLearnRateFactor',10,...  
    'BiasLearnRateFactor',20);
```

WeightLearnRateFactor and BiasLearnR  
multipliers of the global learning rate for  
the fully connected layer.

Add a softmax layer and a classification  
output layer.

```
layersTransfer(end+1) = softmaxLayer();  
layersTransfer(end+1) =  
classificationLayer();
```

Create the options for transfer learning.  
You do not have to train for many epochs

(MaxEpochs can be lower than before).

Set the InitialLearnRate at a lower rate than used for training net to improve convergence by taking smaller steps.

optionsTransfer =

trainingOptions('sgdm',...

'MaxEpochs',5,...

'InitialLearnRate',0.000005,...

'Verbose',true);

Perform transfer learning.

convnetTransfer =

trainNetwork(XTrain,TTrain,...

layersTransfer,optionsTransfer);

Epoch	Iteration	Time Elapsed	Mini-batch	Mini-
-------	-----------	--------------	------------	-------

batch	Base Learning				
		(seconds)	Loss	Accuracy	
Rate					
5	50	0.43	0.0011	100.00%	
0.000005					

Load the letters test data. Similar to the letters training data, scale the testing data between [0 255], because the training data were between that range.

load lettersTestSet.mat

XTest = XTest\*255;

Test the accuracy.

YTest =

classify(convnetTransfer,XTest);

accuracy = sum(YTest ==

TTest)/numel(TTest)

accuracy =

0.9587

## 15.3 IRIS CLUSTERING

This example illustrates how a self-organizing map neural network can cluster iris flowers into classes topologically, providing insight into the types of flowers and a useful tool for further analysis.

In this example we attempt to build a neural network that clusters iris flowers into natural classes, such that similar classes are grouped together. Each iris is described by four features:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm

- Petal width in cm

This is an example of a clustering problem, where we would like to group samples into classes based on the similarity between samples. We would like to create a neural network which not only creates class definitions for the known inputs, but will let us classify unknown inputs accordingly.

## 15.3.1 Why Self-Organizing Map Neural Networks?

Self-organizing maps (SOMs) are very good at creating classifications. Further, the classifications retain topological information about which classes are most similar to others. Self-organizing maps can be created with any desired level of detail. They are particularly well suited for clustering data in many dimensions and with complexly shaped and connected feature spaces. They are well suited to cluster iris flowers.

The four flower attributes will act as inputs to the SOM, which will map them

onto a 2-dimensional layer of neurons.

## 15.3.2 Preparing the Data

Data for clustering problems are set up for a SOM by organizing the data into an input matrix X.

Each ith column of the input matrix will have four elements representing the four measurements taken on a single flower.

Here such a dataset is loaded.

```
x = iris_dataset;
```

We can view the size of inputs X.

Note that X has 150 columns. These represent 150 sets of iris flower attributes. It has four rows, for the four

measurements.

size(x)

ans =

4 150

### 15.3.3 Clustering with a Neural Network

The next step is to create a neural network that will learn to cluster.

**selforgmap** creates self-organizing maps for classify samples with as much detailed as desired by selecting the number of neurons in each dimension of the layer.

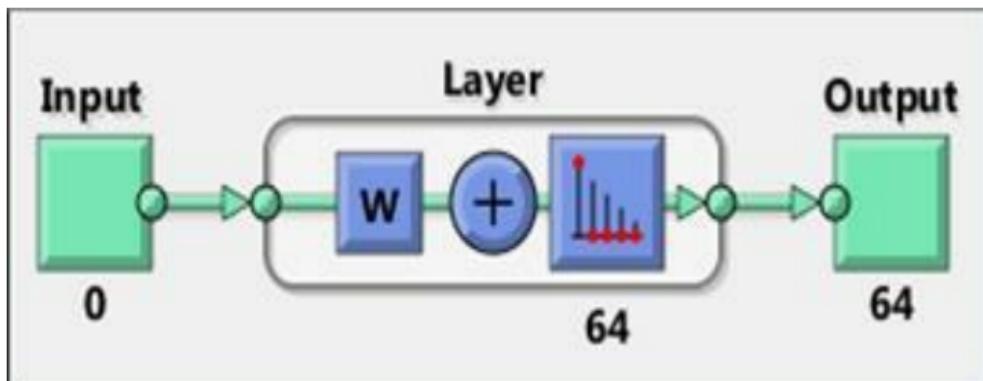
We will try a 2-dimension layer of 64 neurons arranged in an 8x8 hexagonal grid for this example. In general, greater detail is achieved with more neurons, and more dimensions allows for the modelling the topology of more complex

feature spaces.

The input size is 0 because the network has not yet been configured to match our input data. This will happen when the network is trained.

```
net = selforgmap([8  
8]);
```

```
view(net)
```



Now the network is ready to be optimized with **train**.

The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

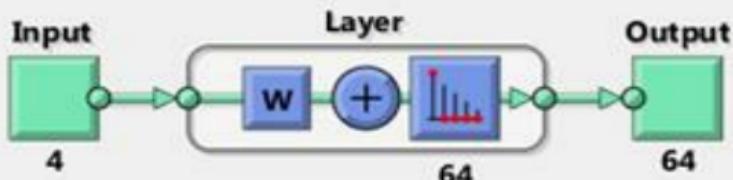
The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net, tr] =
```

```
train(net,x);
```

```
nntraintool
```

## Neural Network



### Algorithms

Training: Batch Weight/Bias Rules (trainbu)  
Performance: Mean Squared Error (mse)  
Calculations: MATLAB

### Progress

Epoch: 0 200 iterations 200  
Time: 0:00:01

### Plots

[SOM Topology](#) (`plotsomtop`)

[SOM Neighbor Connections](#) (`plotsomnc`)

[SOM Neighbor Distances](#) (`plotsomnd`)

[SOM Input Planes](#) (`plotsomplanes`)

[SOM Sample Hits](#) (`plotsomhits`)

[SOM Weight Positions](#) (`plotsompos`)

Plot Interval:  1 epochs



Maximum epoch reached.



Stop Training



Cancel

Here the self-organizing map is used to compute the class vectors of each of the training inputs. These classifications cover the feature space populated by the known flowers, and can now be used to classify new flowers accordingly. The network output will be a  $64 \times 150$  matrix, where each  $i$ th column represents the  $j$ th cluster for each  $i$ th input vector with a 1 in its  $j$ th element.

The function **vec2ind** returns the index of the neuron with an output of 1, for each vector. The indices will range between 1 and 64 for the 64 clusters represented by the 64 neurons.

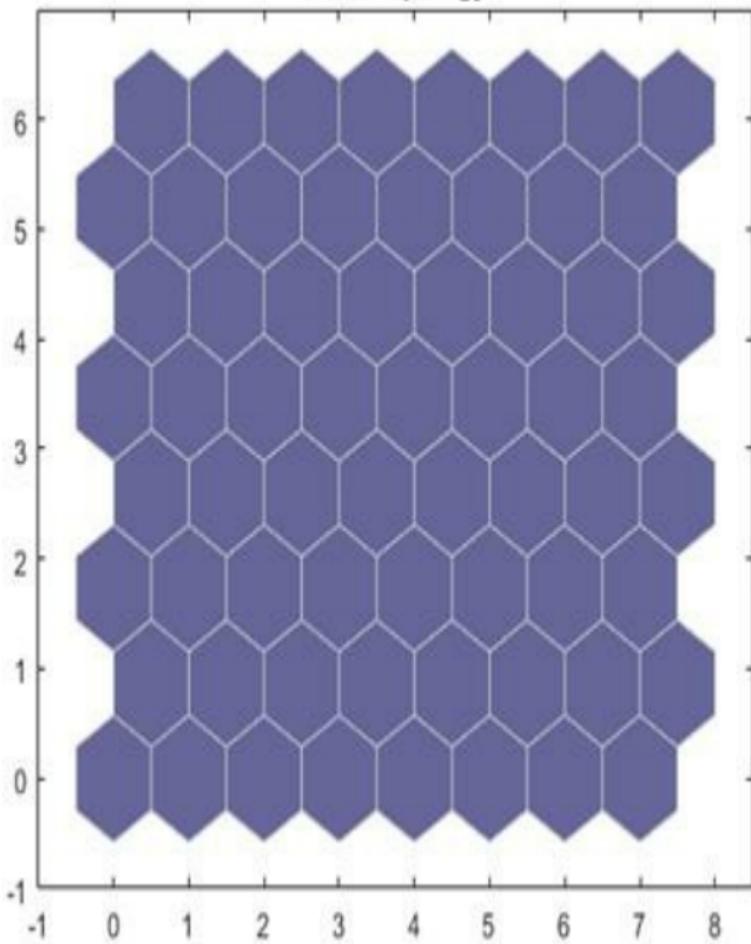
```
y = net(x);
```

```
cluster_index =  
vec2ind(y);
```

**plotsomtop** plots the self-organizing maps topology of 64 neurons positioned in an 8x8 hexagonal grid. Each neuron has learned to represent a different class of flower, with adjacent neurons typically representing similar classes.

```
plotsomtop(net)
```

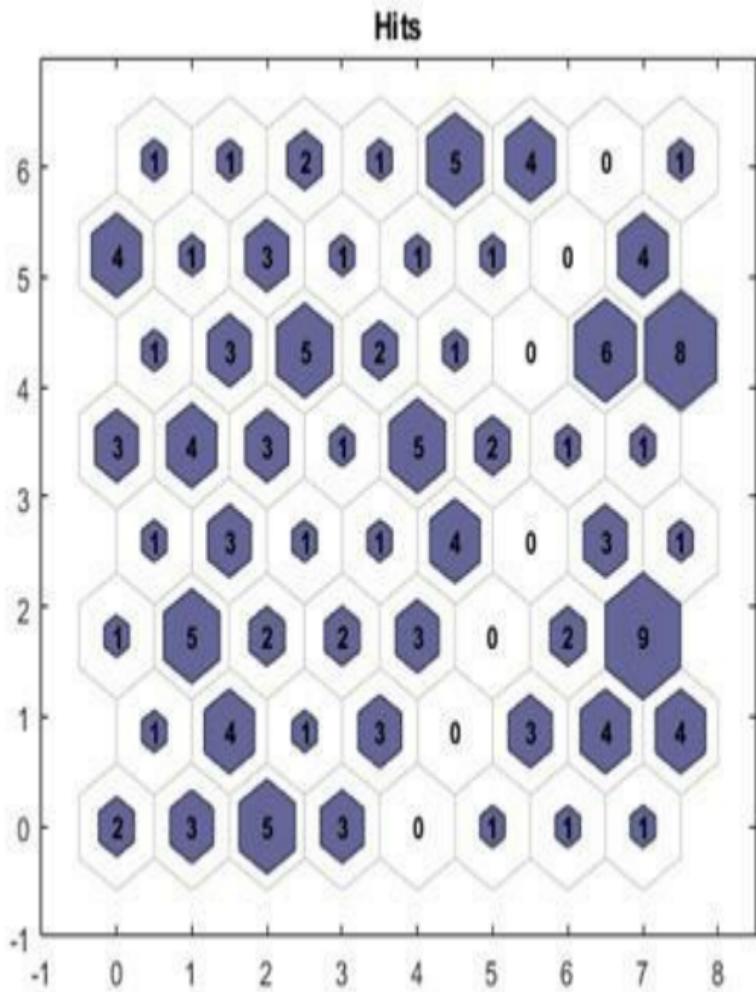
SOM Topology



**plotsomhits** calculates the classes for each flower and shows the number of

flowers in each class. Areas of neurons with large numbers of hits indicate classes representing similar highly populated regions of the feature space. Whereas areas with few hits indicate sparsely populated regions of the feature space.

plotsomhits(net, x)

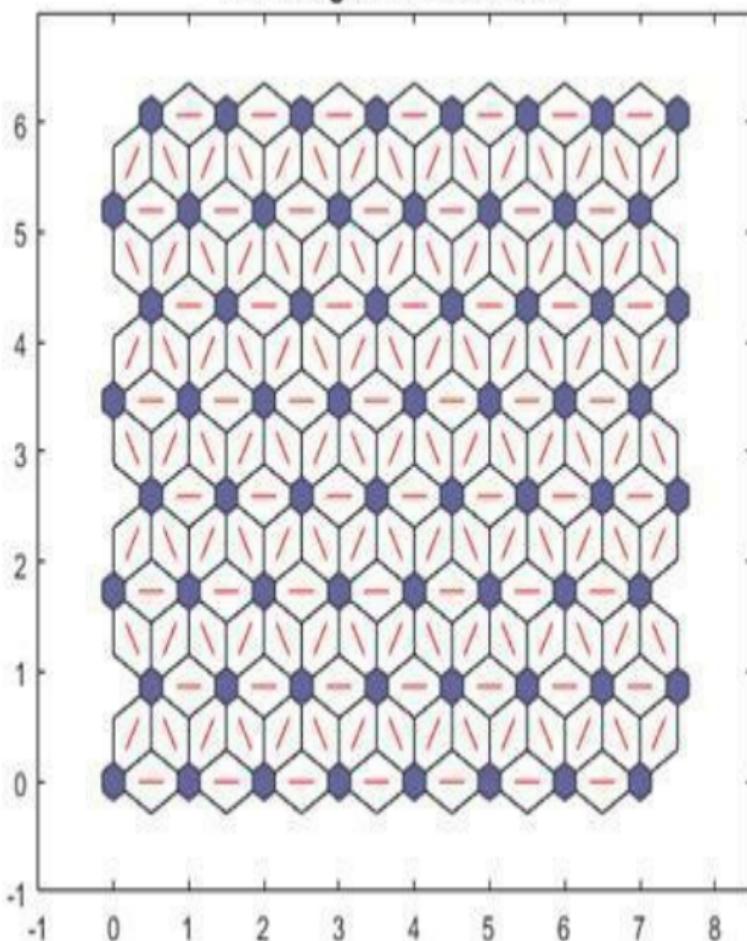


**plotsomnc** shows the neuron neighbor connections. Neighbors typically

classify similar samples.

`plotsomnc(net)`

### SOM Neighbor Connections

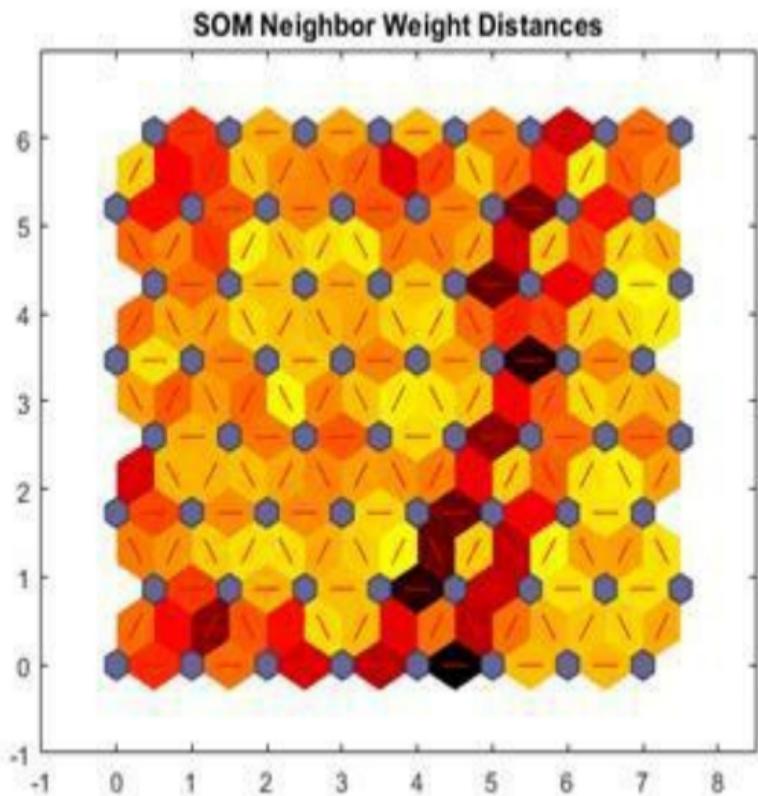


**plotsomnd** shows how distant (in terms of Euclidian distance) each neuron's

class is from its neighbors. Connections which are bright indicate highly connected areas of the input space. While dark connections indicate classes representing regions of the feature space which are far apart, with few or no flowers between them.

Long borders of dark connections separating large regions of the input space indicate that the classes on either side of the border represent flowers with very different features.

plotsomnd(net)

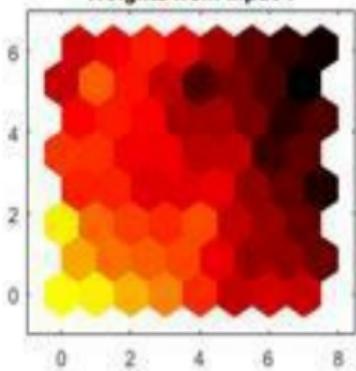


**plotsomplanes** shows a weight plane for each of the four input features. They are visualizations of the weights that connect each input to each of the 64 neurons in the 8x8 hexagonal grid. Darker colors

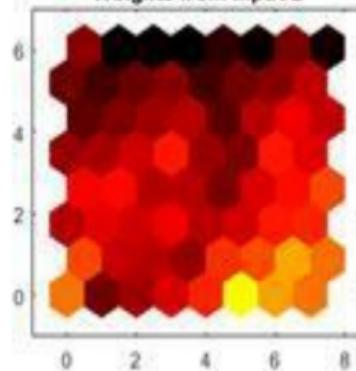
represent larger weights. If two inputs have similar weight planes (their color gradients may be the same or in reverse) it indicates they are highly correlated.

`plotsomplanes (net)`

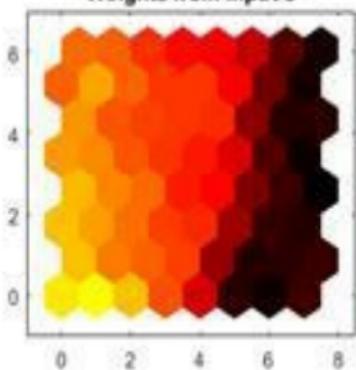
Weights from Input 1



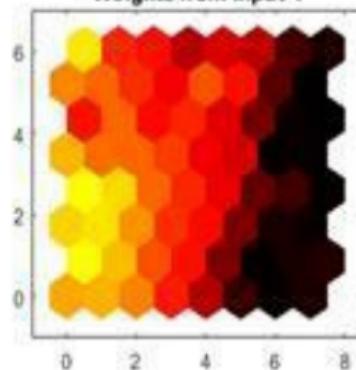
Weights from Input 2



Weights from Input 3



Weights from Input 4



This example illustrated how to design a

neural network that clusters iris flowers based on four of their characteristics.

## 15.4 GENE EXPRESSION ANALYSIS

This example demonstrates looking for patterns in gene expression profiles in baker's yeast using neural networks.

# 15.4.1 The Problem: Analyzing Gene Expressions in Baker's Yeast (*Saccharomyces cerevisiae*)

The goal is to gain some understanding of gene expressions in *Saccharomyces cerevisiae*, which is commonly known as baker's yeast or brewer's yeast. It is the fungus that is used to bake bread and ferment wine from grapes.

*Saccharomyces cerevisiae*, when introduced in a medium rich in glucose, can convert glucose to ethanol. Initially, yeast converts glucose to ethanol by a

metabolic process called "fermentation". However once supply of glucose is exhausted yeast shifts from anaerobic fermentation of glucose to aerobic respiraton of ethanol. This process is called diauxic shift. This process is of considerable interest since it is accompanied by major changes in gene expression.

The example uses DNA microarray data to study temporal gene expression of almost all genes in *Saccharomyces cerevisiae* during the diauxic shift.

You need Bioinformatics Toolbox™ to run this example.

if ~nnDependency.bioInfoAvailable

```
errordlg('This example requires  
Bioinformatics Toolbox.');
```

return;

```
end
```

## 15.4.2 The Data

This example uses data from DeRisi, JL, Iyer, VR, Brown, PO. "Exploring the metabolic and genetic control of gene expression on a genomic scale." Science. 1997 Oct 24;278(5338):680-6. PMID: 9381177

The full data set can be downloaded from the Gene Expression Omnibus website: <http://www.yeastgenome.org>

Start by loading the data into MATLAB®.

```
load yeastdata.mat
```

Gene expression levels were measured

at seven time points during the diauxic shift. The variable times contains the times at which the expression levels were measured in the experiment. The variable genes contains the names of the genes whose expression levels were measured.

The

variable yeastvalues contains the "VALUE" data or LOG\_RAT2N\_MEAN, or log2 of ratio of CH2DN\_MEAN and CH1DN\_MEAN from the seven time steps in the experiment.

To get an idea of the size of the data you can use **numel(genes)** to show how many genes there are in the data set.

**numel(genes)**

ans =

6400

genes is a cell array of the gene names.  
You can access the entries using  
MATLAB cell array indexing:

genes{15}

ans =

YAL054C

This indicates that the 15th row of the  
variable **yeastvalues** contains  
expression levels for the  
ORF YAL054C. You can use the web

command to access information about this ORF in the *Saccharomyces* Genome Database (SGD).

```
url = sprintf(...  
    'http://www.yeastgenome.org/cgi-  
bin/locus.fpl?locus=%s',...  
    genes{15});  
web(url);
```

## 15.4.3 Filtering the Genes

The data set is quite large and a lot of the information corresponds to genes that do not show any interesting changes during the experiment. To make it easier to find the interesting genes, the first thing to do is to reduce the size of the data set by removing genes with expression profiles that do not show anything of interest. There are 6400 expression profiles. You can use a number of techniques to reduce this to some subset that contains the most significant genes.

If you look through the gene list you will see several spots marked as 'EMPTY'.

These are empty spots on the array, and while they might have data associated with them, for the purposes of this example, you can consider these points to be noise. These points can be found using the **strcmp** function and removed from the data set with indexing commands.

```
emptySpots = strcmp('EMPTY',genes);  
yeastvalues(emptySpots,:)=[];  
genes(emptySpots) = [];  
numel(genes)
```

ans =

6314

In the yeastvalues data you will also see several places where the expression level is marked as NaN. This indicates that no data was collected for this spot at the particular time step. One approach to dealing with these missing values would be to impute them using the mean or median of data for the particular gene over time. This example uses a less rigorous approach of simply throwing away the data for any genes where one or more expression level was not measured.

The function **isnan** is used to identify the genes with missing data and indexing commands are used to remove the genes with missing data.

```
nanIndices = any(isnan(yeastvalues),2);  
yeastvalues(nanIndices,:) = [];  
genes(nanIndices) = [];  
numel(genes)
```

ans =

6276

If you were to plot the expression profiles of all the remaining profiles, you would see that most profiles are flat and not significantly different from the others. This flat data is obviously of use as it indicates that the genes associated with these profiles are not significantly affected by the diauxic shift; however, in this example, you are interested in the

genes with large changes in expression accompanying the diauxic shift. You can use filtering functions in the Bioinformatics Toolbox™ to remove genes with various types of profiles that do not provide useful information about genes affected by the metabolic change.

You can use the **genevarfilter** function to filter out genes with small variance over time. The function returns a logical array of the same size as the variable genes with ones corresponding to rows of yeastvalues with variance greater than the 10th percentile and zeros corresponding to those below the threshold.

```
mask = genevarfilter(yeastvalues);
```

```
% Use the mask as an index into the  
values to remove the filtered genes.  
yeastvalues = yeastvalues(mask,:);  
genes = genes(mask);  
numel(genes)
```

ans =

5648

The function **genelowvalfilter** removes genes that have very low absolute expression values. Note that the gene filter functions can also automatically calculate the filtered data and names.

[mask, yeastvalues, genes] = ...

```
genelowvalfilter(yeastvalues,genes,'absv  
numel(genes)
```

ans =

822

Use **geneentropyfilter** to remove genes whose profiles have low entropy:

```
[mask, yeastvalues, genes] = ...
```

```
geneentropyfilter(yeastvalues,genes,'prct'  
numel(genes)
```

ans =

614

## 15.4.4 Principal Component Analysis

Now that you have a manageable list of genes, you can look for relationships between the profiles.

Normalizing the standard deviation and mean of data allows the network to treat each input as equally important over its range of values.

Principal-component analysis (PCA) is a useful technique that can be used to reduce the dimensionality of large data sets, such as those from microarray analysis. This technique isolates the principal components of the dataset

eliminating those components that contribute the least to the variation in the data set.

The two settings variables can be used to apply **mapstd** and **processpca** to other data to consistently when the network is applied to new data.

```
[x,std_settings] = mapstd(yeastvalues');  
% Normalize data  
[x,pca_settings] = processpca(x,0.15);  
% PCA
```

The input vectors are first normalized, using **mapstd**, so that they have zero mean and unity variance. **processpca** is the function that implements the PCA algorithm. The second argument passed

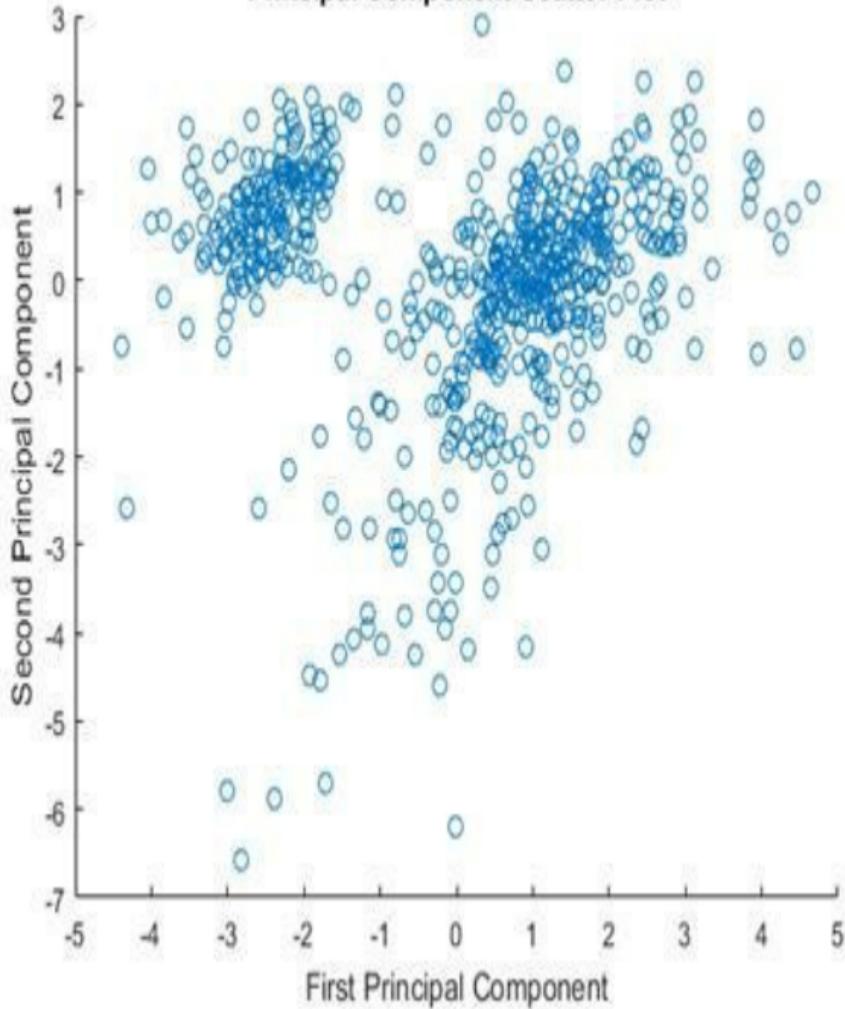
to processpca is 0.15. This means that processpca eliminates those principal components that contribute less than 15% to the total variation in the data set. The variable pc now contains the principal components of the yeastvalues data.

The principal components can be visualized using the **scatter** function.

figure

```
scatter(x(1,:),x(2,:));  
xlabel('First Principal Component');  
ylabel('Second Principal Component');  
title('Principal Component Scatter Plot');
```

### Principal Component Scatter Plot



## 15.4.5 Cluster Analysis: Self-Organizing Maps

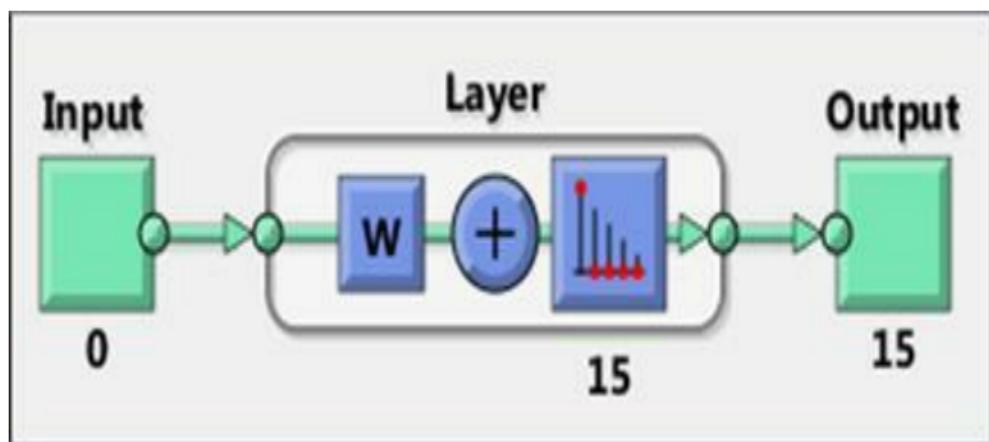
The principal components can be now be clustered using the Self-Organizing map (SOM) clustering algorithm available in Neural Network Toolbox software.

The **selforgmap** function creates a Self-Organizing map network which can then be trained with the **train** function.

The input size is 0 because the network has not yet been configured to match our input data. This will happen when the network is trained.

```
net = selforgmap([5 3]);
```

view(net)



Now the network is ready to be trained.

The NN Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

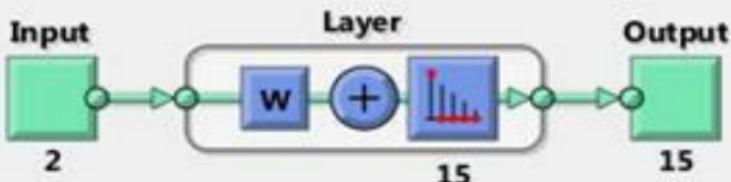
The buttons at the bottom open useful plots which can be opened during and

after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
net = train(net,x);
```

```
nntraintool
```

## Neural Network



### Algorithms

Training: Batch Weight/Bias Rules (trainbu)  
Performance: Mean Squared Error (mse)  
Calculations: MATLAB

### Progress

Epoch: 0 200 iterations 200  
Time: 0:00:00

### Plots

SOM Topology [\(plot somtop\)](#)

SOM Neighbor Connections [\(plot somnc\)](#)

SOM Neighbor Distances [\(plot somnd\)](#)

SOM Input Planes [\(plot somplanes\)](#)

SOM Sample Hits [\(plot somhits\)](#)

SOM Weight Positions [\(plot sompos\)](#)

Plot Interval:  1 epochs



Maximum epoch reached.



Stop Training

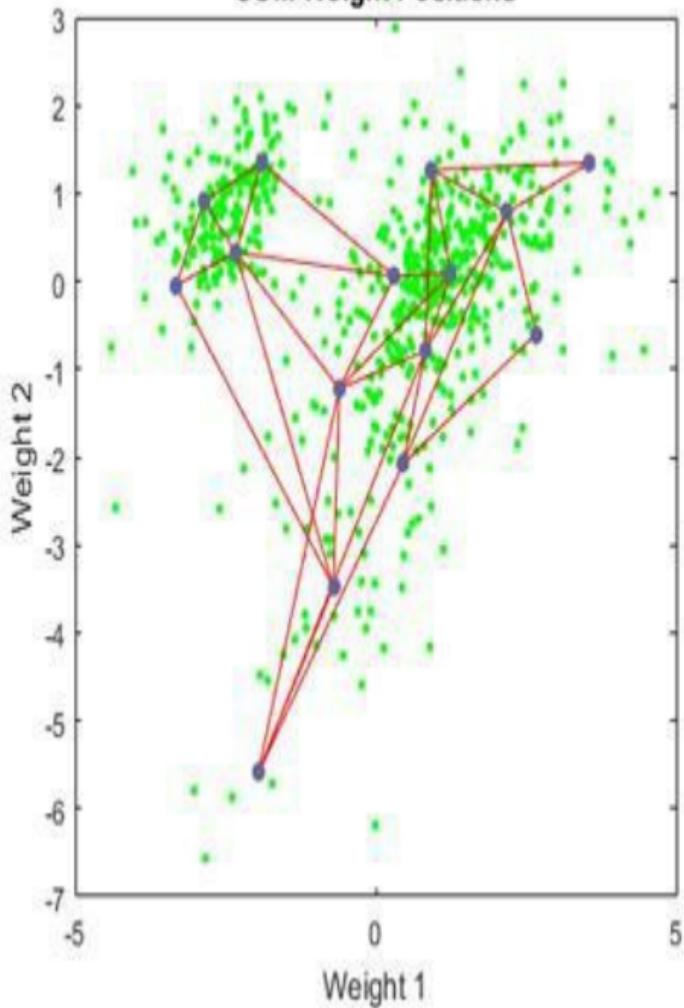


Cancel

Use **plotsompos** to display the network over a scatter plot of the first two dimensions of the data.

```
figure  
plotsompos(net,x);
```

### SOM Weight Positions



You can assign clusters using the SOM by finding the nearest node to each point

in the data set.

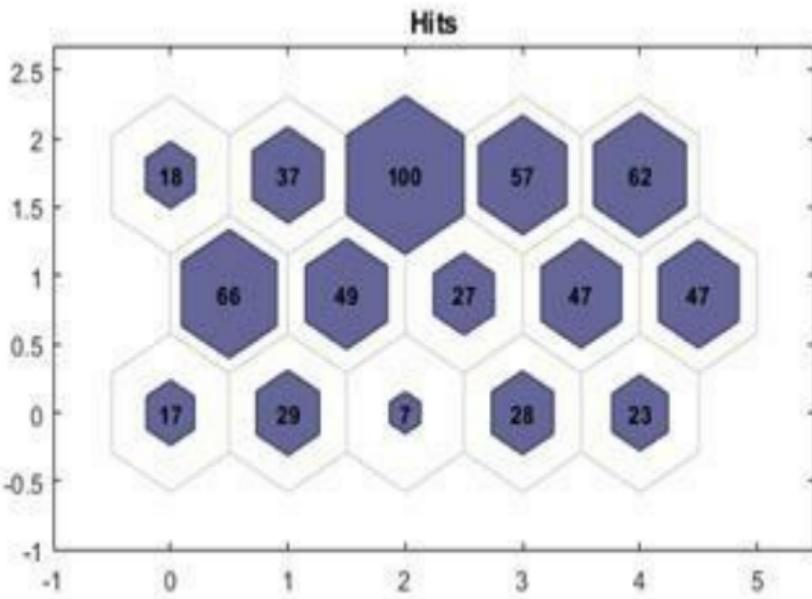
```
y = net(x);
```

```
cluster_indices = vec2ind(y);
```

Use **plotsomhits** to see how many vectors are assigned to each of the neurons in the map.

```
figure
```

```
plotsomhits(net,x);
```



You can also use other clustering algorithms like Hierarchical clustering and K-means, available in the Statistics and Machine Learning Toolbox™ for cluster analysis.



# **Chapter 16**

## **SELF-ORGANIZING NETWORKS. EXAMPLES**

---

# 16.1 COMPETITIVE LEARNING

Neurons in a competitive layer learn to represent different regions of the input space where input vectors occur.

P is a set of randomly generated but clustered test data points. Here the data points are plotted.

A competitive network will be used to classify these points into natural classes.

% Create inputs X.

bounds = [0 1; 0 1]; % Cluster centers to be in these bounds.

clusters = 8; % This many clusters.  
points = 10; % Number of points

in each cluster.

std\_dev = 0.05; % Standard deviation of each cluster.

x =

nngenc(bounds,clusters,points,std\_dev);

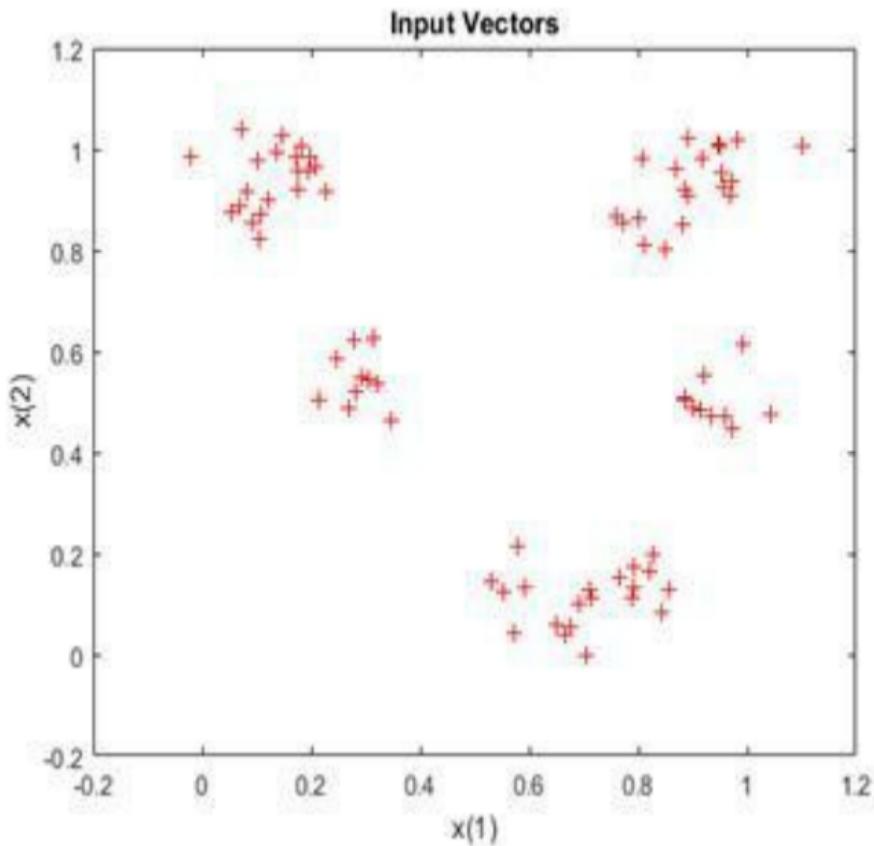
% Plot inputs X.

plot(x(1,:),x(2,:),'+r');

title('Input Vectors');

xlabel('x(1)');

ylabel('x(2)');



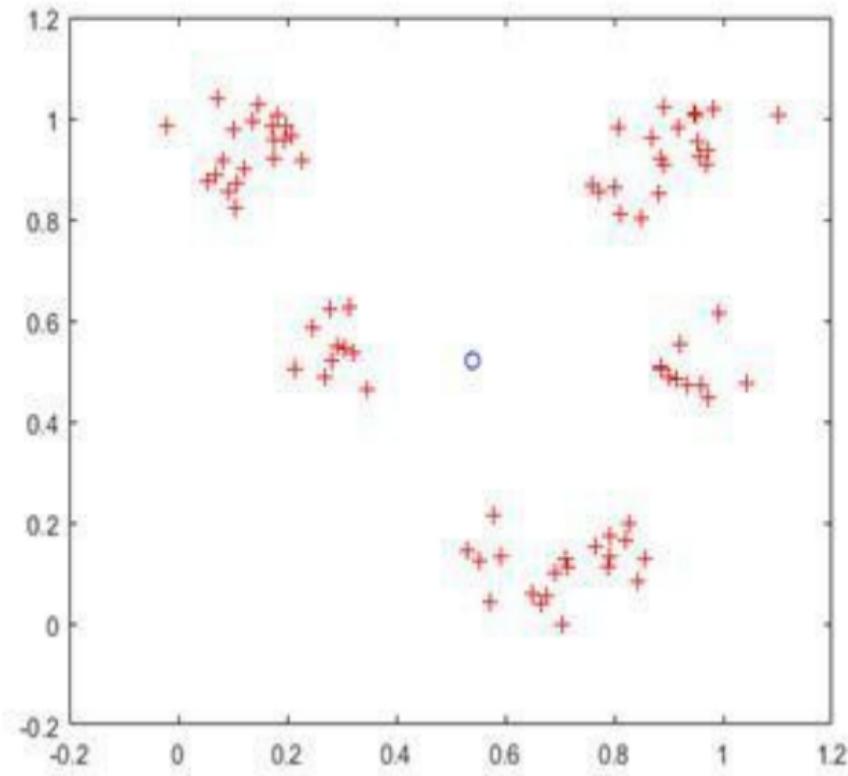
Here COMPETLAYER takes two arguments, the number of neurons and the learning rate.

We can configure the network inputs (normally done automatically by

TRAIN) and plot the initial weight vectors to see their attempt at classification.

The weight vectors (o's) will be trained so that they occur centered in clusters of input vectors (+'s).

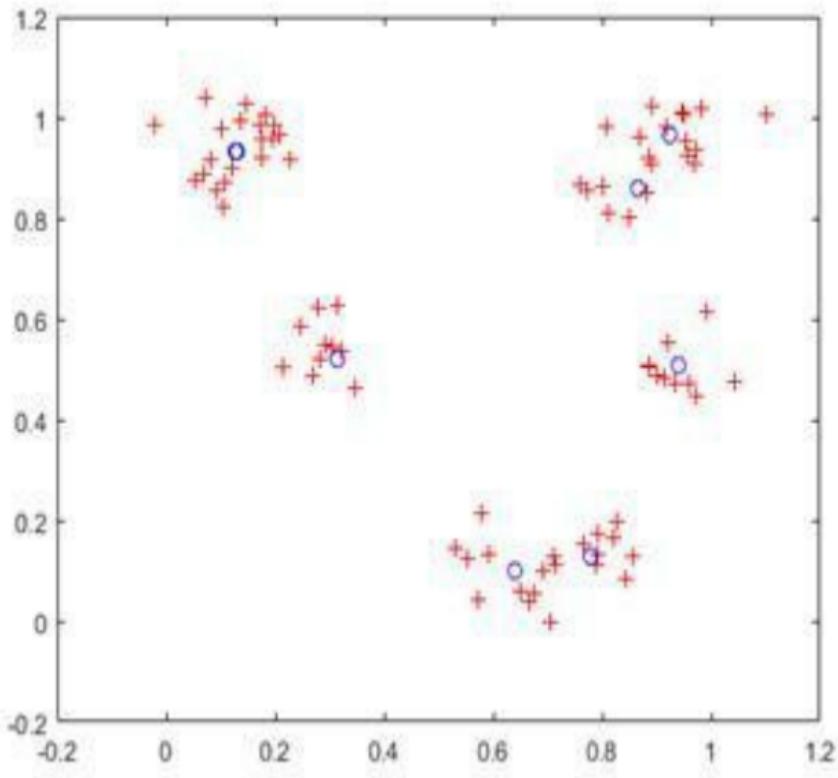
```
net = competlayer(8,.1);
net = configure(net,x);
w = net.IW{1};
plot(x(1,:),x(2,:),'+r');
hold on;
circles = plot(w(:,1),w(:,2),'ob');
```



Set the number of epochs to train before stopping and train this competitive layer (may take several seconds).

Plot the updated layer weights on the same graph.

```
net.trainParam.epochs = 7;  
net = train(net,x);  
w = net.IW{1};  
delete(circles);  
plot(w(:,1),w(:,2),'ob');
```



Now we can use the competitive layer as a classifier, where each neuron corresponds to a different category. Here we define an input vector  $X_1$  as  $[0; 0.2]$ .

The output  $Y$ , indicates which neuron is responding, and thereby which class the input belongs.

$$x_1 = [0; 0.2];$$

$$y = \text{net}(x_1)$$

$$y =$$

0

1

0

0

0

0

0

0

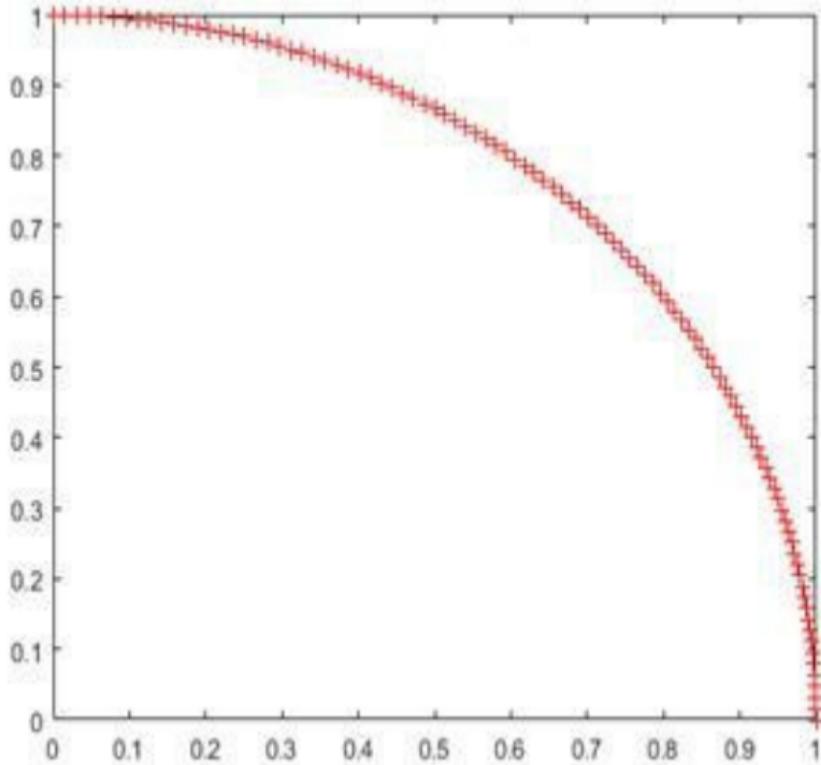
## 16.2 ONE-DIMENSIONAL SELF-ORGANIZING MAP

Neurons in a 2-D layer learn to represent different regions of the input space where input vectors occur. In addition, neighboring neurons learn to respond to similar inputs, thus the layer learns the topology of the presented input space.

Here 100 data points are created on the unit circle.

A competitive network will be used to classify these points into natural classes.

```
angles = 0:0.5*pi/99:0.5*pi;  
X = [sin(angles); cos(angles)];  
plot(X(1,:),X(2,:),'+r')
```



The map will be a 1-dimensional layer of 10 neurons.

```
net = selforgmap(10);
```

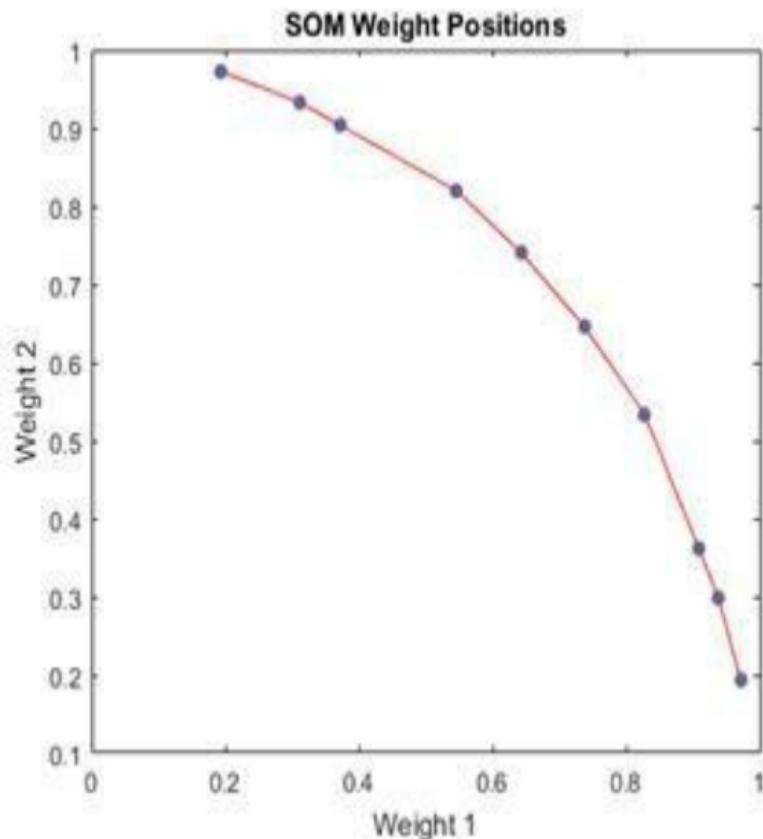
Specify the network is to be trained for 10 epochs and use TRAIN to train the network on the input data P:

```
net.trainParam.epochs = 10;  
net = train(net,X);
```

Now plot the trained network's weight positions with PLOTSOMPOS.

The red dots are the neuron's weight vectors, and the blue lines connect each pair within a distance of 1.

```
plotsompos(net)
```



The map can now be used to classify inputs, like  $[1; 0]$ :

Either neuron 1 or 10 should have an output of 1, as the above input vector was at one end of the presented input

space. The first pair of numbers indicate the neuron, and the single number indicates its output.

$$x = [1; 0];$$

$$a = \text{net}(x)$$

$$a =$$

1

0

0

0

0

0

0

0

0

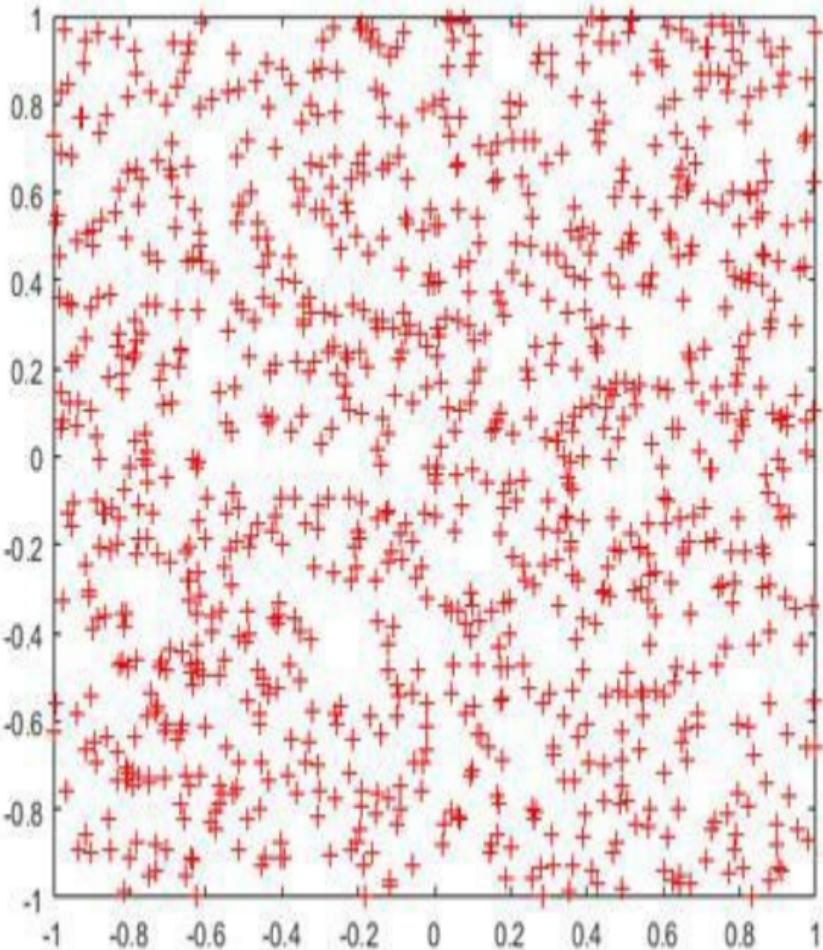
0

# 16.3 TWO-DIMENSIONAL SELF-ORGANIZING MAP

As in DEMOSM1, this self-organizing map will learn to represent different regions of the input space where input vectors occur. In this example, however, the neurons will arrange themselves in a two-dimensional grid, rather than a line.

We would like to classify 1000 two-element vectors occurring in a rectangular shaped vector space.

```
X = rands(2,1000);  
plot(X(1,:),X(2,:),'r')
```



We will use a 5 by 6 layer of neurons to classify the vectors above. We would

like each neuron to respond to a different region of the rectangle, and neighboring neurons to respond to adjacent regions.

The network is configured to match the dimensions of the inputs. This step is required here because we will plot the initial weights. Normally configuration is performed automatically by TRAIN.

```
net = selforgmap([5 6]);
```

```
net = configure(net,X);
```

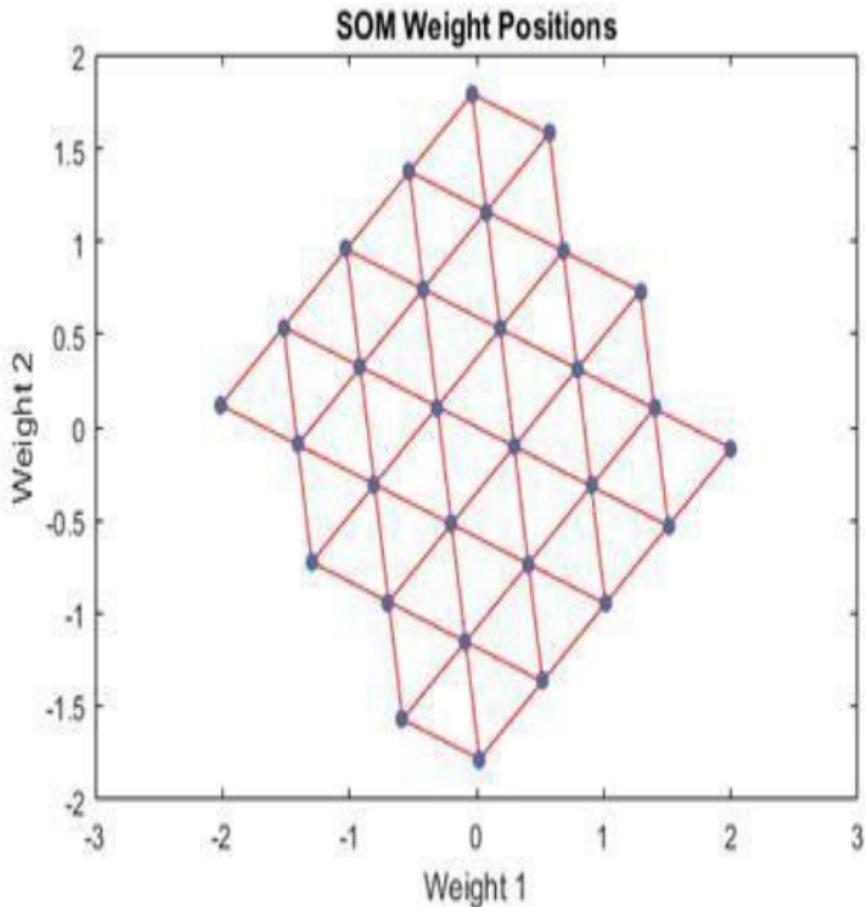
We can visualize the network we have just created with PLOTSOMPOS.

Each neuron is represented by a red dot at the location of its two weights.

Initially all the neurons have the same weights in the middle of the vectors, so

only one dot appears.

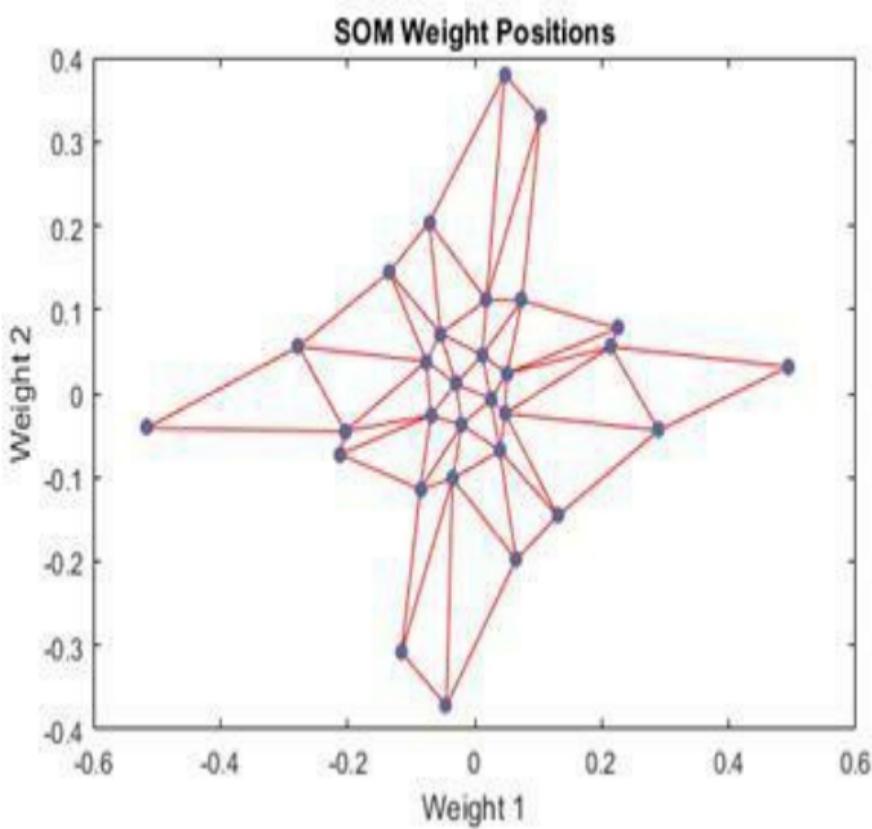
plotsompos(net)



Now we train the map on the 1000 vectors for 1 epoch and replot the network weights.

After training, note that the layer of neurons has begun to self-organize so that each neuron now classifies a different region of the input space, and adjacent (connected) neurons respond to adjacent regions.

```
net.trainParam.epochs = 1;  
net = train(net,X);  
plotsompos(net)
```



We can now use SIM to classify vectors by giving them to the network and seeing which neuron responds.

The neuron indicated by "a" responded with a "1", so  $x$  belongs to that class.

$$x = [0.5; 0.3];$$

$$y = \text{net}(x)$$

$$y =$$

1

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

# **Chapter 17**

# **BIBLIOGRAPHY**

---

# 17.1 NEURAL NETWORK 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*,

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.

[FIRe64] 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*, Dordrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan, M.T., and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999*

*American Control Conference*, San Diego, CA, 1999, pp. 1642–1656.

**[HaJe99]** Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311–340.

**[HaMe94]** Hagan, M.T., and M. Menhaj, "Training feed-forward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1999, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm

for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

**[HaRu78]** Harrison, D., and Rubinfeld, D.L., "Hedonic prices and the demand for clean air," *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81-102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

**[HDB96]** Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing,

1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has example programs, an instructor's guide, and transparency overheads for teaching.

**[HDH09]** Horn, J.M., O. De Jesús and M.T. Hagan, "Spurious Valleys in the Error Surface of Recurrent Networks - Analysis and Avoidance," IEEE Transactions on Neural Networks, Vol. 20, No. 4, pp. 686-700, April 2009.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also explains how training algorithms can be modified to avoid becoming stuck in these valleys.

[Hebb49] Hebb, D.O., *The Organization of Behavior*, New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

[Himm72] Himmelblau, D.M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

**[HuSb92]** Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, Neural Networks for Control System — A Survey," *Automatica*, Vol. 28, 1992, pp. 1083–1112.

**[JaRa04]** Jayadeva and S.A.Rahman, "A neural network with  $O(N)$  neurons for ranking  $N$  numbers in  $O(1/N)$  time," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 51, No. 10, 2004, pp. 2044–2051.

**[Joll86]** Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

**[KaGr96]** Kamwa, I., R. Grondin, V.K. Sood, C. Gagnon, Van Thich Nguyen,

and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *IEEE Transactions on Instrumentation and Measurement*, Vol. 45, No. 2, 1996, pp. 657–664.

[Koho87] Kohonen, T., *Self-Organization and Associative Memory*, 2nd Edition, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

[Koho97] Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications, and hardware of self-organizing maps. It also includes a comprehensive literature survey.

[LiMi89] Li, J., A.N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, 1989, pp. 1405–1422.

This paper discusses a class of neural networks described by first-order linear

differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li, et. al., is implemented in Advanced Topics in the *User's Guide*.

[Lipp87] Lippman, R.P., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, 1987, pp. 4–22.

This paper gives an introduction to the

field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

**[MacK92]** MacKay, D.J.C., "Bayesian interpolation," *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415–447.

**[Marq63]** Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431–441.

**[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.

**[NaPa91]** Narendra, Kumpati S. and Kannan Parthasarathy, "Learning Automata Approach to Hierarchical Multiobjective Analysis," *IEEE Transactions on Systems, Man and*

*Cybernetics*, Vol. 20, No. 1,  
January/February 1991, pp. 263–272.

[NgWi89] Nguyen, D., and B. Widrow,  
"The truck backer-upper: An example of  
self-learning in neural  
networks," *Proceedings of the  
International Joint Conference on  
Neural Networks*, Vol. 2, 1989, pp. 357–  
363.

This paper describes a two-layer  
network that first learned the truck  
dynamics and then learned how to back  
the truck to a specified position at a  
loading dock. To do this, the neural  
network had to solve a highly nonlinear  
control systems problem.

[NgWi90] Nguyen, D., and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 3, 1990, pp. 21–26.

Nguyen and Widrow show that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-

Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

**[Powe77]** Powell, M.J.D., "Restart procedures for the conjugate gradient method," *Mathematical Programming*, Vol. 12, 1977, pp. 241–254.

**[Pulu92]** Purdie, N., E.A. Lucas, and M.B. Talley, "Direct measure of total cholesterol and its distribution among major serum lipoproteins," *Clinical Chemistry*, Vol. 38, No. 9, 1992, pp. 1645–1647.

**[RiBr93]** Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP

algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

**[Robin94]** Robinson, A.J., "An application of recurrent nets to phone probability estimation," *IEEE Transactions on Neural Networks*, Vol. 5 , No. 2, 1994.

**[RoJa96]** Roman, J., and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, Vol. 2, 1996, pp. 454–460.

[Rose61] Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt's results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

[RuHi86a] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation," in D.E. Rumelhart and J.L. McClelland, Eds., *Parallel Data Processing, Vol. 1*, Cambridge, MA: The M.I.T. Press, 1986, pp. 318–362.

This is a basic reference on backpropagation.

[**RuHi86b**] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning representations by back-propagating errors," *Nature*, Vol. 323, 1986, pp. 533–536.

[**RuMc86**] Rumelhart, D.E., J.L. McClelland, and the PDP Research Group, Eds., *Parallel Distributed Processing, Vols. 1 and 2*, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of

publication.

[Scal85] Scales, L.E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

[SoHa96] Soloway, D., and P.J. Haley, "Neural Generalized Predictive Control," *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277–281.

[VoMa88] Vogl, T.P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, Vol. 59, 1988, pp. 256–264.

Backpropagation learning can be

speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

**[WaHa89]** Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, 1989, pp. 328–339.

**[Wass93]** Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.

**[WeGe94]** Weigend, A. S., and N. A. Gershenfeld, eds., *Time Series*

*Prediction: Forecasting the Future and Understanding the Past*, Reading, MA: Addison-Wesley, 1994.

[WiHo60] Widrow, B., and M.E. Hoff,  
"Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96–104.

[WiSt85] Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.

