

# NEURAL NETWORKS by examples using MATLAB

E. MARQUES



**NEURAL  
NETWORKS BY  
EXAMPLES  
USING MATLAB**



**F. MARQUES**

# CONTENTS

---

[NEURAL NETWORKS](#)

[1.1 INTRODUCTION](#)

## 1.2 TYPES OF ARTIFICIAL NEURAL NETWORKS

## 1.3 WORKING OF ANNS

## 1.4 MACHINE LEARNING IN ANNS

## 1.5 BACK PROPAGATION ALGORITHM

## 1.6 BAYESIAN NETWORKS (BN)

## 1.7 BUILDING A BAYESIAN NETWORK

## 1.8 APPLICATIONS OF NEURAL NETWORKS

# NEURAL NETWORKS WITH MATLAB

## 2.1 NEURAL NETWORK TOOLBOX

## 2.2 USING NEURAL NETWORK TOOLBOX

## 2.3 AUTOMATIC SCRIPT GENERATION

## 2.4 NEURAL NETWORK TOOLBOX APPLICATIONS

## 2.5 NEURAL NETWORK DESIGN STEPS

# FIT DATA WITH A NEURAL NETWORK

### 3.1 INTRODUCTION

### 3.2 USING THE NEURAL NETWORK FITTING TOOL

### 3.3 USING COMMAND-LINE FUNCTIONS

## CLASSIFY PATTERNS WITH A NEURAL NETWORK

### 4.1 INTRODUCTION

### 4.2 USING THE NEURAL NETWORK PATTERN RECOGNITION TOOL

### 4.3 USING COMMAND-LINE FUNCTIONS

## CLUSTER DATA WITH A SELF- ORGANIZING MAP

### 5.1 INTRODUCTION

### 5.2 USING THE NEURAL NETWORK CLUSTERING TOOL

### 5.3 USING COMMAND-LINE FUNCTIONS

## NEURAL NETWORK TIME-SERIES PREDICTION AND MODELING

## 6.1 INTRODUCTION

## 6.2 USING THE NEURAL NETWORK TIME SERIES TOOL

## 6.3 USING COMMAND-LINE FUNCTIONS

# PARALLEL COMPUTING ON CPUS AND GPUS

## 7.1 PARALLEL COMPUTING TOOLBOX

## 7.2 PARALLEL CPU WORKERS

## 7.3 GPU COMPUTING

## 7.4 MULTIPLE GPU/CPU COMPUTING

## 7.5 CLUSTER COMPUTING WITH MATLAB DISTRIBUTED COMPUTING SERVER

## 7.6 LOAD BALANCING, LARGE PROBLEMS, AND BEYOND. NEURAL NETWORKS WITH PARALLEL AND GPU COMPUTING

### 7.6.1 Modes of Parallelism

7.6.2 Distributed Computing

7.6.3 Single GPU Computing

7.6.4 Distributed GPU Computing

7.6.5 Deep Learning

7.6.6 Parallel Time Series

7.6.7 Parallel Availability, Fallbacks, and Feedback

## NEURAL NETWORK TOOLBOX SAMPLE DATA SETS AND GLOSARY

8.1 NEURAL NETWORK TOOLBOX SAMPLE DATA SETS

8.2 GLOSARY

## BIBLIOGRAPHY

## **Chapter 1**

# **NEURAL NETWORKS**

---

# 1.1 INTRODUCTION

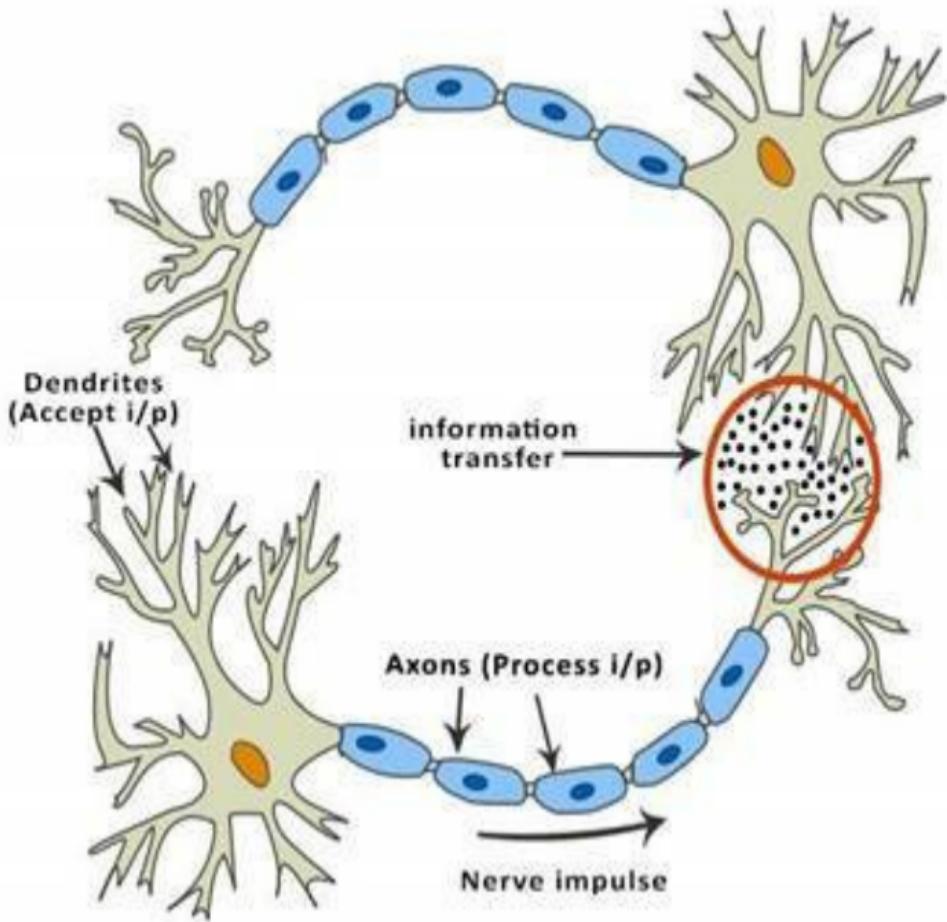
Neural networks theory is inspired from the natural neural network of human nervous system. Is possible define a neural network as a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.

The idea of ANNs is based on the belief that working of human brain by making the right connections, can be imitated using silicon and wires as living neurons and dendrites.

The human brain is composed of 100 billion nerve cells called neurons. They are connected to other thousand cells by Axons. Stimuli from external environment or inputs from sensory organs are accepted by dendrites. These inputs create electric impulses, which quickly travel through the neural network. A neuron can then send the message to other neuron to handle the issue or does not send it forward.

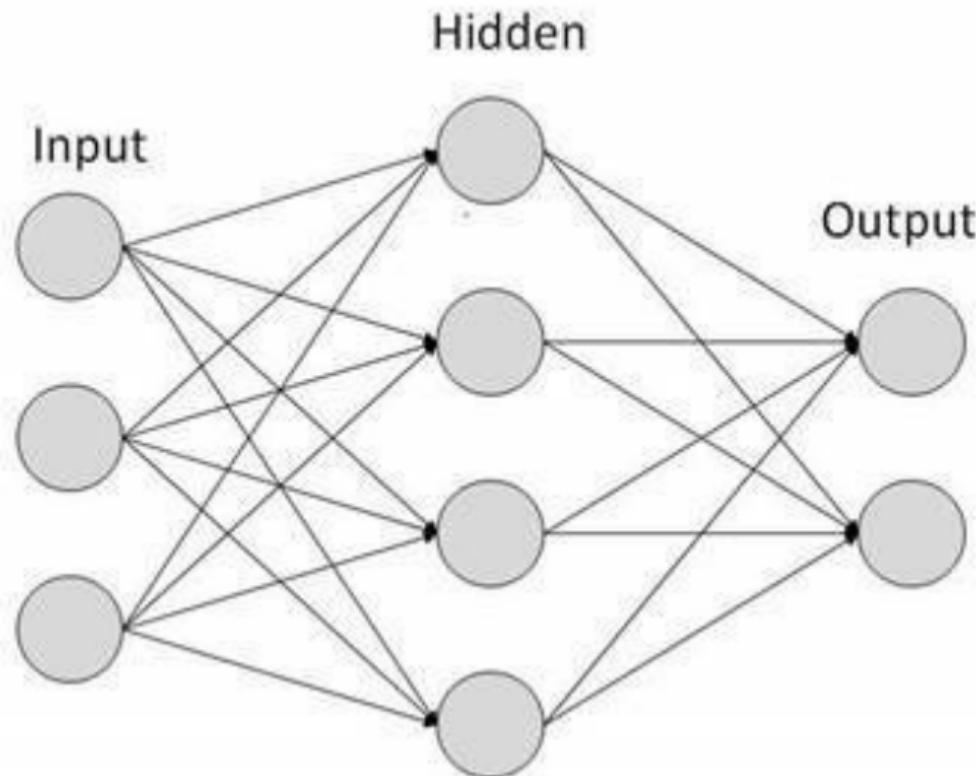
ANNs are composed of multiple nodes, which imitate biological neurons of human brain. The neurons are connected by links and they interact with each other. The nodes can take input data

and perform simple operations on the data. The result of these operations is passed to other neurons. The output at each node is called its activation or node value.



Each link is associated with weight. ANNs are capable of learning, which takes place by altering weight values. The following illustration

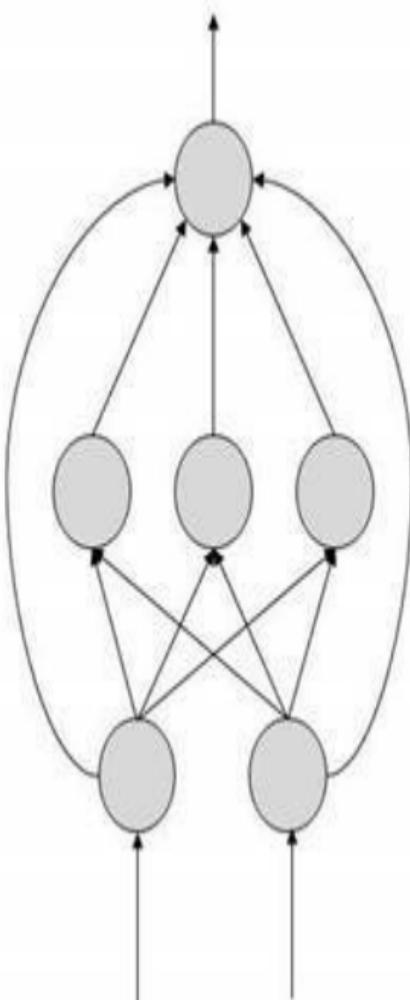
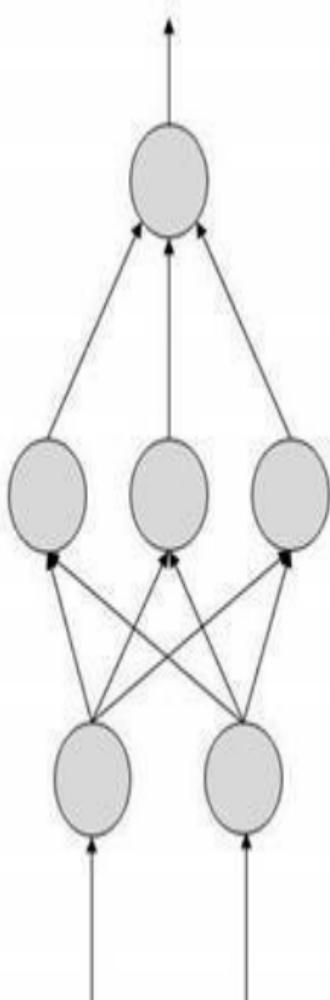
shows a simple ANN



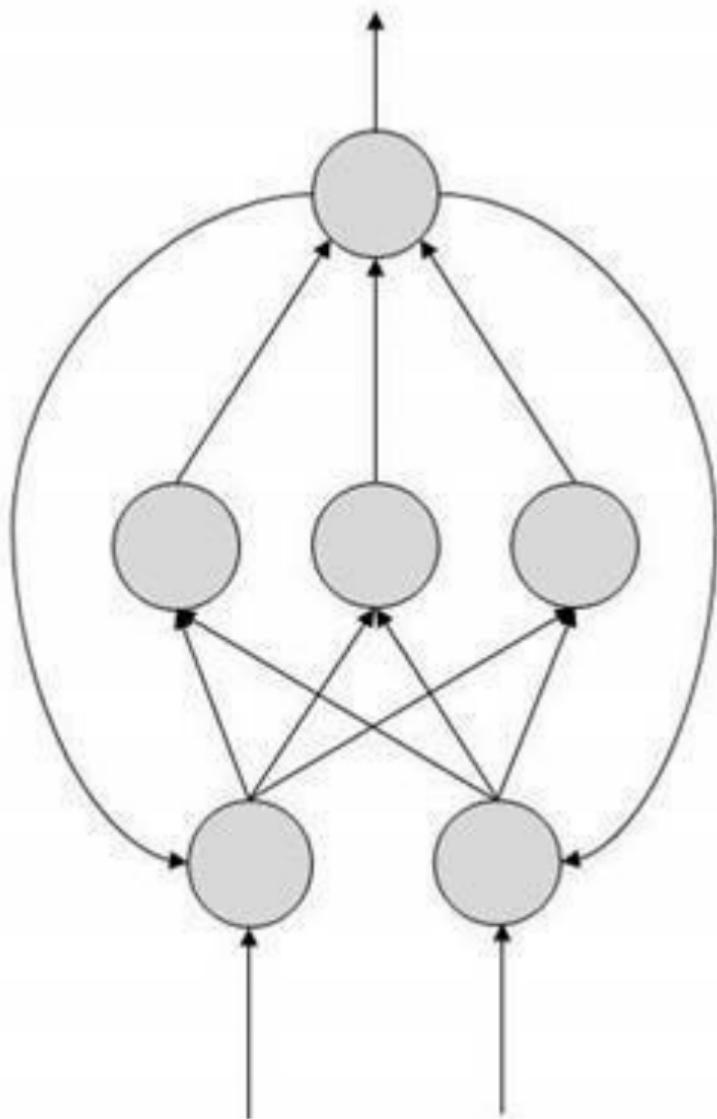
# 1.2 TYPES OF ARTIFICIAL NEURAL NETWORKS

There are two Artificial Neural Network topologies:  
FreeForward and Feedback.

*FeedForward ANN:* The information flow is unidirectional. A unit sends information to other unit from which it does not receive any information. There are no feedback loops. They are used in pattern generation/recognition/classification. They have fixed inputs and outputs.



*FeedBack ANN:* Here, feedback loops are allowed. They are used in content addressable memories.





# 1.3 WORKING OF ANNS

In the topology diagrams shown, each arrow represents a connection between two neurons and indicates the pathway for the flow of information. Each connection has a weight, an integer number that controls the signal between the two neurons.

If the network generates a “good or desired” output, there is no need to adjust the weights. However, if the network generates a “poor or undesired” output or an error, then the system alters the weights in order to improve subsequent results.

# 1.4 MACHINE LEARNING IN ANNS

ANNs are capable of learning and they need to be trained. There are several learning strategies:

- **Supervised Learning** – It involves a teacher that is scholar than the ANN itself. For example, the teacher feeds some example data about which the teacher already knows the answers. For example, pattern recognizing. The ANN comes up with guesses while recognizing. Then the teacher provides the ANN with

the answers. The network then compares it guesses with the teacher's "correct" answers and makes adjustments according to errors.

- **Unsupervised Learning** – It is required when there is no example data set with known answers. For example, searching for a hidden pattern. In this case, clustering i.e. dividing a set of elements into groups according to some unknown pattern is carried out based on the existing data sets present.

- **Reinforcement Learning** – This

strategy built on observation. The ANN makes a decision by observing its environment. If the observation is negative, the network adjusts its weights to be able to make a different required decision the next time.

# **1.5 BACK PROPAGATION ALGORITHM**

It is the training or learning algorithm. It learns by example. If you submit to the algorithm the example of what you want the network to do, it changes the network's weights so that it can produce desired output for a particular input on finishing the training.

Back Propagation networks are ideal for simple Pattern Recognition and Mapping Tasks.

# 1.6 BAYESIAN NETWORKS (BN)

These are the graphical structures used to represent the probabilistic relationship among a set of random variables. Bayesian networks are also called Belief Networks or Bayes Nets. BNs reason about uncertain domain. In these networks, each node represents a random variable with specific propositions. For example, in a medical diagnosis domain, the node Cancer represents the proposition that a patient has cancer.

The edges connecting the nodes represent probabilistic dependencies

among those random variables. If out of two nodes, one is affecting the other then they must be directly connected in the directions of the effect. The strength of the relationship between variables is quantified by the probability associated with each node.

There is an only constraint on the arcs in a BN that you cannot return to a node simply by following directed arcs. Hence the BNs are called Directed Acyclic Graphs (DAGs). BNs are capable of handling multivalued variables simultaneously. The BN variables are composed of two dimensions:

## Range of prepositions

Probability assigned to each of the prepositions.

Consider a finite set  $X = \{X_1, X_2, \dots, X_n\}$  of discrete random variables, where each variable  $X_i$  may take values from a finite set, denoted by  $\text{Val}(X_i)$ . If there is a directed link from variable  $X_i$  to variable,  $X_j$ , then variable  $X_i$  will be a parent of variable  $X_j$  showing direct dependencies between the variables.

The structure of BN is ideal for combining prior knowledge and observed data. BN can be used to learn

the causal relationships and understand various problem domains and to predict future events, even in case of missing data.

# 1.7 BUILDING A BAYESIAN NETWORK

A knowledge engineer can build a Bayesian network. There are a number of steps the knowledge engineer needs to take while building it.

**Example problem – *Lung cancer*.** A patient has been suffering from breathlessness. He visits the doctor, suspecting he has lung cancer. The doctor knows that barring lung cancer, there are various other possible diseases the patient might have such as tuberculosis and bronchitis.

## **Gather Relevant Information of Problem**

- Is the patient a smoker? If yes, then high chances of cancer and bronchitis.
- Is the patient exposed to air pollution? If yes, what sort of air pollution?
- Take an X-Ray positive X-ray would indicate either TB or lung cancer.

## **Identify Interesting Variables**

The knowledge engineer tries to

answer the questions :

- Which nodes to represent?
- What values can they take? In which state can they be?

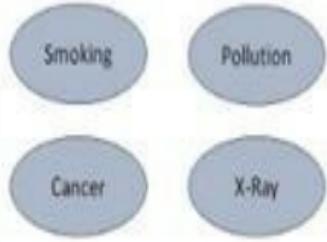
For now let us consider nodes, with only discrete values. The variable must take on exactly one of these values at a time.

**Common types of discrete nodes are:**

- **Boolean nodes** – They represent propositions, taking binary values TRUE (T) and FALSE (F).

- **Ordered values** – A node *Pollution* might represent and take values from {low, medium, high} describing degree of a patient's exposure to pollution.
- **Integral values** – A node called *Age* might represent patient's age with possible values from 1 to 120. Even at this early stage, modeling choices are being made.

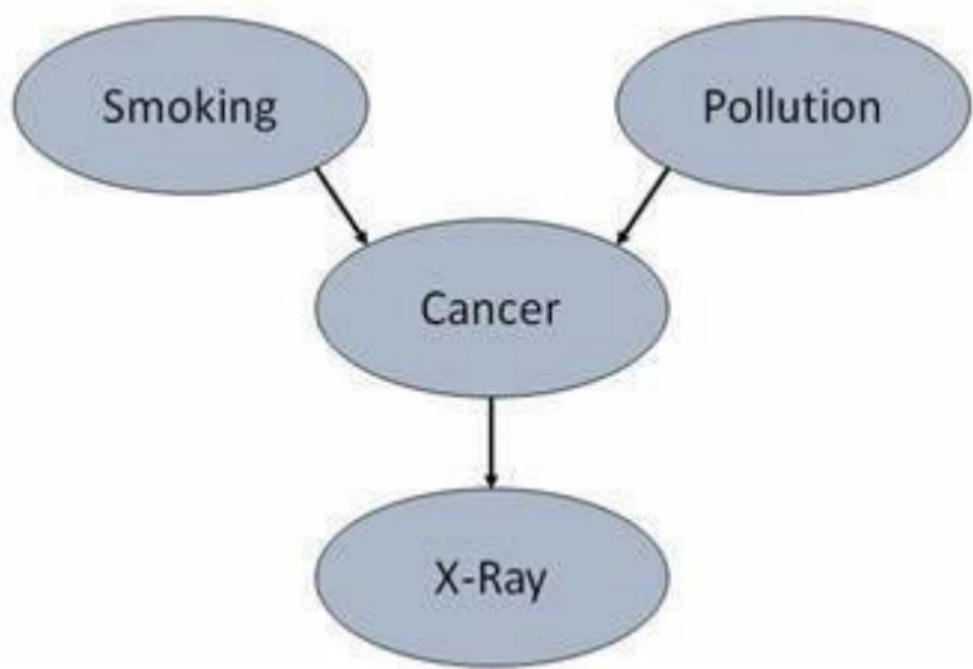
Possible nodes and values for the lung cancer example:

Node Name	Type	Value	Nodes Creation
Polution	Binary	{LOW, HIGH, MEDIUM}	
Smoker	Boolean	{TRUE, FALSE}	
Lung-Cancer	Boolean	{TRUE, FALSE}	
X-Ray	Binary	{Positive, Negative}	

## Create Arcs between Nodes

Topology of the network should capture qualitative relationships between variables. For example, what causes a patient to have lung cancer? -

Pollution and smoking. Then add arcs from node *Pollution* and node *Smoker* to node *Lung-Cancer*. Similarly if patient has lung cancer, then X-ray result will be positive. Then add arcs from node *Lung-Cancer* to node *X-Ray*.



**Specify Topology**

Conventionally, BNs are laid out so that the arcs point from top to bottom. The set of parent nodes of a node X is given by  $\text{Parents}(X)$ .

The *Lung-Cancer* node has two parents (reasons or causes): *Pollution* and *Smoker*, while node *Smoker* is an **ancestor** of node *X-Ray*. Similarly, *X-Ray* is a child (consequence or effects) of node *Lung-Cancer* and **successor** of nodes *Smoker* and *Pollution*.

## Conditional Probabilities

Now quantify the relationships

between connected nodes: this is done by specifying a conditional probability distribution for each node. As only discrete variables are considered here, this takes the form of a **Conditional Probability Table (CPT)**.

First, for each node we need to look at all the possible combinations of values of those parent nodes. Each such combination is called an **instantiation** of the parent set. For each distinct instantiation of parent node values, we need to specify the probability that the child will take.

For example, the *Lung-Cancer* node's parents

are *Pollution* and *Smoking*. They take the possible values = { (H,T), ( H,F), (L,T), (L,F) }. The CPT specifies the probability of cancer for each of these cases as <0.05, 0.02, 0.03, 0.001> respectively.

Each node will have conditional probability associated as follows:

<b>Smoking</b>
$P(S = T)$
0.30

<b>Pollution</b>
$P(P = L)$
0.90

<b>Lung-Cancer</b>		
<b>P</b>	<b>S</b>	<b><math>P(C = T   P, S)</math></b>
H	T	0.05
H	F	0.02
L	T	0.03
L	F	0.001

<b>X-Ray</b>	
<b>C</b>	<b><math>X = (Pos   C)</math></b>
T	0.90
F	0.20

# 1.8 APPLICATIONS OF NEURAL NETWORKS

They can perform tasks that are easy for a human but difficult for a machine:

- **Aerospace** – Autopilot aircrafts, aircraft fault detection.
- **Automotive** – Automobile guidance systems.
- **Military** – Weapon orientation and steering, target tracking, object discrimination, facial

recognition, signal/image identification.

- **Electronics** – Code sequence prediction, IC chip layout, chip failure analysis, machine vision, voice synthesis.
- **Financial** – Real estate appraisal, loan advisor, mortgage screening, corporate bond rating, portfolio trading program, corporate financial analysis, currency value prediction, document readers, credit application evaluators.
- **Industrial** – Manufacturing process control, product design and analysis, quality

inspection systems, welding quality analysis, paper quality prediction, chemical product design analysis, dynamic modeling of chemical process systems, machine maintenance analysis, project bidding, planning, and management.

- **Medical** – Cancer cell analysis, EEG and ECG analysis, prosthetic design, transplant time optimizer.
- **Speech** – Speech recognition, speech classification, text to speech conversion.
- **Telecommunications** – Image and data compression,

automated information services, real-time spoken language translation.

- **Transportation** – Truck Brake system diagnosis, vehicle scheduling, routing systems.
- **Software** – Pattern Recognition in facial recognition, optical character recognition, etc.
- **Time Series Prediction** – ANNs are used to make predictions on stocks and natural calamities.
- **Signal Processing** – Neural

networks can be trained to process an audio signal and filter it appropriately in the hearing aids.

- **Control** – ANNs are often used to make steering decisions of physical vehicles.
- **Anomaly Detection** – As ANNs are expert at recognizing patterns, they can also be trained to generate an output when something unusual occurs that misfits the pattern.

## **Chapter 2**

# **NEURAL NETWORKS WITH MATLAB**

---

## 2.1 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

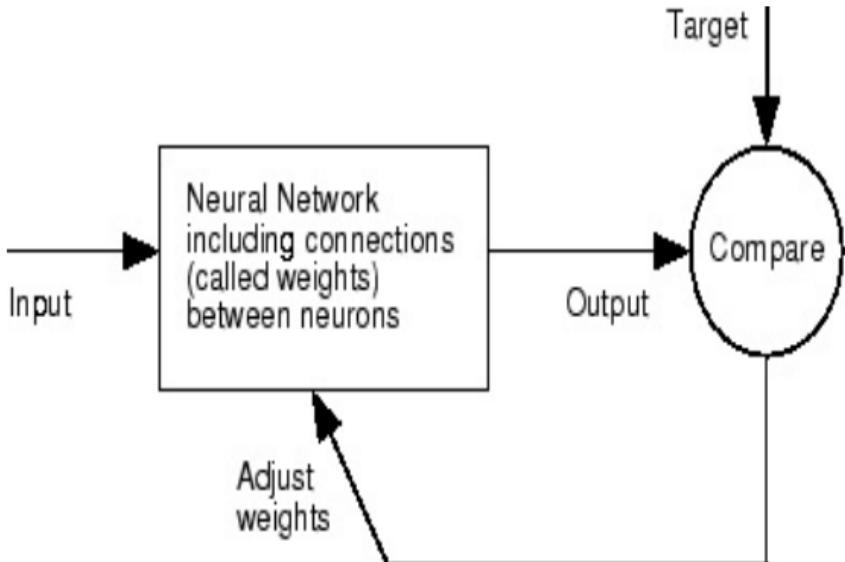
## (RNN)

- 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.

## 2.2 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.

## 2.3 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.

## **2.4 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	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
	Check and other

Banking

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
	Real estate appraisal, loan advising, mortgage screening,

Financial

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 process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, welding quality analysis, paper

## Manufacturing

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

Breast cancer cell analysis, EEG and

	analysis, ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, and emergency-room test advisement
Medical	
Oil and gas	Exploration
Robotics	Trajectory control, forklift robot, manipulator controllers, and vision systems

	market analysis,
Securities	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

## **2.5 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 3**

# **FIT DATA WITH A NEURAL NETWORK**

---

## 3.1 INTRODUCTION

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

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

You can solve this problem in two ways:

- Use a graphical user

interface, *nftool*, as described in [Using the Neural Network Fitting Tool](#).

• Use command-line functions, 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, first define the problem by selecting a data set. Each GUI has access to many sample data sets that you can use to experiment with the toolbox (see [Neural Network Toolbox Sample Data Sets](#)). If you have a specific problem that you want to solve, you can load your own

data into the workspace. The next section describes the data format.

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

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

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

The next section shows how to train a network to fit a data set, using the neural network fitting tool GUI, [\*nftool\*](#). This example uses the housing data set provided with the toolbox.

## 3.2 USING THE NEURAL NETWORK FITTING TOOL

- 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 *Fitting Tool* to open the Neural Network Fitting Tool. (You can also use the command [nftool](#).)



## Welcome to the Neural Fitting app.

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

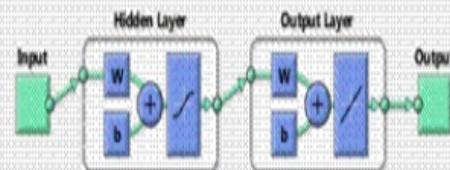
### Introduction

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

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

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

### Neural Network



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

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

To continue, click [Next].

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

- Click *Next* to proceed.



## Select Data

What inputs and targets define your fitting problem?

Get Data from Workspace

Input data to present to the network.

Inputs:

(none)

Target data defining desired network output.

Targets:

(none)

Samples are

Matrix columns  Matrix rows

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

Summary

No inputs selected.

No targets selected.

Select inputs and targets, then click [Next].

Welcome

Back

Next

Cancel

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

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



Select a data set:

Simple Fitting Problem

House Pricing

Abalone Rings

Body Fat

Building Energy

Chemical

Engine

Description

Filename: chemical dataset

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

This dataset can be used to train a neural network to estimate one sensor signal from eight other sensor signals.

LOAD chemical dataset.MAT loads these two variables:

chemicalInputs - a 8x498 matrix defining measurements taken from eight sensors during a chemical process.

chemicalTargets - a 1x498 matrix of a ninth sensor's measurements, to be estimated from the first eight.

A good estimator for the ninth sensor will allow it to be removed and estimations used in its place.



Import



Cancel

Select **Chemical**, and click **Import**.

This returns you to the Select Data window.

- Click **Next** to display the Validation and Test Data window, shown in the following figure.

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



## Validation and Test Data

Set aside some samples for validation and testing.

### Select Percentages

- Randomly divide up the 498 samples:

Training:	70%	348 samples
Validation:	15% ▾	75 samples
Testing:	15% ▾	75 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

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

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

Click **Next**.

The standard network that is used

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



## Network Architecture

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

### Hidden Layer

Define a fitting neural network... (fitnet)

Number of Hidden Neurons:

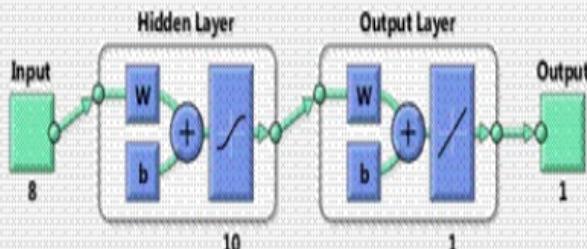
10

### Recommendation

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

[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 fit the inputs and targets.

### Train Network

Choose a training algorithm:

Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt... (trainlm)

Train

### Results

Samples

MSE

R

Training:

348

Validation:

75

Testing:

75

Plot Fit

Plot Error Histogram

Plot Regression

### Notes

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

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

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

Train network, then click [Next].

Neural Network Start

Welcome

Back

Next

Cancel

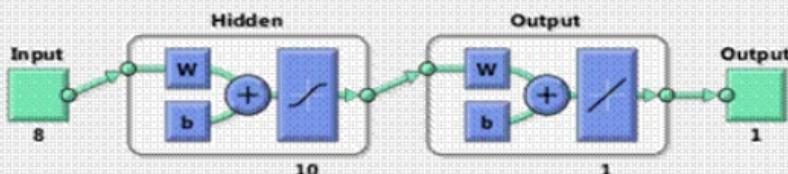
Select a training algorithm, then click **Train..**. Levenberg-Marquardt (`trainlm`) is recommended for most problems, but for some noisy and small problems Bayesian Regularization (`trainbr`) can take longer but obtain a better solution. For large problems, however, Scaled Conjugate Gradient (`trainscg`)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

The training continued until the validation error failed to decrease for

six iterations (validation stop).

# 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	19 iterations	1000
Time:		0:00:00	
Performance:	908	2.34	0.00
Gradient:	$3.07e+03$	9.55	$1.00e-07$
Mu:	0.00100	0.00100	$1.00e+10$
Validation Checks:	0	6	6

## Plots

Performance (plotperform)

Training State (plottrainstate)

Error Histogram (ploterrhist)

Regression (plotregression)

Fit (plotfit)

Plot Interval:



Validation stop.



Stop Training



Cancel

Under **Plots**, click **Regression**.

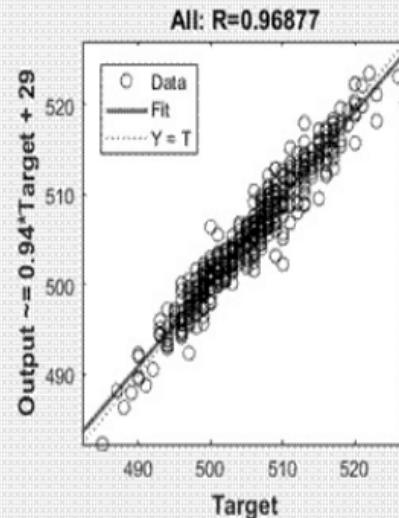
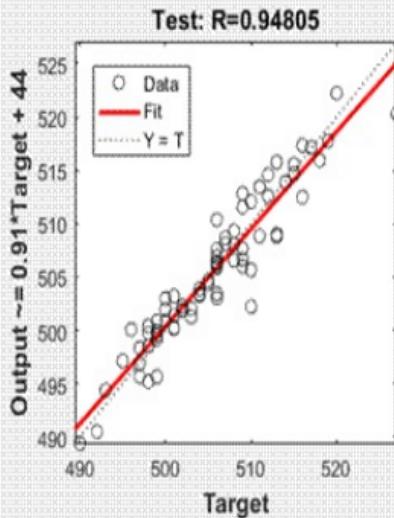
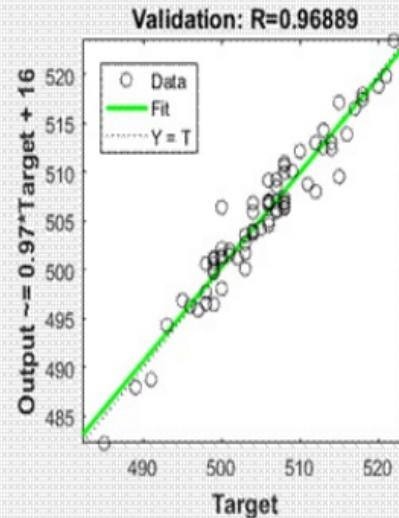
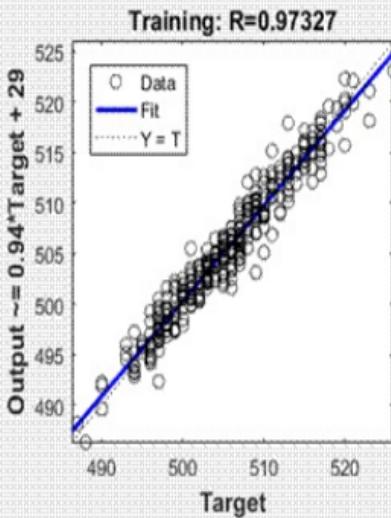
This is used to validate the network performance.

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

weights and biases of the network, and may produce an improved network after retraining. Other options are provided on the following pane.

# Regression (plotregression)

File Edit View Insert Tools Desktop Window Help

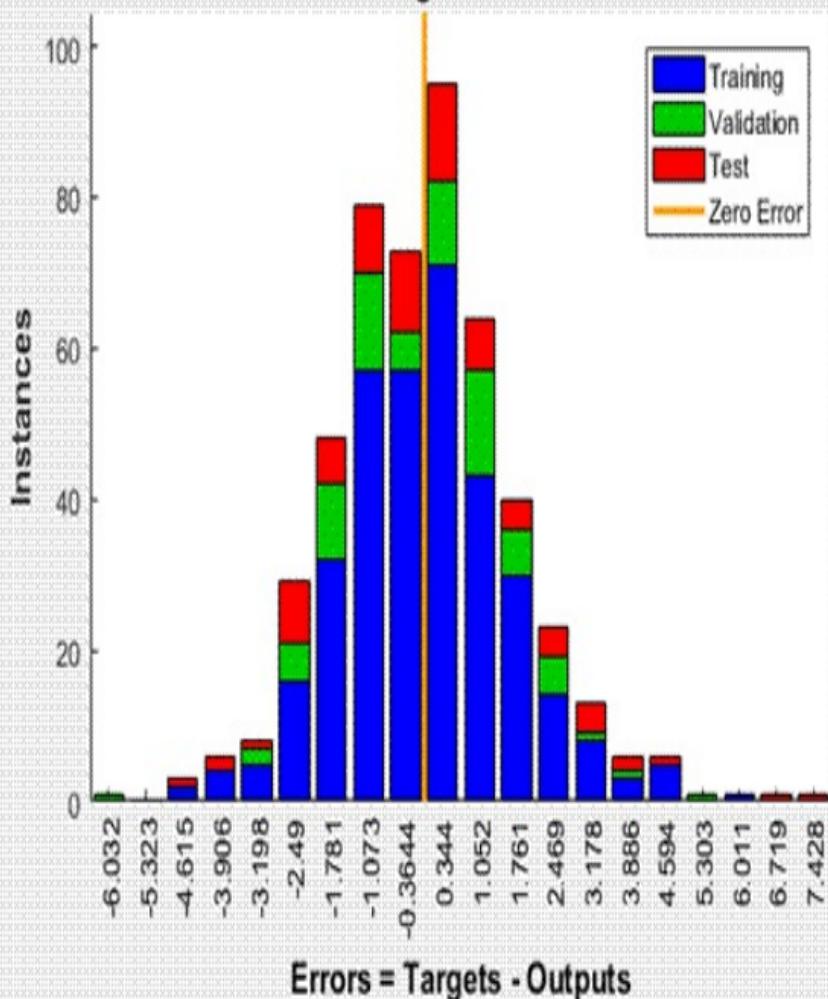


View the error histogram to obtain additional verification of network performance. Under the **Plots** pane, click **Error Histogram**.

# Error Histogram (ploterrhist)

File Edit View Insert Tools Desktop Window Help

## Error Histogram with 20 Bins



The blue bars represent training data, the green bars represent validation data, and the red bars represent testing data. The histogram can give you an indication of outliers, which are data points where the fit is significantly worse than the majority of data. In this case, you can see that while most errors fall between -5 and 5, there is a training point with an error of 17 and validation points with errors of 12 and 13. These outliers are also visible on the testing regression plot. The first corresponds to the point with a target of 50 and output near 33. It is a good idea to check the outliers to

determine if the data is bad, or if those data points are different than the rest of the data set. If the outliers are valid data points, but are unlike the rest of the data, then the network is extrapolating for these points. You should collect more data that looks like the outlier points, and retrain the network.

Click **Next** in the Neural Network Fitting Tool 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

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

Optionally perform additional tests

Inputs:

(none)



Targets:

(none)



Samples are:

Matrix columns

Matrix rows

No inputs selected.

No targets selected.

MSE

R

Test Network

Plot Fit

Plot Error Histogram

Plot Regression

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 do one of the following:

1. Train it again.
2. Increase the number of neurons.
3. Get a larger training data set.

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

number of neurons.

If 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

Use the buttons on this screen to generate scripts or 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 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 have the network saved as net in the workspace. You can perform additional tests on it or put it to work on new inputs.

When you have created the MATLAB

code and saved your results, click **Finish**.

### 3.3 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 at step 14 of the previous section.

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

```
% Script generated by NFTOOL
```

```
%
```

```
% This script assumes these variables are  
defined:
```

%

% houseInputs - input data.

% houseTargets - target data.

inputs = houseInputs;

targets = houseTargets;

% Create a Fitting Network

hiddenLayerSize = 10;

net = fitnet(hiddenLayerSize);

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

net.divideParam.trainRatio = 70/100;

net.divideParam.valRatio = 15/100;

```
net.divideParam.testRatio = 15/100;
```

```
% Train the Network
```

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

```
% Test the Network
```

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

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

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

```
% View the Network
```

```
view(net)
```

```
% Plots
```

% Uncomment these lines to enable various plots.

% figure, plotperform(tr)

% figure, plottrainstate(tr)

% figure, plotfit(targets,outputs)

% figure, plotregression(targets,outputs)

% figure, ploterrhist(errors)

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

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:

1. `load house_dataset`
2. `inputs = houseInputs;`
3. `targets = houseTargets;`

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

`[inputs,targets] = house_dataset;`

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

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

```
hiddenLayerSize = 10;
```

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

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

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

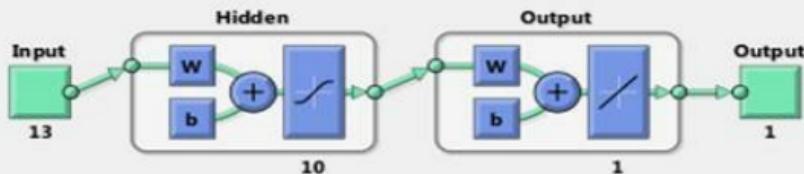
```
net.trainFcn = 'trainbr';
```

```
net.trainFcn = 'trainscg';
```

To train the network, enter:

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

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

**Neural Network****Algorithms**

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

**Progress**

Epoch:	0	20 iterations	1000
Time:		0:00:00	
Performance:	194	2.97	0.00
Gradient:	914	5.59	1.00e-07
Mu:	0.00100	0.0100	1.00e+10
Validation Checks:	0	6	6

**Plots**

- Performance** (plotperform)
- Training State** (plottrainstate)
- Error Histogram** (ploterrhist)
- Regression** (plotregression)
- Fit** (plotfit)

**Plot Interval:**  epochs

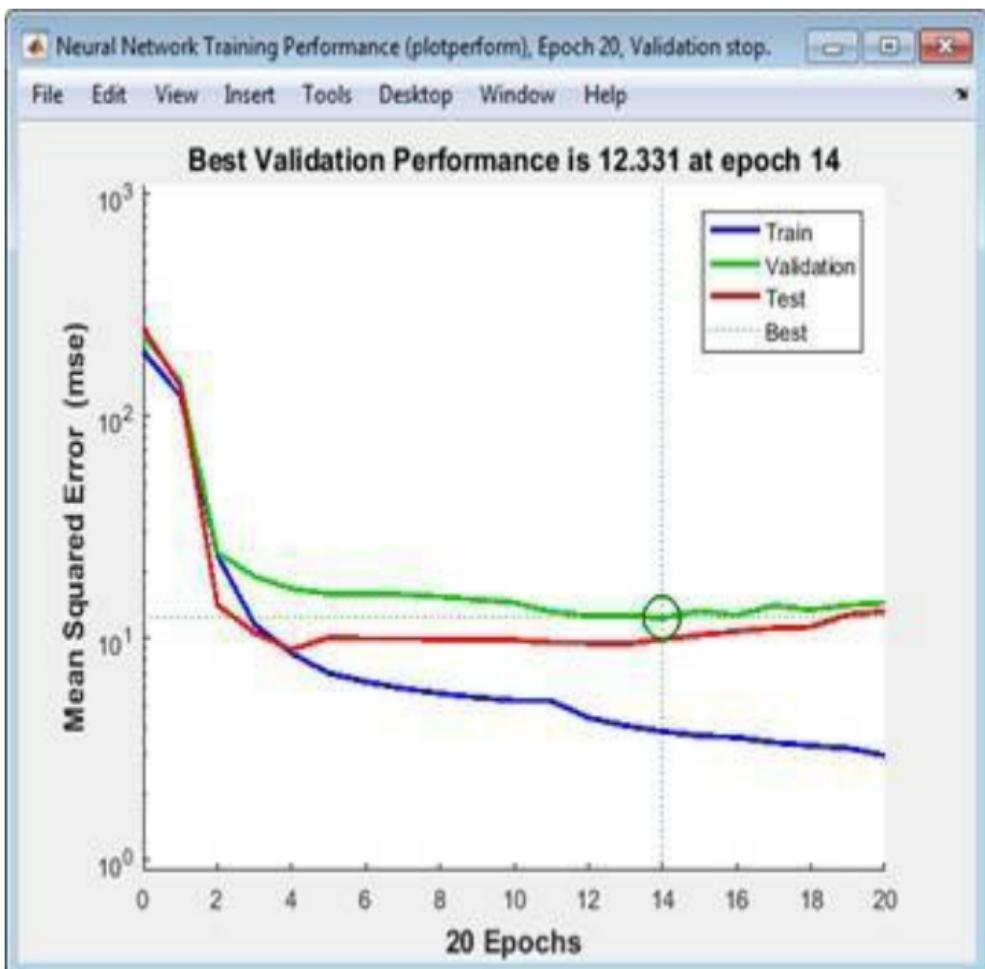


Opening Regression Plot

This training stopped when the validation error increased for six iterations, which occurred at iteration 20. If you click **Performance** in the training window, a plot of the training errors, validation errors, and test errors appears, as shown in the following figure. In this example, the result is reasonable because of the following considerations:

1. The final mean-square error is small.
2. The test set error and the validation set error have similar characteristics.
3. No significant overfitting has

occurred by iteration 14 (where the best validation performance occurs).



Test the network. After the network has

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

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

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

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

```
performance =
```

6.0023

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

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

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

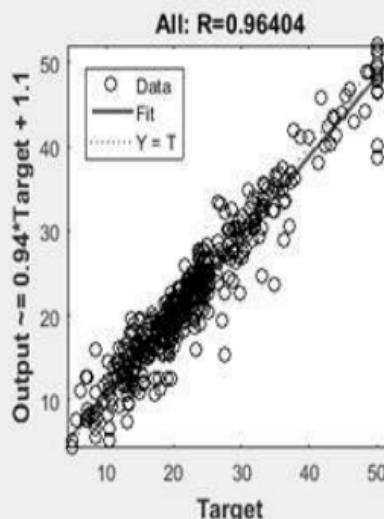
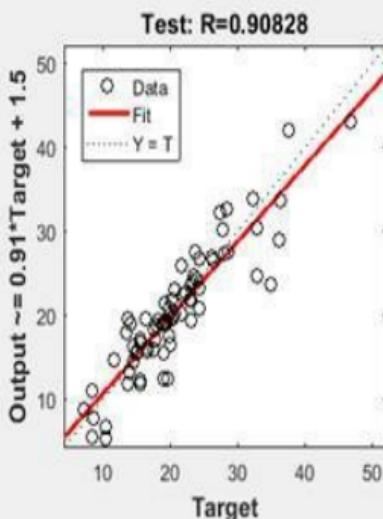
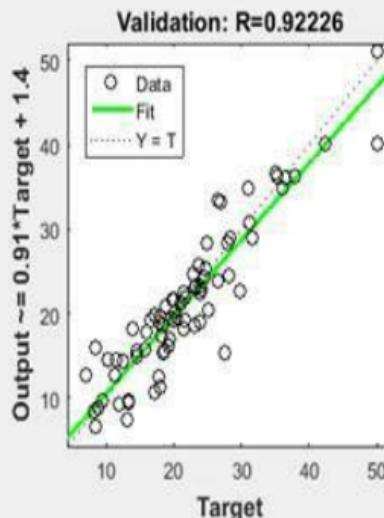
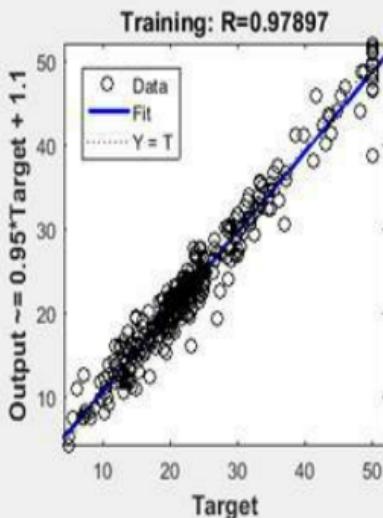
```
tstPerform =  
perform(net,targets(tInd),tstOutputs)
```

```
tstPerform =
```

9.8912

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

The following figure shows the results.



The output tracks the targets very well for training, testing, and validation, and the R-value is over 0.96 for the total response. If even more accurate results were required, you could try any of these approaches:

1. Reset the initial network weights and biases to new values with [init](#) and train again .
2. Increase the number of hidden neurons.
3. Increase the number of training vectors.
4. Increase the number of input values, if more relevant information is

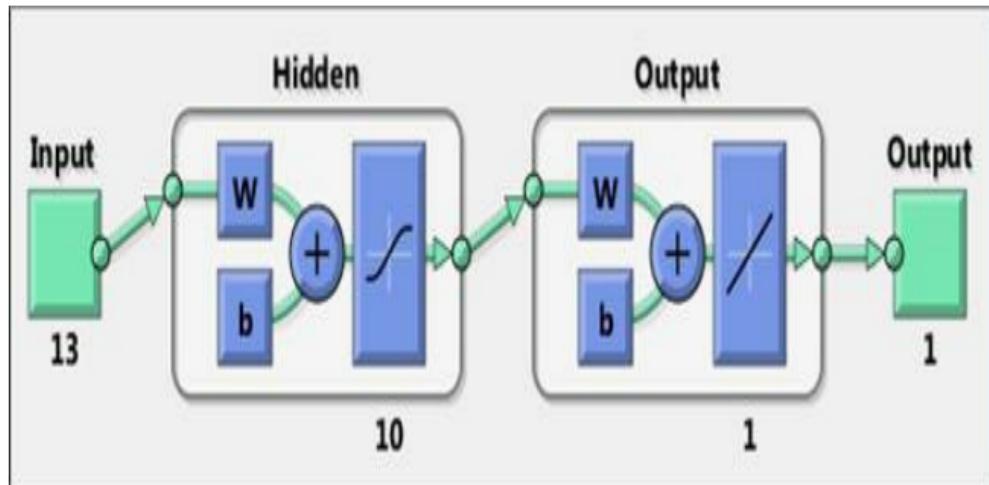
available.

## 5. Try a different training algorithm.

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

View the network diagram.

```
view(net)
```



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

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

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

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a

result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

## **Chapter 4**

# **CLASSIFY PATTERNS WITH A NEURAL NETWORK**

---



## 4.1 INTRODUCTION

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

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

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

- Use the nprtool GUI, as described in [Using the Neural Network Pattern Recognition Tool](#).

- 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.



## **4.2 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.

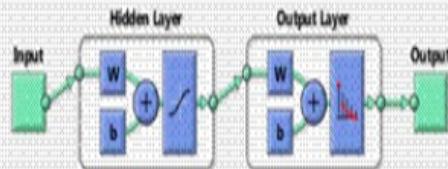
### Introduction

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)

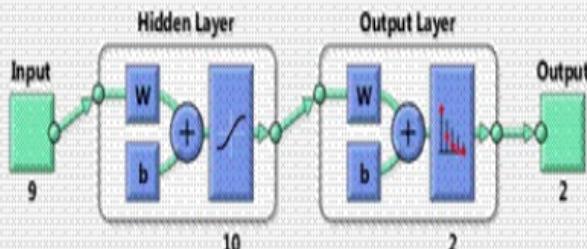
Number of Hidden Neurons:

### 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

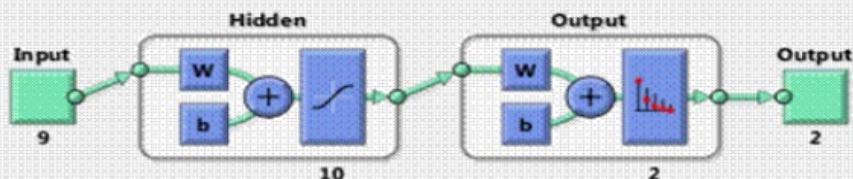
Back

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**

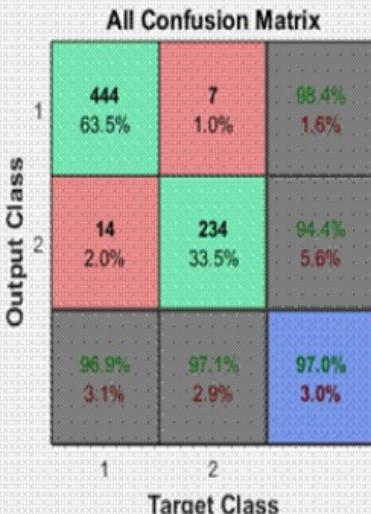
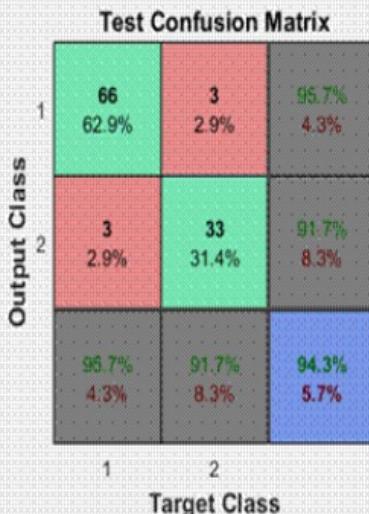
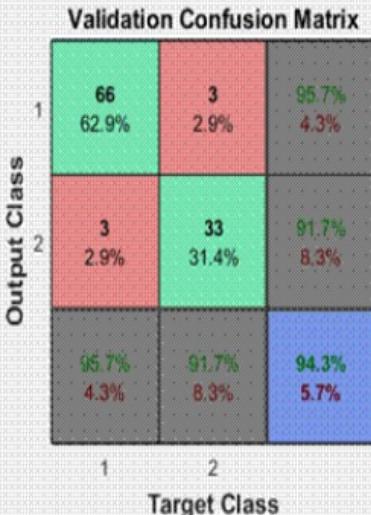
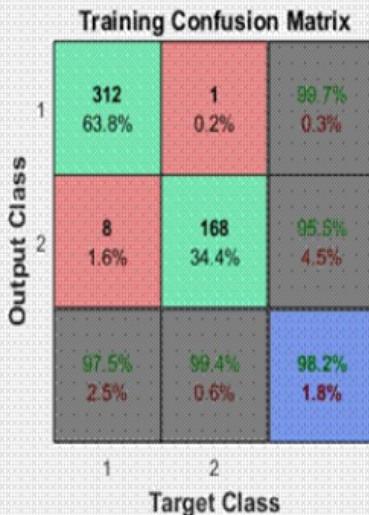
The training continues for 55 iterations.

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)

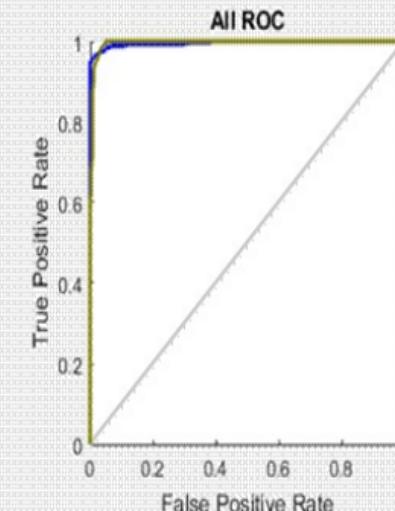
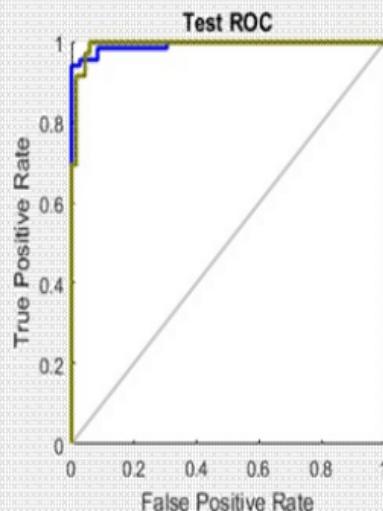
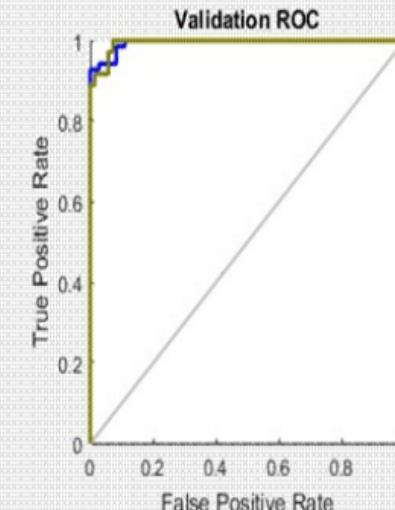
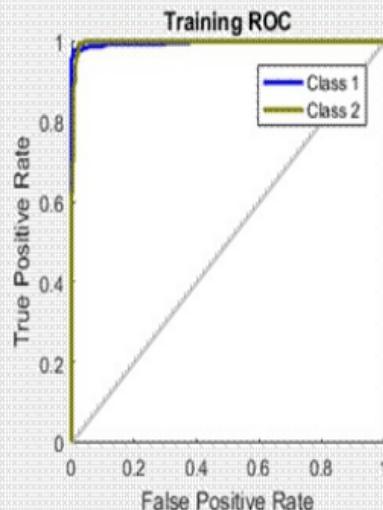
File Edit View Insert Tools Desktop Window Help



- 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**.



## 4.3 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.

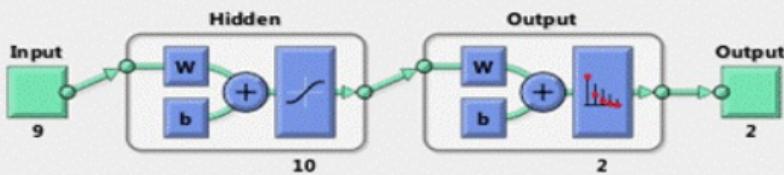
Train the network. The pattern recognition network uses the default Scaled Conjugate Gradient ([trainscg](#)) algorithm for training. To train the network, enter this command:

```
[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:  1 epochs



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));
```

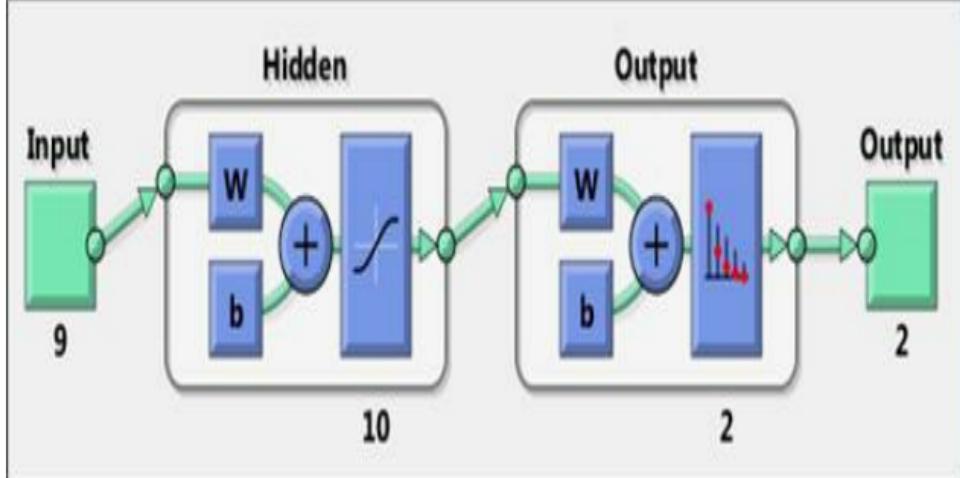
```
tstPerform =  
perform(net,targets(:,tInd),tstOutputs)
```

```
tstPerform =
```

0.0257

View the network diagram.

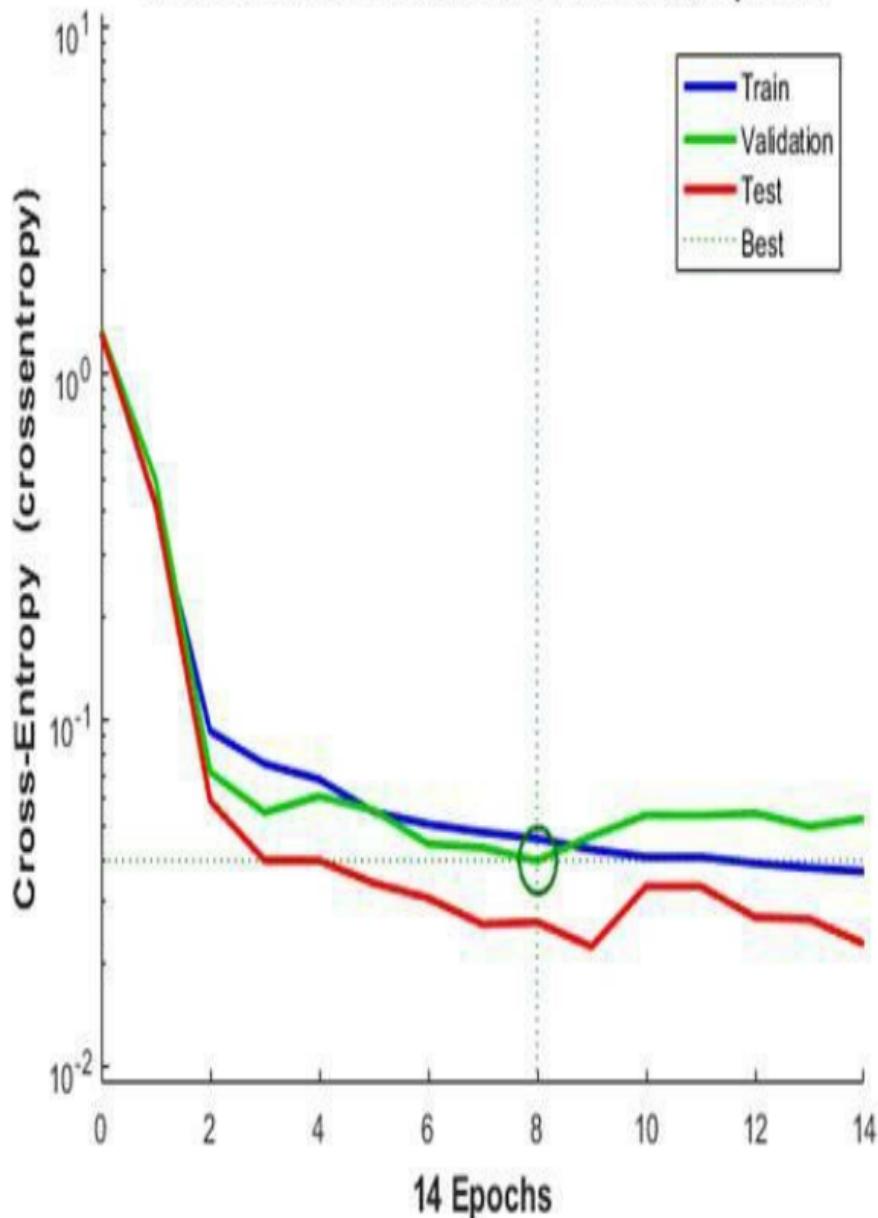
```
view(net)
```



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 5**

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

---

# 5.1 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.

## **5.2 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

(npztool)

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.

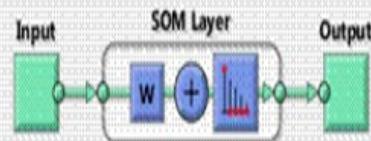
### Introduction

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.



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)

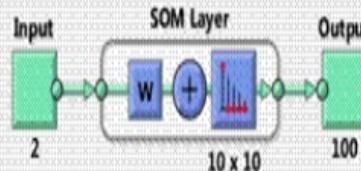
Size of two-dimensional Map:

### 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

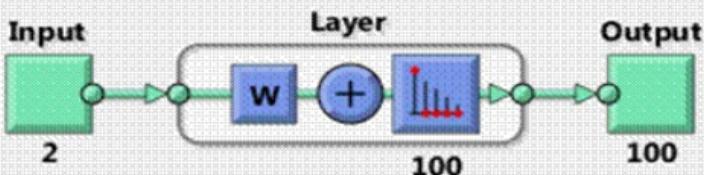
Back

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**.

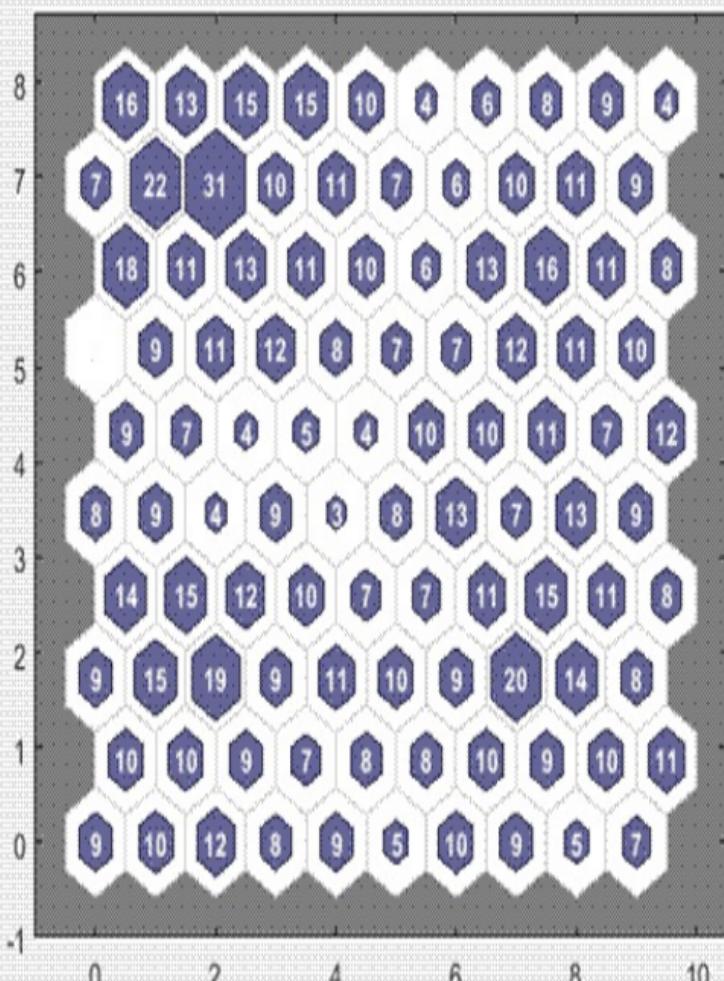


# SOM Sample Hits (plotsomhits)



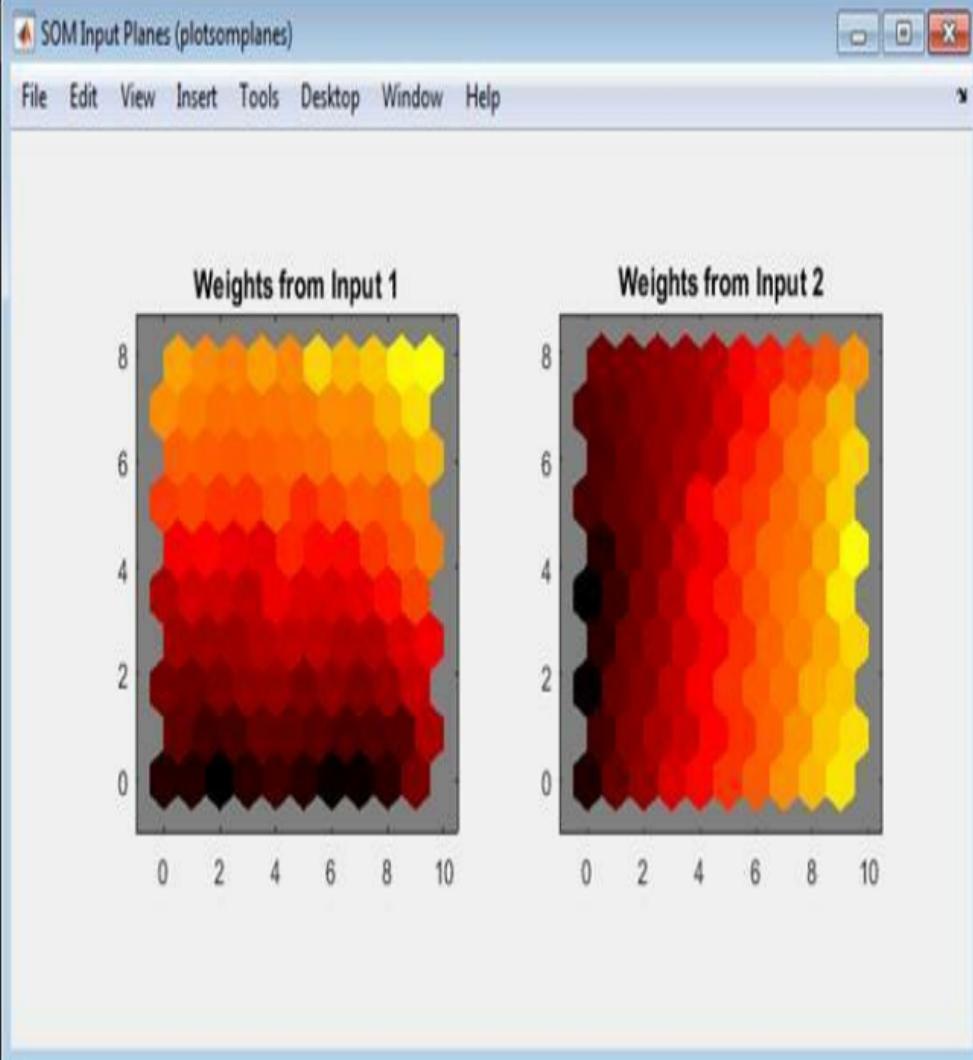
File Edit View Insert Tools Desktop Window Help

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

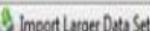
Try training again if a first try did not generate good results or you require marginal improvement.



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.



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



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

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

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.

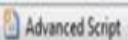
### Generate Scripts

**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 and click [Finish].



You can click **Simple Script** or **Advanced Script** to create MATLAB® code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.

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**.

## 5.3 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]);
```

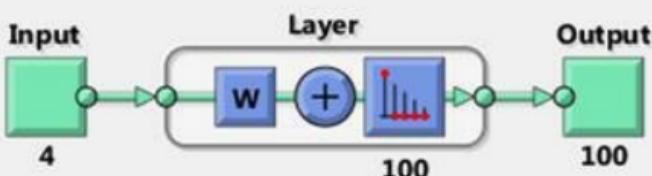
Train the network. The SOM network uses the default batch SOM algorithm for training.

```
[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



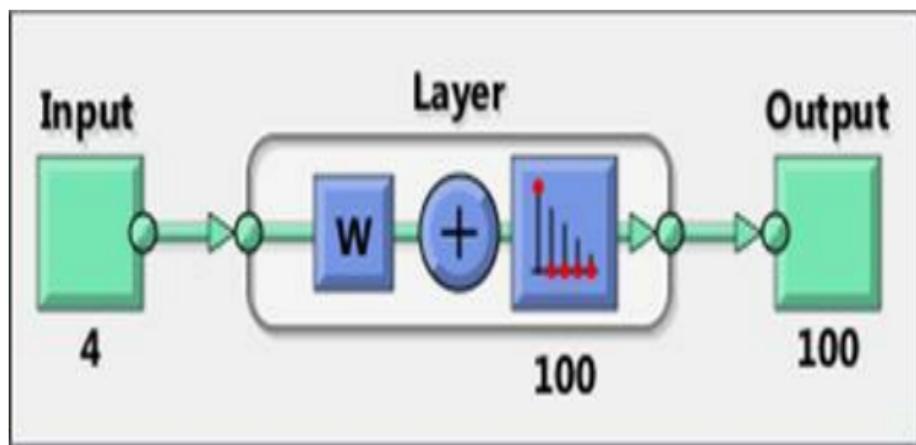
Cancel

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)
```

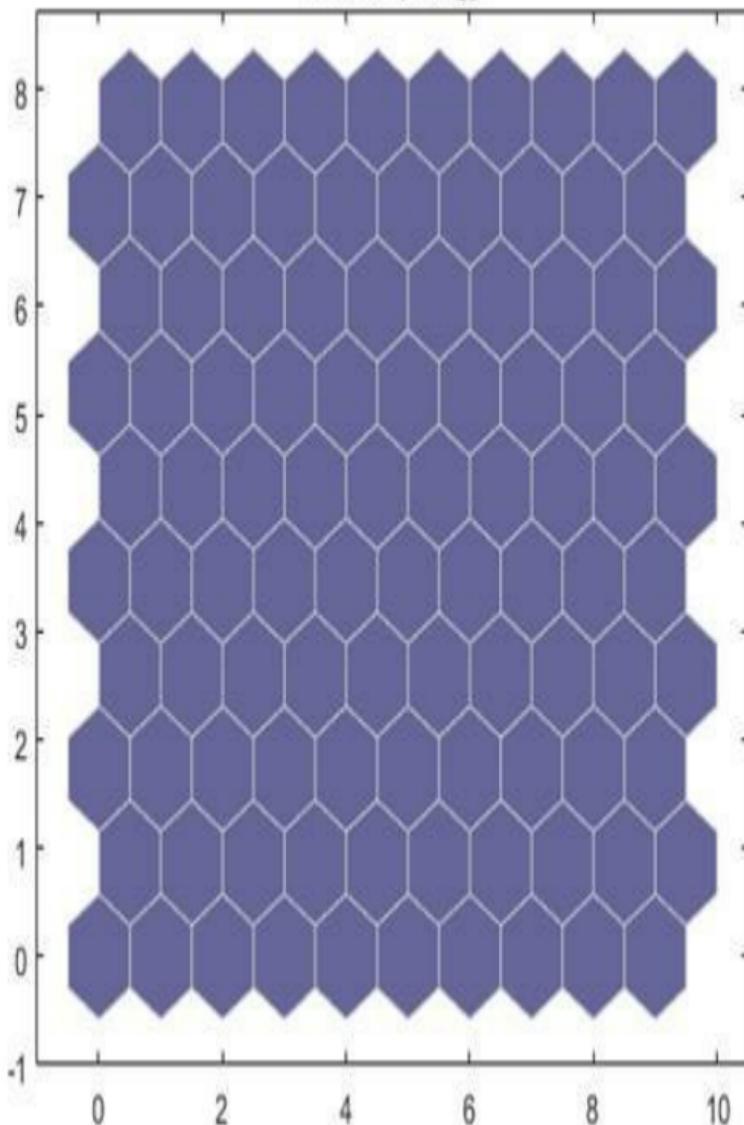


For SOM training, the weight vector associated with each neuron moves to

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*).

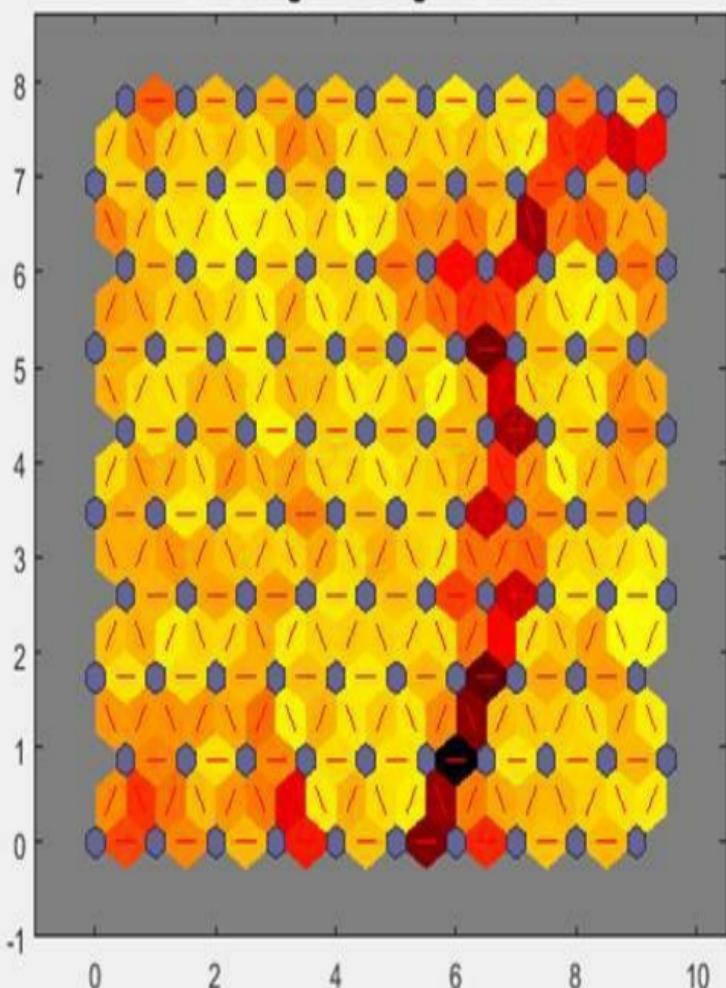
To view the U-matrix, click **SOM Neighbor Distances** in the training window.

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 6**

# **NEURAL NETWORK TIME-SERIES PREDICTION AND**

# MODELING

---

# 6.1 INTRODUCTION

Dynamic neural networks are good at time-series prediction.

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

You can solve this problem in two ways:

- Use a graphical user interface, [ntstool](#).
- Use 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. Each GUI has access to many sample data sets that you can use to experiment with the toolbox. If you have a specific problem that you want to solve, you can load your own data into the workspace. The next section describes the data format.

To define a time-series problem for the toolbox, arrange a set of TS input vectors as columns in a cell array. Then, arrange another set of TS target vectors (the correct output vectors for each of

the input vectors) into a second cell array. However, there are cases in which you only need to have a target data set. For example, you can define the following time-series problem, in which you want to use previous values of a series to predict the next value:

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

The next section shows how to train a network to fit a time-series data set, using the neural network time-series tool GUI, [\*ntstool\*](#). This example uses the pH neutralization data set provided with the toolbox.

## **6.2 USING THE NEURAL NETWORK TIME SERIES 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 Time Series Tool** to open  
the Neural Network Time Series Tool.  
(You can also use the command [ntstool](#).)



## Welcome to the Neural Network Time Series Tool.

Solve a nonlinear time series problem with a dynamic neural network.

### Introduction

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

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

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

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

### Select a Problem

Nonlinear Autoregressive with External (Exogenous) Input (NARX)

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



Nonlinear Autoregressive (NAR)

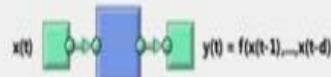
Predict series  $y(t)$  given  $d$  past values of  $y(t)$ .



Nonlinear Input-Output

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

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



Choose a problem, then click [Next].

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

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

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

$$(t - d))$$

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

In the second type of time-series problem, there is only one series involved. The future values of a time series  $y(t)$  are predicted only from past values of that series. This form of prediction is called nonlinear

autoregressive, or NAR, and can be written as follows:

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

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

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

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

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

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



## Select Data

What inputs and targets define your nonlinear autoregressive problem?

### Get Data from Workspace

Input time series  $x(t)$ :



(none)



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



(none)



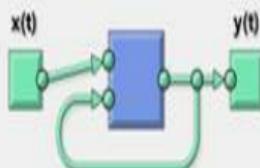
Select the time series format. (tonndata)

Time step:  [■] Cell column  [■] Matrix column  [■] Matrix row

### Summary

No inputs selected.

No targets selected.



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

[Load Example Data Set](#)



Select inputs and targets, then click [Next].

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

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

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



Select a data set:

Description

**Simple NARX Problem**

Heat Exchanger

Magnetic Levitation

pH Neutralization Process

Pollution Mortality

Fluid Flow in Pipe

Filename: [simplenarx\\_dataset](#)

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

This dataset can be used to demonstrate how a neural network can be trained to make predictions.

LOAD [simplenarx\\_dataset](#).MAT loads these two variables:

simpnarxInputs - a 1x100 cell array of scalar values representing a 100 timestep time-series.

simpnarxTargets - a 1x100 cell array of scalar values representing a 100 timestep time-series to be predicted.

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

For an intro to prediction with the [Neural Time Series app](#)



# Select pH Neutralization Process, and

click **Import**. This returns you to the Select Data window.

Click **Next** to open the Validation and Test Data window, shown in the following figure.

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



## Validation and Test Data

Set aside some target timesteps for validation and testing.

### Select Percentages

Randomly divide up the 2001 target timesteps:

Training:

70%

1401 target timesteps

Validation:

15% ▾

300 target timesteps

Testing:

15% ▾

300 target timesteps

### Explanation

Three Kinds of Target Timesteps:

Training:

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

Validation:

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

Testing:

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

Restore Defaults

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

Neural Network Start

Welcome

Back

Next

Cancel

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

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

Click Next.



## Network Architecture

Choose the number of neurons and input/feedback delays.

### Architecture Choices

Define a NARX neural network. (narxnet)

Number of Hidden Neurons:

10

Number of delays d:

2

Problem definition:

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

[Restore Defaults](#)

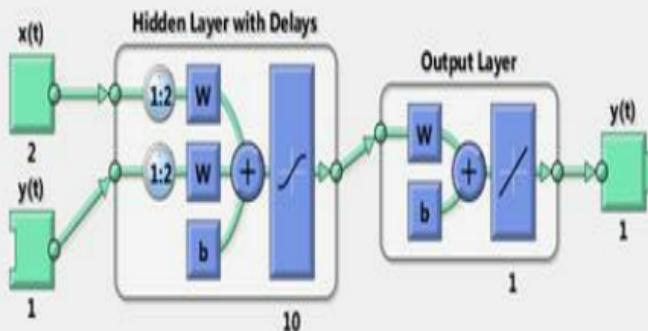
### Recommendation

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

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

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

### Neural Network



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

[Neural Network Start](#)

[Welcome](#)

[Back](#)

[Next](#)

[Cancel](#)

The standard NARX network is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This network also uses tapped delay lines to store previous values of the  $x(t)$  and  $y(t)$  sequences.

Note that the output of the NARX network,  $y(t)$ , is fed back to the input of the network (through delays), since  $y(t)$  is a function of  $y(t - 1)$ ,  $y(t - 2)$ , ...,  $y(t - d)$ . However, for efficient training this feedback loop can be opened.

Because the true output is available during the training of the network, you can use the open-loop architecture

shown above, in which the true output is used instead of feeding back the estimated output. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and therefore a more efficient algorithm can be used for training. This network is discussed in more detail in ["NARX Network"](#) (`narxnet`, `closeloop`).

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

**Click Next.**



## Train Network

Train the network to fit the inputs and targets.

### Train Network

#### Choose a training algorithm:

Levenberg-Marquardt

This algorithm typically requires more memory but less time. Training automatically stops when generalization stops improving, as indicated by an increase in the mean square error of the validation samples.

Train using Levenberg-Marquardt. (trainlm)



### Results

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

Plot Error Histogram

Plot Response

Plot Error Autocorrelation

Plot Input-Error Correlation

### Notes

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

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

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

Train network, then click [Next].

Neural Network Start

Welcome

Back

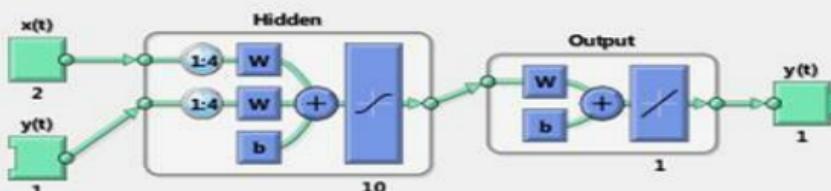
Next

Cancel

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

The training continued until the validation error failed to decrease for

six iterations (validation stop).

**Neural Network****Algorithms**

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

**Progress**

Epoch:	0	39 iterations	1000
Time:		0:00:00	
Performance:	33.5	0.00196	0.00
Gradient:	79.8	0.0756	1.00e-07
Mu:	0.00100	1.00e-07	1.00e+10
Validation Checks:	0	6	6

**Plots**

**Performance** (plotperform)

**Training State** (plottrainstate)

**Error Histogram** (ploterrhist)

**Regression** (plotregression)

**Time-Series Response** (plotresponse)

**Error Autocorrelation** (ploterrcorr)

**Input-Error Cross-correlation** (plotinerrcorr)

**Plot Interval:**



**Validation stop.**



**Stop Training**



**Cancel**

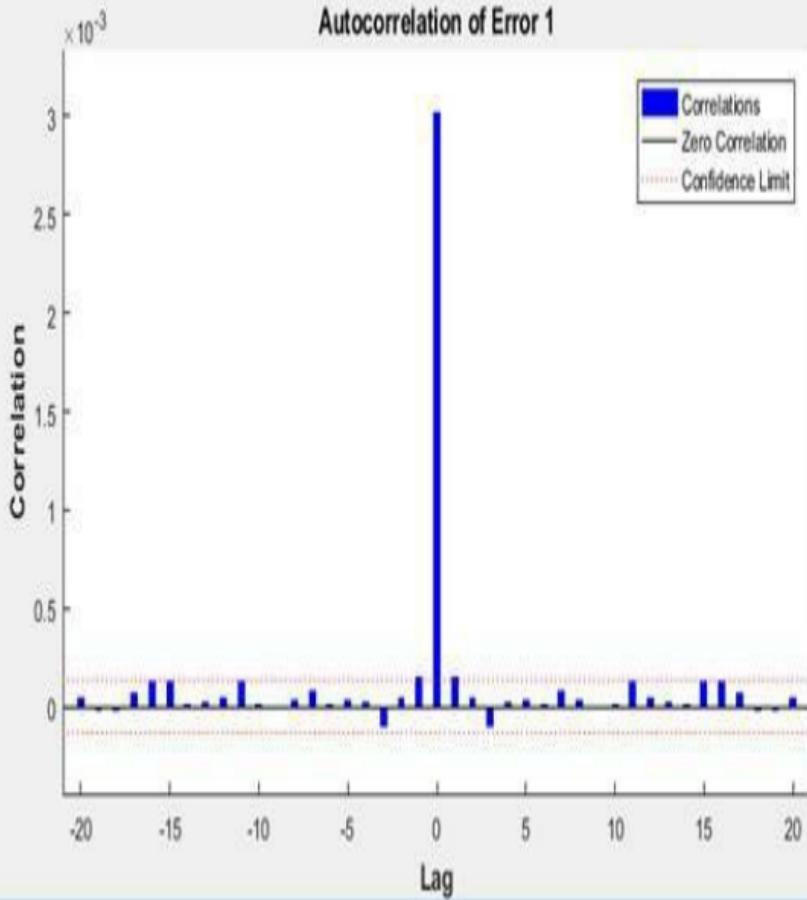
Under **Plots**, click **Error Autocorrelation**. This is used to validate the network performance.

The following plot displays the error autocorrelation function. It describes how the prediction errors are related in time. For a perfect prediction model, there should only be one nonzero value of the autocorrelation function, and it should occur at zero lag. (This is the mean square error.) This would mean that the prediction errors were completely uncorrelated with each other (white noise). If there was significant correlation in the prediction errors, then

it should be possible to improve the prediction - perhaps by increasing the number of delays in the tapped delay lines. In this case, the correlations, except for the one at zero lag, fall approximately within the 95% confidence limits around zero, so the model seems to be adequate. If even more accurate results were required, you could retrain the network by clicking **Retrain** in [ntstool](#). This will change the initial weights and biases of the network, and may produce an improved network after retraining.

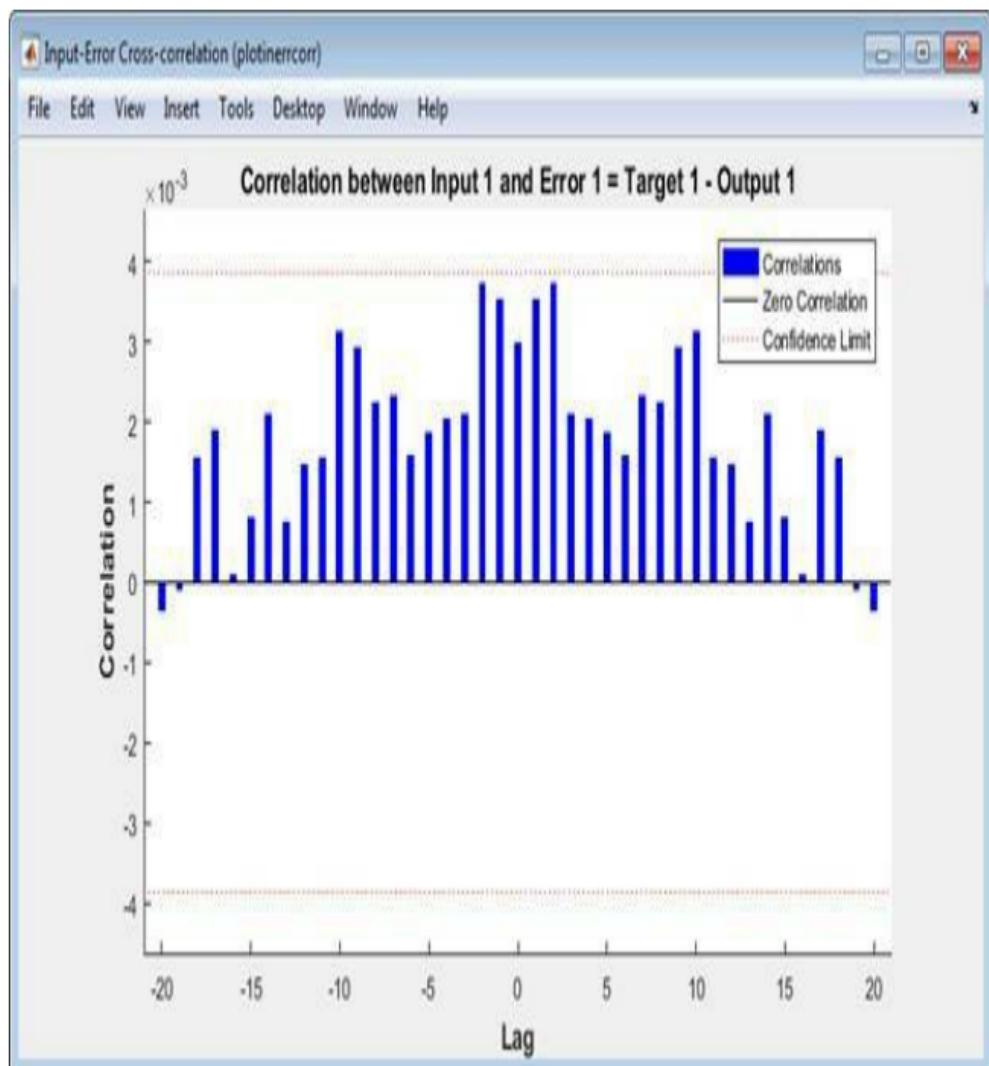
### Error Autocorrelation (plottercorr)

File Edit View Insert Tools Desktop Window Help



View the input-error cross-correlation function to obtain additional verification

of network performance. Under the **Plots** pane, click **Input-Error Cross-correlation**.



This input-error cross-correlation function illustrates how the errors are correlated with the input sequence  $x(t)$ . For a perfect prediction model, all of the correlations should be zero. If the input is correlated with the error, then it should be possible to improve the prediction, perhaps by increasing the number of delays in the tapped delay lines. In this case, all of the correlations fall within the confidence bounds around zero.

Under **Plots**, click **Time Series Response**. This displays the inputs, targets and errors versus time. It also indicates which time points were

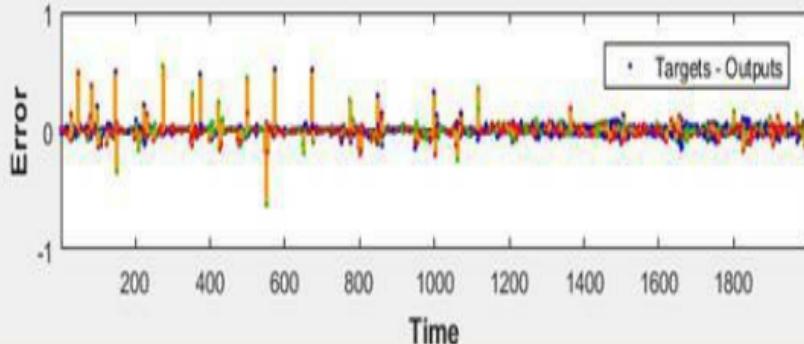
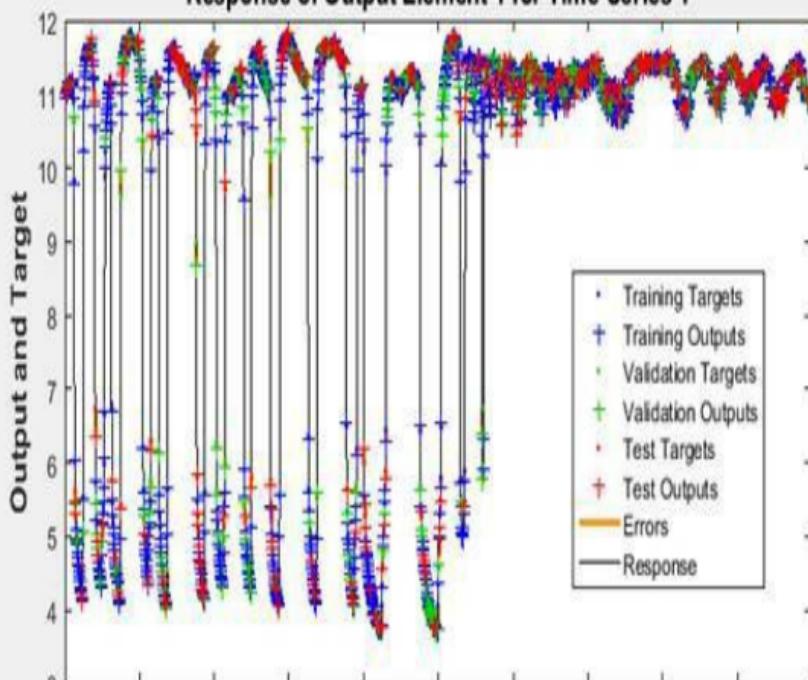
selected for training, testing and validation.

# Time-Series Response (plotresponse)



File Edit View Insert Tools Desktop Window Help

## Response of Output Element 1 for Time-Series 1



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



## Evaluate Network

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

### Iterate for improved performance

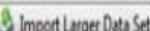
Try training again if a first try did not generate good results or you require marginal improvement.



Increase network size if retraining did not help.



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



### Optionally perform additional tests



(none)



(none)



Time step:



Cell column



Matrix column



Matrix row

No inputs selected.

No targets selected.



MSE



R



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



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

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

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

If the performance on the training set is good, but the test set performance is significantly worse, which could indicate overfitting, then reducing the

number of neurons can improve your results.

If 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)

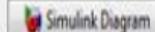


### Simulink Deployment

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

Generate a Simulink diagram:

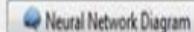
(gensim)



### 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 generate scripts or 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:

net1

Save performance and data set information to MATLAB struct named:

info

Save outputs to MATLAB matrix named:

output

Save errors to MATLAB matrix named:

error

Save inputs to MATLAB matrix named:

input

Save feedback to MATLAB matrix named:

feedback

Save ALL selected values above to MATLAB struct named:

results



Save results and click [Finish].

Neural Network Start

Welcome

Back

Next

Finish

You can click **Simple Script** or **Advanced Script** to create MATLAB code that can be used to reproduce all of the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process.

You can also have the network saved as net in the workspace. You can perform additional tests on it or put it to work on new inputs.

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

## 6.3 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 at step 15 of the previous section.

```
% Solve an Autoregression Problem with  
External  
% Input with a NARX Neural Network  
% Script generated by NTSTOOL  
%  
% This script assumes the variables on the right  
of  
% these equalities are defined:
```

%

% phInputs - input time series.

% phTargets - feedback time series.

inputSeries = phInputs;

targetSeries = phTargets;

% Create a Nonlinear Autoregressive Network  
with External Input

inputDelays = 1:4;

feedbackDelays = 1:4;

hiddenLayerSize = 10;

net =

narxnet(inputDelays,feedbackDelays,hiddenLay

% Prepare the Data for Training and Simulation

% The function PREPARETS prepares time  
series data

% for a particular network, shifting time by the  
minimum

% amount to fill input states and layer states.

```
% Using PREPARETS allows you to keep your  
original  
% time series data unchanged, while easily  
customizing it  
% for networks with differing numbers of  
delays, with  
% open loop or closed loop feedback modes.  
[inputs,inputStates,layerStates,targets] = ...  
    preparets(net,inputSeries,{},targetSeries);
```

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

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

```
% Train the Network
```

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

```
% Test the Network
```

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

```
% View the Network
view(net)
```

```
% Plots
```

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

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

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

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

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

```
% figure, ploterrcorr(errors)
```

```
% figure, plotinerrcorr(inputs,errors)
```

```
% Closed Loop Network
```

```
% Use this network to do multi-step
prediction.
```

```
% The function CLOSELOOP replaces the
```

feedback input with a direct  
% connection from the output layer.

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

% Early Prediction Network

% For some applications it helps to get the prediction a

% timestep early.

% The original network returns predicted  $y(t+1)$  at the same

% time it is given  $y(t+1)$ .

% For some applications such as decision making, it would

% help to have predicted  $y(t+1)$  once  $y(t)$  is available, but

% before the actual  $y(t+1)$  occurs.  
% The network can be made to return its output  
a timestep early  
% by removing one delay so that its minimal  
tap delay is now  
% 0 instead of 1. The new network returns the  
same outputs as  
% the original network, but outputs are shifted  
left one timestep.

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

You can save the script, and then run it from the command line to reproduce the results of the previous GUI session. You

can also edit the script to customize the training process. In this case, follow each of the steps in the script.

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

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

Create a network. The NARX network, [narxnet](#), is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. This network has two inputs. One is an external input, and the other is a feedback connection from the network output. (After the network has been trained, this feedback connection can

be closed, as you will see at a later step.) For each of these inputs, there is a tapped delay line to store previous values. To assign the network architecture for a NARX network, you must select the delays associated with each tapped delay line, and also the number of hidden layer neurons. In the following steps, you assign the input delays and the feedback delays to range from 1 to 4 and the number of hidden neurons to be 10.

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

Note Increasing the number of neurons and the number of delays requires more computation, and this has a tendency to

overfit the data when the numbers are set too high, but it allows the network to solve more complicated problems. More layers require more computation, but their use might result in the network solving complex problems more efficiently. To use more than one hidden layer, enter the hidden layer sizes as elements of an array in the [fitnet](#) command.

Prepare the data for training. When training a network containing tapped delay lines, it is necessary to fill the delays with initial values of the inputs and outputs of the network. There is a toolbox command that facilitates this process - [preprets](#). This function has three input arguments: the network, the

input sequence and the target sequence. The function returns the initial conditions that are needed to fill the tapped delay lines in the network, and modified input and target sequences, where the initial conditions have been removed. You can call the function as follows:

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

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

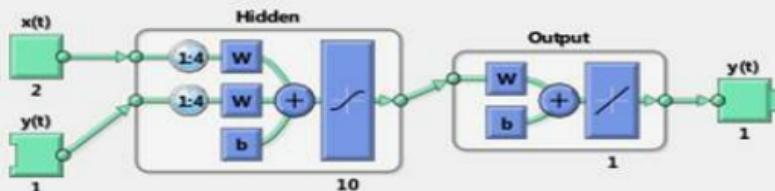
```
net.trainFcn = 'trainbr';
```

```
net.trainFcn = 'trainscg';
```

To train the network, enter:

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

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

**Neural Network****Algorithms**

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

**Progress**

Epoch:	0	44 iterations	1000
Time:		0:00:00	
Performance:	79.1	0.00219	0.00
Gradient:	134	0.0187	1.00e-07
Mu:	0.00100	1.00e-05	1.00e+10
Validation Checks:	0	6	6

**Plots**

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

Plot Interval:  1 epochs



Validation stop.

This training stopped when the validation error increased for six iterations, which occurred at iteration 44.

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

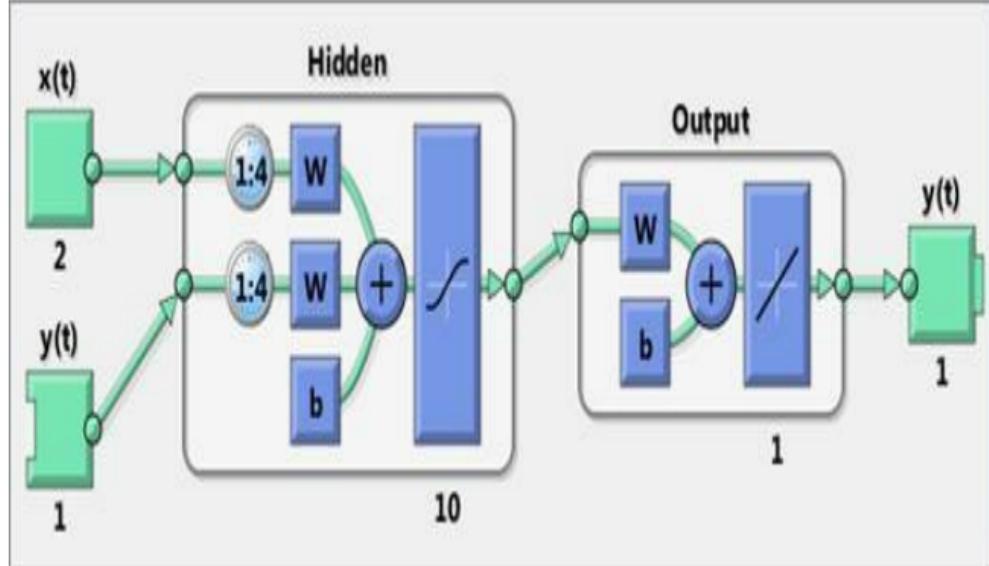
```
outputs = net(inputs,inputStates,layerStates);  
errors = gsubtract(targets,outputs);  
performance = perform(net,targets,outputs)
```

performance =

0.0042

View the network diagram.

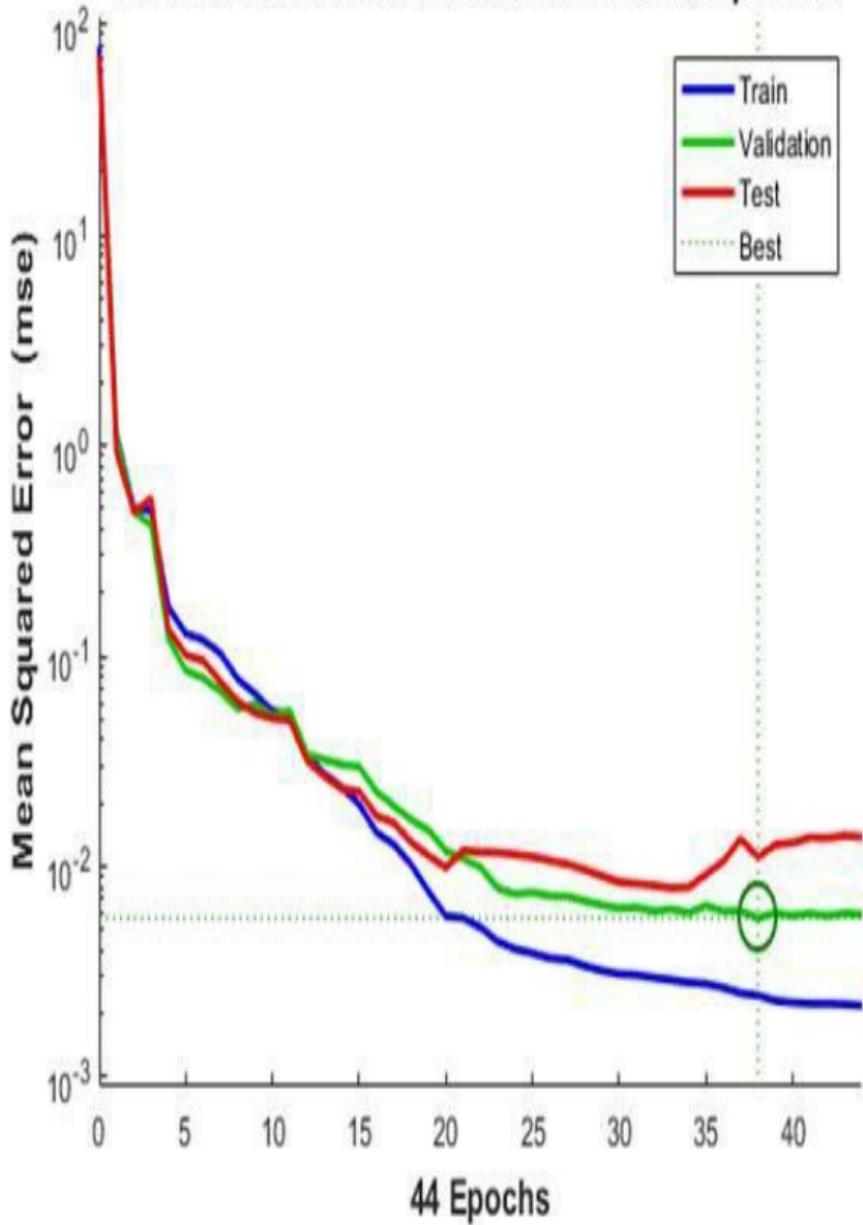
```
view(net)
```



Plot the performance training record to check for potential overfitting.

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

Best Validation Performance is 0.0056917 at epoch 38



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

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

computed based on the open-loop training results.

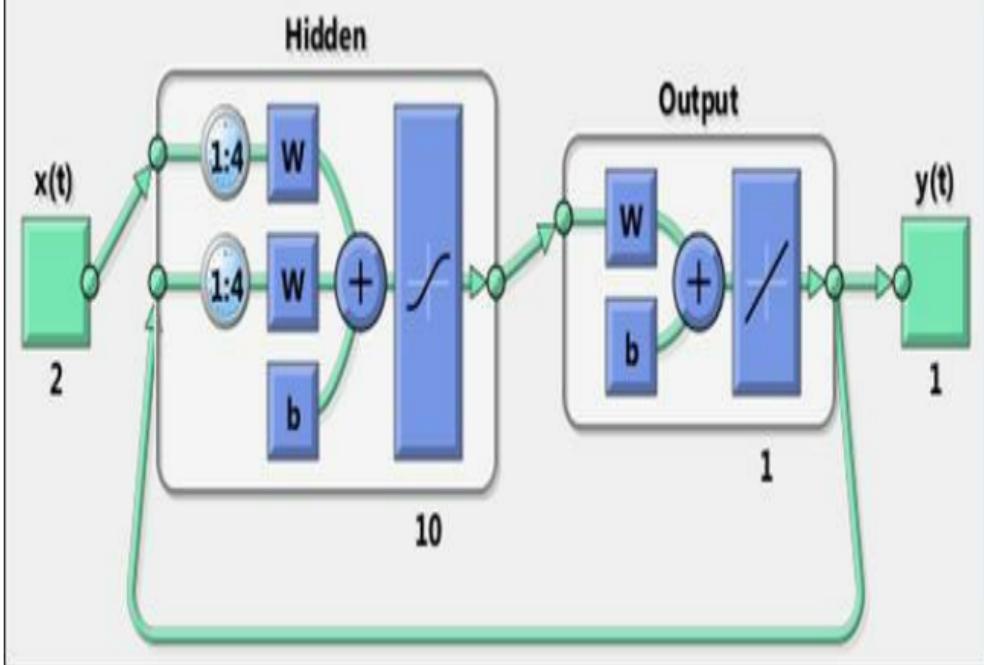
Close the loop on the NARX network. When the feedback loop is open on the NARX network, it is performing a one-step-ahead prediction. It is predicting the next value of  $y(t)$  from previous values of  $y(t)$  and  $x(t)$ . With the feedback loop closed, it can be used to perform multi-step-ahead predictions.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

```
netc = closeloop(net);  
netc.name = [net.name ' - Closed Loop'];
```

```
view(netc)
[xc,xic,aic,tc] = prepares(netc,inputSeries,
{},targetSeries);
yc = netc(xc,xic,aic);
perfc = perform(netc,tc,yc)
```

perfc =

2.8744



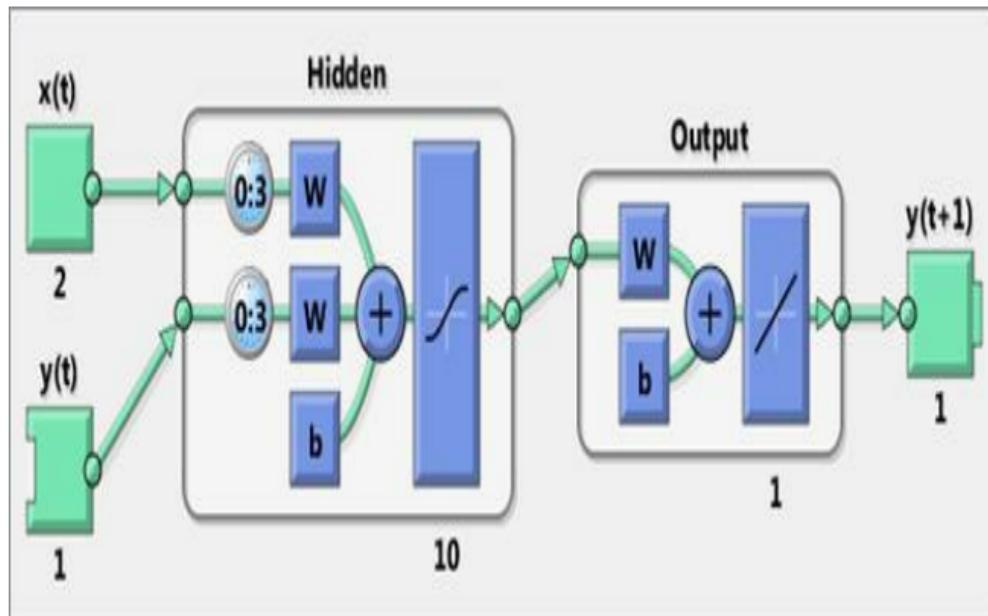
Remove a delay from the network, to get the prediction one time step early.

```

nets = removedelay(net);
nets.name = [net.name '- Predict One Step
Ahead'];
view(nets)
[xs,xis,ais,ts] = prepares(nets,inputSeries,

```

```
{},targetSeries);  
ys = nets(xs,xis,ais);  
earlyPredictPerformance = perform(nets,ts,ys)  
  
earlyPredictPerformance =  
  
0.0042
```



From this figure, you can see that the network is identical to the previous

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

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

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

- Increase the number of training vectors.
- Increase the number of input values, if more relevant information is available.
- Try a different training algorithm.

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

- During training, open a plot window (such as the error correlation plot), and watch it animate.
- Plot from the command line with functions such as [plotresponse](#), [ploterrcorr](#) and [plo](#)

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

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

There are several other techniques for improving upon initial solutions if higher accuracy is desired.



## **Chapter 7**

# **PARALLEL COMPUTING ON CPUS AND GPUS**

---



# 7.1 PARALLEL COMPUTING TOOLBOX

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

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

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

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

## 7.2 PARALLEL CPU WORKERS

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

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

```
pool = gcp;  
pool.NumWorkers
```

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

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

```
net2 = train(net1,x,t,'useParallel','yes');  
y = net2(x,'useParallel','yes');
```

## 7.3 GPU COMPUTING

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

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

`gpuDeviceCount`

`gpuDevice`

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

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

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

```
net2 = train(net1,x,t,'useGPU','yes');
```

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

## 7.4 MULTIPLE GPU/CPU COMPUTING

You can use multiple GPUs for higher levels of parallelism.

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

```
net2 =
```

```
train(net1,x,t,'useParallel','yes','useGPU','yes');  
y = net2(x,'useParallel','yes','useGPU','yes');
```

For some problems, using GPUs and

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

```
net2 =  
train(net1,x,t,'useParallel','yes','useGPU','only');  
y = net2(x,'useParallel','yes','useGPU','only');
```

# **7.5 CLUSTER COMPUTING WITH MATLAB DISTRIBUTED COMPUTING SERVER**

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

After opening a parallel pool, train the network by calling `train` with

the 'useParallel' and 'useGPU' options.

```
net2 = train(net1,x,t,'useParallel','yes');
```

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

```
net2 =
```

```
train(net1,x,t,'useParallel','yes','useGPU','only');
```

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

# **7.6 LOAD BALANCING, LARGE PROBLEMS, AND BEYOND.**

**NEURAL NETWORKS  
WITH PARALLEL AND  
GPU COMPUTING**

## 7.6.1 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.

## 7.6.2 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 **Parallel 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
```

Now you can train and simulate the neural network with data split by sample across all the workers. To do this, set:

the train and sim parameter 'useParallel' to 'yes'.

```
net2 = train(net1,x,t,'useParallel','yes')  
y = net2(x,'useParallel','yes')
```

Use the '*showResources*' argument to verify that the calculations ran across multiple workers.

```
net2 =  
train(net1,x,t,'useParallel','yes','showResources'  
y =  
net2(x,'useParallel','yes','showResources','yes');
```

MATLAB indicates which resources were used. For example:

## Computing Resources: Parallel Workers

Worker 1 on MyComputer, MEX on  
PCWIN64

Worker 2 on MyComputer, MEX on  
PCWIN64

Worker 3 on MyComputer, MEX on  
PCWIN64

Worker 4 on MyComputer, MEX on  
PCWIN64

When `train` and `sim` are called, they divide the input matrix or cell array data into distributed Composite values before training and simulation. When `sim` has calculated a Composite, this output is converted back to the same matrix or cell array form before it is returned.

However, you might want to perform this

data division manually if:

- The problem size is too large for the host computer. Manually defining the elements of Composite values sequentially allows much bigger problems to be defined.
- It is known that some workers are on computers that are faster or have more memory than others. You can distribute the data with differing numbers of samples per worker. This is called load balancing.
- The following code sequentially creates a series of random datasets and saves them to separate files:

```
pool = gcp;
for i=1:pool.NumWorkers
    x = rand(2,1000);
    save(['inputs' num2str(i)],'x');
    t = x(1,:).*x(2,:)+2*(x(1,:)+x(2,:));
    save(['targets' num2str(i)],'t');
    clear x t
end
```

Because the data was defined sequentially, you can define a total dataset larger than can fit in the host PC memory. PC memory must accommodate only a sub-dataset at a time.

Now you can load the datasets sequentially across parallel workers, and train and simulate a network on the Composite data. When train or sim is called with Composite data,

the 'useParallel' argument is automatically set to 'yes'. When using Composite data, configure the network's input and outputs to match one of the datasets manually using the configure function before training.

```
xc = Composite;
tc = Composite;
for i=1:pool.NumWorkers
    data = load(['inputs' num2str(i)],'x');
    xc{i} = data.x;
    data = load(['targets' num2str(i)],'t');
    tc{i} = data.t;
    clear data
end
net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc);
yc = net2(xc);
```

To convert the Composite output

returned by sim, you can access each of its elements, separately if concerned about memory limitations.

```
for i=1:pool.NumWorkers  
    yi = yc{i}  
end
```

Combined the Composite value into one local value if you are not concerned about memory limitations.

```
y = {yc{:}};
```

When load balancing, the same process happens, but, instead of each dataset having the same number of samples (1000 in the previous example), the numbers of samples can be adjusted to best take advantage of the memory and

speed differences of the worker host computers.

It is not required that each worker have data. If element  $i$  of a Composite value is undefined, worker  $i$  will not be used in the computation.

## 7.6.3 Single GPU Computing

The number of cores, size of memory, and speed efficiencies of GPU cards are growing rapidly with each new generation. Where video games have long benefited from improved GPU performance, these cards are now flexible enough to perform general numerical computing tasks like training neural networks.

For the latest GPU requirements, see the web page for Parallel Computing Toolbox; or query MATLAB to determine whether your PC has a supported GPU. This function returns the number of GPUs in your system:

```
count = gpuDeviceCount
```

```
count =
```

```
1
```

If the result is one or more, you can query each GPU by index for its characteristics. This includes its name, number of multiprocessors, SIMDWidth of each multiprocessor, and total memory.

```
gpu1 = gpuDevice(1)
```

```
gpu1 =
```

CUDADevice with properties:

Name: 'GeForce GTX 470'

Index: 1

ComputeCapability: '2.0'

SupportsDouble: 1  
DriverVersion: 4.1000  
MaxThreadsPerBlock: 1024  
MaxShmemPerBlock: 49152  
MaxThreadBlockSize: [1024 1024 64]  
MaxGridSize: [65535 65535 1]  
SIMDWidth: 32  
TotalMemory: 1.3422e+09  
AvailableMemory: 1.1056e+09  
MultiprocessorCount: 14  
ClockRateKHz: 1215000  
ComputeMode: 'Default'  
GPUOverlapsTransfers: 1  
KernelExecutionTimeout: 1  
CanMapHostMemory: 1  
DeviceSupported: 1  
DeviceSelected: 1

The simplest way to take advantage of the GPU is to specify call train and sim with the parameter

argument 'useGPU' set to 'yes' ('no' is the default).

```
net2 = train(net1,x,t,'useGPU','yes')
y = net2(x,'useGPU','yes')
```

If `net1` has the default training function `trainlm`, you see a warning that GPU calculations do not support Jacobian training, only gradient training. So the training function is automatically changed to the gradient training function `trainscg`. To avoid the notice, you can specify the function before training:

```
net1.trainFcn = 'trainscg';
```

To verify that the training and simulation occur on the GPU device, request that

the computer resources be shown:

```
net2 =  
train(net1,x,t,'useGPU','yes','showResources','ye  
y =  
net2(x,'useGPU','yes','showResources','yes')
```

Each of the above lines of code outputs the following resources summary:

Computing Resources:

GPU device #1, GeForce GTX 470

Many MATLAB functions automatically execute on a GPU when any of the input arguments is a gpuArray. Normally you move arrays to and from the GPU with the functions gpuArray and gather. However, for neural network calculations on a GPU to be efficient, matrices need to be transposed and the

columns padded so that the first element in each column aligns properly in the GPU memory. Neural Network Toolbox provides a special function called [nndata2gpu](#) to move an array to a GPU and properly organize it:

```
xg = nndata2gpu(x);  
tg = nndata2gpu(t);
```

Now you can train and simulate the network using the converted data already on the GPU, without having to specify the 'useGPU' argument. Then convert and return the resulting GPU array back to MATLAB with the complementary function [gpu2nndata](#).

Before training with gpuArray data, the network's input and outputs must be

manually configured with regular MATLAB matrices using the configure function:

```
net2 = configure(net1,x,t); % Configure with  
MATLAB arrays  
net2 = train(net2,xg,tg); % Execute on GPU  
with NNET formatted gpuArrays  
yg = net2(xg); % Execute on GPU  
y = gpu2nndata(yg); % Transfer array to  
local workspace
```

On GPUs and other hardware where you might want to deploy your neural networks, it is often the case that the exponential function `exp` is not implemented with hardware, but with a software library. This can slow down neural networks that use the `tansig` sigmoid transfer function. An

alternative function is the Elliot sigmoid function whose expression does not include a call to any higher order functions:

$$(equation) \quad a = n / (1 + \text{abs}(n))$$

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.

## 7.6.4 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','yes','sl  
y =  
net2(x,'useParallel','yes','useGPU','yes','showRe
```

These lines of code use all available workers in the parallel pool. One worker for each unique GPU employs that GPU, while other workers operate as CPUs. In some cases, it might be faster to use only GPUs. For instance, if a single computer has three GPUs and four workers each, the three workers that are accelerated by the three GPUs might be speed limited by the fourth CPU worker. In these cases, you can specify that train and sim use only workers with unique GPUs.

```
net2 =
```

```
train(net1,x,t,'useParallel','yes','useGPU','only','  
y =  
net2(x,'useParallel','yes','useGPU','only','showR
```

As with simple distributed computing, distributed GPU computing can benefit from manually created Composite values. Defining the Composite values yourself lets you indicate which workers to use, how many samples to assign to each worker, and which workers use GPUs.

For instance, if you have four workers and only three GPUs, you can define larger datasets for the GPU workers. Here, a random dataset is created with different sample loads per Composite element:

```
numSamples = [1000 1000 1000 300];
xc = Composite;
tc = Composite;
for i=1:4
    xi = rand(2,numSamples(i));
    ti = xi(1,:).^2 + 3*xi(2,:);
    xc{i} = xi;
    tc{i} = ti;
end
```

You can now specify that train and sim use the three GPUs available:

```
net2 = configure(net1,xc{1},tc{1});
net2 =
train(net2,xc,tc,'useGPU','yes','showResources';
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.

## 7.6.5 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](#)).

## 7.6.6 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','yes')
```

```
net2 = train(net1,x,t,xi,ai,'useParallel','yes')  
y =  
net2(x,xi,ai,'useParallel','yes','useGPU','only')
```

```
net2 =  
train(net1,x,t,xi,ai,'useParallel','yes','useGPU','o  
y =
```

```
net2(x,xi,ai,'useParallel','yes','useGPU','only')
```

Note that parallelism happens across samples, or in the case of time series across different series. However, if the network has only input delays, with no layer delays, the delayed inputs can be precalculated so that for the purposes of computation, the time steps become different samples and can be parallelized. This is the case for networks such as timedelaynet and open-loop versions of narxnet and narnet. If a network has layer delays, then time cannot be "flattened" for purposes of computation, and so single series data cannot be parallelized. This is the case for networks such as layrecnet and closed-loop versions

of narxnet and narnet. However, if the data consists of multiple sequences, it can be parallelized across the separate sequences.

## 7.6.7 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`' s

to 'yes' to verify what resources were actually used.

## **Capítulo 8**

# **NEURAL NETWORK TOOLBOX SAMPLE DATA SETS AND**

# GLOSSARY

---

# **8.1 NEURAL NETWORK TOOLBOX SAMPLE DATA SETS**

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

```
help nndatasets
```

Neural Network Datasets

---

Function Fitting, Function approximation and Curve fitting.

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

- simplefit\_dataset
  - Simple fitting dataset.
- abalone\_dataset
  - Abalone shell rings dataset.
- bodyfat\_dataset
  - Body fat percentage dataset.
- building\_dataset
  - Building energy dataset.
- chemical\_dataset
  - Chemical sensor dataset.
- cho\_dataset
  - Cholesterol dataset.
- engine\_dataset
  - Engine behavior dataset.

house\_dataset

- House value dataset

---

## Pattern Recognition and Classification

Pattern recognition is the process of training a neural network

to assign the correct target classes to a set of input patterns.

Once trained the network can be used to classify patterns it has not seen before.

simpleclass\_dataset - Simple pattern recognition dataset.

cancer\_dataset - Breast cancer dataset.

crab\_dataset - Crab gender dataset.

glass\_dataset - Glass chemical dataset.

iris\_dataset - Iris flower dataset.

thyroid\_dataset - Thyroid function

dataset.

wine\_dataset

- Italian wines dataset.

---

Clustering, Feature extraction and Data dimension reduction

Clustering is the process of training a neural network on

patterns so that the network comes up with its own

classifications according to pattern similarity and relative

topology. This is useful for gaining insight into data, or

simplifying it before further processing.

simplecluster\_dataset - Simple clustering dataset.

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

---

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

Input-output time series problems consist of predicting the next value of one time-series given another time-series.

Past values of both series (for best accuracy), or only one of the series (for a simpler system) may be used to predict the target series.

`simpleseries_dataset` - Simple time-series prediction dataset.

`simplenarx_dataset` - Simple time-series prediction dataset.

`exchanger_dataset` - Heat exchanger dataset.

`maglev_dataset` - Magnetic levitation dataset.

`ph_dataset` - Solution PH dataset.

`pollution_dataset` - Pollution mortality dataset.

`valve_dataset` - Valve fluid flow dataset.

-----

Single Time-Series Prediction, Forecasting,  
Dynamic modelling,

Nonlinear autoregression, System  
identification, and Filtering

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

`simplenar_dataset` - Simple single series prediction dataset.

`chickenpox_dataset` - Monthly chickenpox instances dataset.

`ice_dataset` - Gobal ice volume dataset.

`laser_dataset` - Chaotic far-infrared laser dataset.

`oil_dataset` - Monthly oil price dataset.

`river_dataset` - River flow dataset.

`solar_dataset` - Sunspot activity dataset

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

command such as

```
load simplefit_dataset
```

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

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

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

```
help maglev_dataset
```



# **8.2 GLOSARY**

## **ADALINE**

Acronym for a linear neuron: ADaptive LINear Element.

## **adaption**

Training method that proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.

## **adaptive filter**

Network that contains delays and whose weights are adjusted after each new input vector is presented. The network adapts to changes in the input signal properties if such occur. This kind of filter is used in long

distance telephone lines to cancel echoes.

## **adaptive learning rate**

Learning rate that is adjusted according to an algorithm during training to minimize training time.

## **architecture**

Description of the number of the layers in a neural network, each layer's transfer function, the number of neurons per layer, and the connections between layers.

## **backpropagation learning rule**

Learning rule in which weights and biases are adjusted by error-derivative (delta) vectors backpropagated through the network. Backpropagation is commonly applied to feedforward multilayer networks. Sometimes

this rule is called the *generalized delta rule*.

## **backtracking search**

Linear search routine that begins with a step multiplier of 1 and then backtracks until an acceptable reduction in performance is obtained.

## **batch**

Matrix of input (or target) vectors applied to the network simultaneously. Changes to the network weights and biases are made just once for the entire set of vectors in the input matrix. (The term *batch* is being replaced by the more descriptive expression "concurrent vectors.")

## **batching**

Process of presenting a set of input vectors for simultaneous calculation of a matrix of output

vectors and/or new weights and biases.

## **Bayesian framework**

Assumes that the weights and biases of the network are random variables with specified distributions.

## **BFGS quasi-Newton algorithm**

Variation of Newton's optimization algorithm, in which an approximation of the Hessian matrix is obtained from gradients computed at each iteration of the algorithm.

## **bias**

Neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.

## **bias vector**

Column vector of bias values for a layer of neurons.

## **Brent's search**

Linear search that is a hybrid of the golden section search and a quadratic interpolation.

## **cascade-forward network**

Layered network in which each layer only receives inputs from previous layers.

## **Charalambous' search**

Hybrid line search that uses a cubic interpolation together with a type of sectioning.

## **classification**

Association of an input vector with a particular target vector.

## **competitive layer**

Layer of neurons in which only the neuron with maximum net input has an output of 1 and all other neurons have an output of 0. Neurons compete with each other for the right to respond to a given input vector.

## **competitive learning**

Unsupervised training of a competitive layer with the instar rule or Kohonen rule. Individual neurons learn to become feature detectors. After training, the layer categorizes input vectors among its neurons.

## **competitive transfer function**

Accepts a net input vector for a layer and

returns neuron outputs of 0 for all neurons except for the winner, the neuron associated with the most positive element of the net input  $\mathbf{n}$ .

## **concurrent input vectors**

Name given to a matrix of input vectors that are to be presented to a network simultaneously. All the vectors in the matrix are used in making just one set of changes in the weights and biases.

## **conjugate gradient algorithm**

In the conjugate gradient algorithms, a search is performed along conjugate directions, which produces generally faster convergence than a search along the steepest descent directions.

## **connection**

One-way link between neurons in a network.

## **connection strength**

Strength of a link between two neurons in a network. The strength, often called weight, determines the effect that one neuron has on another.

## **cycle**

Single presentation of an input vector, calculation of output, and new weights and biases.

## **dead neuron**

Competitive layer neuron that never won any competition during training and so has not become a useful feature detector. Dead neurons do not respond to any of the training vectors.

## **decision boundary**

Line, determined by the weight and bias vectors, for which the net input  $n$  is zero.

## **delta rule**

See **Widrow-Hoff learning rule**.

## **delta vector**

The delta vector for a layer is the derivative of a network's output error with respect to that layer's net input vector.

## **distance**

Distance between neurons, calculated from their positions with a distance function.

## **distance function**

Particular way of calculating distance, such as

the Euclidean distance between two vectors.

## **early stopping**

Technique based on dividing the data into three subsets. The first subset is the training set, used for computing the gradient and updating the network weights and biases. The second subset is the validation set. When the validation error increases for a specified number of iterations, the training is stopped, and the weights and biases at the minimum of the validation error are returned. The third subset is the test set. It is used to verify the network design.

## **epoch**

Presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.

## **error jumping**

Sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.

## **error ratio**

Training parameter used with adaptive learning rate and momentum training of backpropagation networks.

## **error vector**

Difference between a network's output vector in response to an input vector and an associated target output vector.

## **feedback network**

Network with connections from a layer's output to that layer's input. The feedback connection

can be direct or pass through several layers.

## **feedforward network**

Layered network in which each layer only receives inputs from previous layers.

## **Fletcher-Reeves update**

Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

## **function approximation**

Task performed by a network trained to respond to inputs with an approximation of a desired function.

## **generalization**

Attribute of a network whose output for a new

input vector tends to be close to outputs for similar input vectors in its training set.

## **generalized regression network**

Approximates a continuous function to an arbitrary accuracy, given a sufficient number of hidden neurons.

## **global minimum**

Lowest value of a function over the entire range of its input parameters. Gradient descent methods adjust weights and biases in order to find the global minimum of error for a network.

## **golden section search**

Linear search that does not require the calculation of the slope. The interval containing the minimum of the performance is subdivided

at each iteration of the search, and one subdivision is eliminated at each iteration.

## **gradient descent**

Process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.

## **hard-limit transfer function**

Transfer function that maps inputs greater than or equal to 0 to 1, and all other values to 0.

## **Hebb learning rule**

Historically the first proposed learning rule for neurons. Weights are adjusted proportional to the product of the outputs of pre- and postweight neurons.

## **hidden layer**

Layer of a network that is not connected to the network output (for instance, the first layer of a two-layer feedforward network).

## **home neuron**

Neuron at the center of a neighborhood.

## **hybrid bisection-cubic search**

Line search that combines bisection and cubic interpolation.

## **initialization**

Process of setting the network weights and biases to their original values.

## **input layer**

Layer of neurons receiving inputs directly from

outside the network.

## **input space**

Range of all possible input vectors.

## **input vector**

Vector presented to the network.

## **input weight vector**

Row vector of weights going to a neuron.

## **input weights**

Weights connecting network inputs to layers.

## **Jacobian matrix**

Contains the first derivatives of the network errors with respect to the weights and biases.

## Kohonen learning rule

Learning rule that trains a selected neuron's weight vectors to take on the values of the current input vector.

## layer

Group of neurons having connections to the same inputs and sending outputs to the same destinations.

## layer diagram

Network architecture figure showing the layers and the weight matrices connecting them. Each layer's transfer function is indicated with a symbol. Sizes of input, output, bias, and weight matrices are shown. Individual neurons and connections are not shown. (See Network Objects, Data and Training Styles in the *Neural Network Toolbox User's Guide*.)

## **layer weights**

Weights connecting layers to other layers. Such weights need to have nonzero delays if they form a recurrent connection (i.e., a loop).

## **learning**

Process by which weights and biases are adjusted to achieve some desired network behavior.

## **learning rate**

Training parameter that controls the size of weight and bias changes during learning.

## **learning rule**

Method of deriving the next changes that might be made in a network *or* a procedure for modifying the weights and biases of a network.

## **Levenberg-Marquardt**

Algorithm that trains a neural network 10 to 100 times faster than the usual gradient descent backpropagation method. It always computes the approximate Hessian matrix, which has dimensions  $n$ -by- $n$ .

## **line search function**

Procedure for searching along a given search direction (line) to locate the minimum of the network performance.

## **linear transfer function**

Transfer function that produces its input as its output.

## **link distance**

Number of links, or steps, that must be taken to

get to the neuron under consideration.

## local minimum

Minimum of a function over a limited range of input values. A local minimum might not be the global minimum.

## log-sigmoid transfer function

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

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

## Manhattan distance

The Manhattan distance between two vectors  $\mathbf{x}$  and  $\mathbf{y}$  is calculated as

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

## **maximum performance increase**

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

## **maximum step size**

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

## **mean square error function**

Performance function that calculates the average squared error between the network outputs  $\mathbf{a}$  and the target outputs  $\mathbf{t}$ .

## **momentum**

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

## **momentum constant**

Training parameter that controls how much momentum is used.

## **mu parameter**

Initial value for the scalar  $\mu$ .

## **neighborhood**

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

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

## **net input vector**

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

## **neuron**

Basic processing element of a neural network. Includes weights and bias, a summing junction, and an output transfer function. Artificial neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons.

## **neuron diagram**

Network architecture figure showing the neurons and the weights connecting them. Each neuron's transfer function is indicated with a symbol.

## **ordering phase**

Period of training during which neuron weights are expected to order themselves in the input space consistent with the associated neuron positions.

## **output layer**

Layer whose output is passed to the world outside the network.

## **output vector**

Output of a neural network. Each element of the output vector is the output of a neuron.

## **output weight vector**

Column vector of weights coming from a neuron or input. (See also **outstar learning rule**.)

## **outstar learning rule**

Learning rule that trains a neuron's (or input's) output weight vector to take on the values of the current output vector of the postweight layer. Changes in the weights are proportional to the neuron's output.

## **overfitting**

Case in which the error on the training set is driven to a very small value, but when new data is presented to the network, the error is large.

## **pass**

Each traverse through all the training input and target vectors.

## **pattern**

A vector.

## **pattern association**

Task performed by a network trained to respond with the correct output vector for each input vector presented.

## **pattern recognition**

Task performed by a network trained to respond when an input vector close to a learned vector is presented. The network "recognizes" the input as one of the original target vectors.

## **perceptron**

Single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.

## **perceptron learning rule**

Learning rule for training single-layer hard-limit networks. It is guaranteed to result in a perfectly functioning network in finite time,

given that the network is capable of doing so.

## **performance**

Behavior of a network.

## **performance function**

Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions. Type nnets and look under performance functions.

## **Polak-Ribière update**

Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure.

## **positive linear transfer function**

Transfer function that produces an output of zero for negative inputs and an output equal to the input for positive inputs.

## **postprocessing**

Converts normalized outputs back into the same units that were used for the original targets.

## **Powell-Beale restarts**

Method for computing a set of conjugate directions. These directions are used as search directions as part of a conjugate gradient optimization procedure. This procedure also periodically resets the search direction to the negative of the gradient.

## **preprocessing**

Transformation of the input or target data

before it is presented to the neural network.

## **principal component analysis**

Orthogonalize the components of network input vectors. This procedure can also reduce the dimension of the input vectors by eliminating redundant components.

## **quasi-Newton algorithm**

Class of optimization algorithm based on Newton's method. An approximate Hessian matrix is computed at each iteration of the algorithm based on the gradients.

## **radial basis networks**

Neural network that can be designed directly by fitting special response elements where they will do the most good.

## **radial basis transfer function**

The transfer function for a radial basis neuron is

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

## **regularization**

Modification of the performance function, which is normally chosen to be the sum of squares of the network errors on the training set, by adding some fraction of the squares of the network weights.

## **resilient backpropagation**

Training algorithm that eliminates the harmful effect of having a small slope at the extreme ends of the sigmoid squashing transfer functions.

## **saturating linear transfer function**

Function that is linear in the interval (-1,+1) and saturates outside this interval to -1 or +1.  
(The toolbox function is satlin.)

## **scaled conjugate gradient algorithm**

Avoids the time-consuming line search of the standard conjugate gradient algorithm.

## **sequential input vectors**

Set of vectors that are to be presented to a network one after the other. The network weights and biases are adjusted on the presentation of each input vector.

## **sigma parameter**

Determines the change in weight for the calculation of the approximate Hessian matrix

in the scaled conjugate gradient algorithm.

## **sigmoid**

Monotonic S-shaped function that maps numbers in the interval  $(-\infty, \infty)$  to a finite interval such as  $(-1, +1)$  or  $(0, 1)$ .

## **simulation**

Takes the network input **p**, and the network object **net**, and returns the network outputs **a**.

## **spread constant**

Distance an input vector must be from a neuron's weight vector to produce an output of 0.5.

## **squashing function**

Monotonically increasing function that takes input values between  $-\infty$  and  $+\infty$  and returns

values in a finite interval.

## **star learning rule**

Learning rule that trains a neuron's weight vector to take on the values of the current input vector. Changes in the weights are proportional to the neuron's output.

## **sum-squared error**

Sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.

## **supervised learning**

Learning process in which changes in a network's weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets.

## **symmetric hard-limit transfer function**

Transfer that maps inputs greater than or equal to 0 to +1, and all other values to -1.

## **symmetric saturating linear transfer function**

Produces the input as its output as long as the input is in the range -1 to 1. Outside that range the output is -1 and +1, respectively.

## **tan-sigmoid transfer function**

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

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

## **tapped delay line**

Sequential set of delays with outputs available at each delay output.

## **target vector**

Desired output vector for a given input vector.

## **test vectors**

Set of input vectors (not used directly in training) that is used to test the trained network.

## **topology functions**

Ways to arrange the neurons in a grid, box, hexagonal, or random topology.

## **training**

Procedure whereby a network is adjusted to do a particular job. Commonly viewed as an offline job, as opposed to an adjustment made

during each time interval, as is done in adaptive training.

## **training vector**

Input and/or target vector used to train a network.

## **transfer function**

Function that maps a neuron's (or layer's) net output  $\mathbf{n}$  to its actual output.

## **tuning phase**

Period of SOFM training during which weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

## **underdetermined system**

System that has more variables than constraints.

## **unsupervised learning**

Learning process in which changes in a network's weights and biases are not due to the intervention of any external teacher. Commonly changes are a function of the current network input vectors, output vectors, and previous weights and biases.

## **update**

Make a change in weights and biases. The update can occur after presentation of a single input vector or after accumulating changes over several input vectors.

## **validation vectors**

Set of input vectors (not used directly in

training) that is used to monitor training progress so as to keep the network from overfitting.

## **weight function**

Weight functions apply weights to an input to get weighted inputs, as specified by a particular function.

## **weight matrix**

Matrix containing connection strengths from a layer's inputs to its neurons. The element  $w_{i,j}$  of a weight matrix  $W$  refers to the connection strength from input  $j$  to neuron  $i$ .

## **weighted input vector**

Result of applying a weight to a layer's input, whether it is a network input or the output of another layer.

## **Widrow-Hoff learning rule**

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

## **Chapter 9**

# **BIBLIOGRAPHY**

---

# 9.1 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*, Vol. 18, No. 1, January 2007, pp. 14 -27.

This paper provides detailed algorithms for the calculation of gradients and Jacobians for arbitrarily-connected neural networks. Both the backpropagation-through-time and real-time recurrent learning algorithms are covered.

[DeSc83] Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

**[DHH01]** De Jesús, O., J.M. Horn, and M.T. Hagan, "Analysis of Recurrent Network Training and Suggestions for Improvements," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2632–2637.

**[Elma90]** Elman, J.L., "Finding structure in time," *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

This paper is a superb introduction to the Elman networks described in Chapter 10, "Recurrent Networks."

**[FeTs03]** Feng, J., C.K. Tse, and F.C.M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 50, No. 7, 2003, pp. 954–957.

[F1Re64] Fletcher, R., and C.M. Reeves, "Function minimization by conjugate gradients," *Computer Journal*, Vol. 7, 1964, pp. 149–154.

[FoHa97] Foressee, F.D., and M.T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997, pp. 1930–1935.

[GiMu81] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

[GiPr02] Gianluca, P., D. Przybylski, B. Rost, P. Baldi, "Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles," *Proteins: Structure, Function, and Genetics*, Vol. 47, No. 2, 2002, pp. 228–235.

[Gros82] Grossberg, S., *Studies of the Mind*

*and Brain*, Dordrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan, M.T., and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642–1656.

[HaJe99] Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311–340.

[HaMe94] Hagan, M.T., and M. Menhaj, "Training feed-forward networks with the Marquardt algorithm," *IEEE Transactions on*

*Neural Networks*, Vol. 5, No. 6, 1999, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

**[HaRu78]** Harrison, D., and Rubinfeld, D.L., "Hedonic prices and the demand for clean air," *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81-102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

**[HDB96]** Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has example programs, an instructor's guide, and transparency overheads for teaching.

[**HDH09**] Horn, J.M., O. De Jesús and M.T. Hagan, "Spurious Valleys in the Error Surface of Recurrent Networks - Analysis and Avoidance," IEEE Transactions on Neural Networks, Vol. 20, No. 4, pp. 686-700, April 2009.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also explains how training algorithms can be modified to avoid becoming stuck in these valleys.

[**Hebb49**] Hebb, D.O., *The Organization of*

*Behavior*, New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

**[Himm72]** Himmelblau, D.M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

**[HuSb92]** Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, "Neural Networks for Control System — A Survey," *Automatica*, Vol. 28, 1992, pp. 1083–1112.

**[JaRa04]** Jayadeva and S.A.Rahman, "A neural network with  $O(N)$  neurons for ranking  $N$  numbers in  $O(1/N)$  time," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 51, No. 10, 2004, pp. 2044–2051.

**[Joll86]** Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

**[KaGr96]** Kamwa, I., R. Grondin, V.K. Sood, C. Gagnon, Van Thich Nguyen, and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *IEEE Transactions on Instrumentation and Measurement*, Vol. 45, No. 2, 1996, pp. 657–664.

**[Koho87]** Kohonen, T., *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

**[Koho97]** Kohonen, T., *Self-Organizing Maps, Second Edition*, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications, and hardware of self-

organizing maps. It also includes a comprehensive literature survey.

[LiMi89] Li, J., A.N. Michel, and W. Porod, "Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube," *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, 1989, pp. 1405–1422.

This paper discusses a class of neural networks described by first-order linear differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li, et. al., is implemented in Advanced Topics in the *User's Guide*.

**[Lipp87]** Lippman, R.P., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, 1987, pp. 4–22.

This paper gives an introduction to the field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

**[MacK92]** MacKay, D.J.C., "Bayesian interpolation," *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415–447.

**[Marq63]** Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431–441.

**[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.



