# MATHEMATICA®

# NEURAL NETWORKS

TRAIN AND ANALYZE NEURAL NETWORKS TO FIT YOUR DATA

WOLFRAMRESEARCH
www.wolfram.com

September 2005
First edition
Intended for use with *Mathematica* 5

Software and manual written by: Jonas Sjöberg

Product managers: Yezabel Dooley and Kristin Kummer
Project managers: Julienne Davison and Jennifer Peterson
Editors: Rebecca Bigelow and Jan Progen
Proofreader: Sam Daniel
Software quality assurance: Jay Hawkins, Cindie Strater, Angela Thelen, and Rachelle Bergmann
Package design by: Larry Adelston, Megan Gillette, Richard Miske, and Kara Wilson

Special thanks to the many alpha and beta testers and the people at Wolfram Research who gave me valuable input and feedback during the development of this package. In particular, I would like to thank Rachelle Bergmann and Julia Guelfi at Wolfram Research and Sam Daniel, a technical staff member at Motorola's Integrated Solutions Division, who gave thousands of suggestions on the software and the documentation.

Published by Wolfram Research, Inc., 100 Trade Center Drive, Champaign, Illinois 61820-7237, USA
phone: +1-217-398-0700; fax: +1-217-398-0747; email: info@wolfram.com; web: www.wolfram.com

**Wolfram Research, Inc. makes no representations, express or implied, with respect to this Product, including without limitations, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which Wolfram Research, Inc. is willing to license the Product is a provision that Wolfram Research, Inc. and its distribution licensees, distributors, and dealers shall in no event be liable for any indirect, incidental, or consequential damages, and that liability for direct damages shall be limited to the amount of the purchase price paid for the Product.**

**In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. Wolfram Research, Inc. shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this document or the package software it describes, whether or not they are aware of the errors or omissions. Wolfram Research, Inc. does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury, or significant loss.**

T4055 267204 0905.rcm

# Table of Contents

# 1 Introduction

*Neural Networks* is a *Mathematica* package designed to train, visualize, and validate neural network models. A neural network model is a structure that can be adjusted to produce a mapping from a given set of data to features of or relationships among the data. The model is adjusted, or trained, using a collection of data from a given source as input, typically referred to as the training set. After successful training, the neural network will be able to perform classification, estimation, prediction, or simulation on new data from the same or similar sources. The *Neural Networks* package supports different types of training or learning algorithms.

More specifically, the *Neural Networks* package uses numerical data to specify and evaluate artificial neural network models. Given a set of data, $\{x_i, y_i\}_{i=1}^N$ from an unknown function, $y = f(x)$, this package uses numerical algorithms to derive reasonable estimates of the function, $f(x)$. This involves three basic steps: First, a neural network structure is chosen that is considered suitable for the type of data and underlying process to be modeled. Second, the neural network is trained by using a sufficiently representative set of data. Third, the trained network is tested with different data, from the same or related sources, to validate that the mapping is of acceptable quality.

The package contains many of the standard neural network structures and related learning algorithms. It also includes some special functions needed to address a number of typical problems, such as classification and clustering, time series and dynamic systems, and function estimation problems. In addition, special performance evaluation functions are included to validate and illustrate the quality of the desired mapping.

The documentation contains a number of examples that demonstrate the use of the different neural network models. You can solve many problems simply by applying the example commands to your own data.

Most functions in the *Neural Networks* package support a number of different options that you can use to modify the algorithms. However, the default values have been chosen so as to give good results for a large variety of problems, allowing you to get started quickly using only a few commands. As you gain experience, you will be able to customize the algorithms by changing the options.

Choosing the proper type of neural network for a certain problem can be a critical issue. The package contains many examples illustrating the possible uses of the different neural network types. Studying these examples will help you choose the network type suited to the situation.

Solved problems, illustrations, and other facilities available in the *Neural Networks* package should enable the interested reader to tackle many problems after reviewing corresponding parts of the guide. However, this guide does not contain an exhaustive introduction to neural networks. Although an attempt was made to illustrate the possibilities and limitations of neural network methods in various application areas, this guide is by no means a substitute for standard textbooks, such as those listed in the references at the end of most chapters. Also, while this guide contains a number of examples in which *Mathematica* functions are used with *Neural Networks* commands, it is definitely not an introduction to *Mathematica* itself. The reader is advised to consult the standard *Mathematica* reference: Wolfram, Stephen, *The Mathematica Book*, 5th ed. (Wolfram Media, 2003).

## 1.1 Features of This Package

The following table lists the neural network types supported by the *Neural Networks* package along with their typical usage. Chapter 2, Neural Network Theory—A Short Tutorial, gives brief explanations of the different neural network types.

| *Network type* | *Typical use (s) of the network* |
|---|---|
| Radial basis function | function approximation, classification, dynamic systems modeling |
| Feedforward | function approximation, classification, dynamic systems modeling |
| Dynamic | dynamic systems modeling, time series |
| Hopfield | classification, auto-associative memory |
| Perceptron | classification |
| Vector quantization | classification |
| Unsupervised | clustering, self-organizing maps, Kohonen networks |

Neural network types supported by the *Neural Networks* package.

The functions in the package are constructed so that only the minimum amount of information has to be specified by the user. For example, the number of inputs and outputs of a network are automatically extracted from the dimensionality of the data so they do not need to be entered explicitly.

Trained networks are contained in special objects with a head that identifies the type of network. You do not have to keep track of all of the parameters and other information contained in a neural network model; everything is contained in the network object. Extracting or changing parts of the neural network information can be done by addressing the appropriate part of the object.

Intermediate information is logged during the training of a network and returned in a special training record at the end of the training. This record can be used to analyze the training performance and to access parameter values at intermediate training stages.

The structure of *feedforward* and *radial basis function* neural network types can be modified to customize the network for your specific problem. For example, the neuron activation function can be changed to some other suitable function. You can also set some of the network parameters to predefined values and exclude them from the training.

A neural network model can be customized when the unknown function is known to have a special structure. For example, in many situations the unknown function is recognized as more nonlinear in some inputs than in others. The *Neural Networks* package allows you to define a model that is linear with respect to some of the inputs and nonlinear with respect to other inputs. After the neural network structure has been defined, you can proceed with the network's training as you would with a network that does not have a defined structure.

The *Neural Networks* package contains special initialization algorithms for the network parameters, or weights, that start the training with reasonably good performance. After this initialization, an iterative training algorithm is applied to the network and the parameter set is optimized. The special initialization makes the training much faster than a completely random choice for the parameters. This also alleviates difficulties encountered in problems with multiple local minima.

For *feedforward*, *radial basis function*, and *dynamic neural networks,* the weights are adjusted iteratively using gradient-based methods. The Levenberg-Marquardt algorithm is used by default, because it is considered to be the best choice for most problems. Another feature in favor of this algorithm is that it can take advantage of a situation where a network is linear in some of its parameters. Making use of the separability of the linear and nonlinear parts of the underlying minimization problem will speed up training considerably.

For large data sets and large neural network models, the training algorithms for some types of neural networks will become computation intensive. This package reduces the computation load in two ways: (1) the expressions are optimized before numerical evaluation, thus minimizing the number of operations, and (2) the computation-intensive functions use the `Compile` command to send compiled code to *Mathematica*. Because compiled code can only work with machine-precision numbers, numerical precision will be somewhat restricted. In most practical applications this limitation will be of little significance. If you would prefer noncompiled evaluation, you could set the compiled option to false, `Compiled → False`.

# 2 Neural Network Theory—A Short Tutorial

Starting with measured data from some known or unknown source, a neural network may be trained to perform classification, estimation, simulation, and prediction of the underlying process generating the data. Therefore, neural networks, or neural nets, are software tools designed to estimate relationships in data. An estimated relationship is essentially a mapping, or a function, relating raw data to its features. The *Neural Networks* package supports several function estimation techniques that may be described in terms of different types of neural networks and associated learning algorithms.

The general area of artificial neural networks has its roots in our understanding of the human brain. In this regard, initial concepts were based on attempts to mimic the brain's way of processing information. Efforts that followed gave rise to various models of biological neural network structures and learning algorithms. This is in contrast to the computational models found in this package, which are only concerned with artificial neural networks as a tool for solving different types of problems where unknown relationships are sought among given data. Still, much of the nomenclature in the neural network arena has its origins in biological neural networks, and thus, the original terminology will be used alongside with more traditional nomenclature from statistics and engineering.

## 2.1 Introduction to Neural Networks

In the context of this package, a neural network is nothing more than a function with adjustable or tunable parameters. Let the input to a neural network be denoted by $x$, a real-valued (row) vector of arbitrary dimensionality or length. As such, $x$ is typically referred to as *input*, *input vector*, *regressor*, or sometimes, *pattern vector*. Typically, the length of vector $x$ is said to be the *number of inputs* to the network. Let the network output be denoted by $\hat{y}$, an approximation of the desired output $y$, also a real-valued vector having one or more components, and the *number of outputs* from the network. Often data sets contain many input-output pairs. Thus $x$ and $y$ denote matrices with one input and one output vector on each row.

Generally, a neural network is a structure involving weighted interconnections among *neurons*, or *units*, which are most often nonlinear scalar transformations, but they can also be linear. Figure 2.1 shows an example of a one-hidden-layer neural network with three inputs, $x = \{x_1, x_2, x_3\}$ that, along with a *unity bias* input, feed each of the two neurons comprising the *hidden layer*. The two outputs from this layer and a unity bias are then fed into the single output layer neuron, yielding the scalar output, $\hat{y}$. The layer of neurons is called hidden because its outputs are not directly seen in the data. This particular type of neural network is described in detail in Section 2.5, Feedforward and Radial Basis Function Networks. Here, this network will be used to explain common notation and nomenclature used in the package.

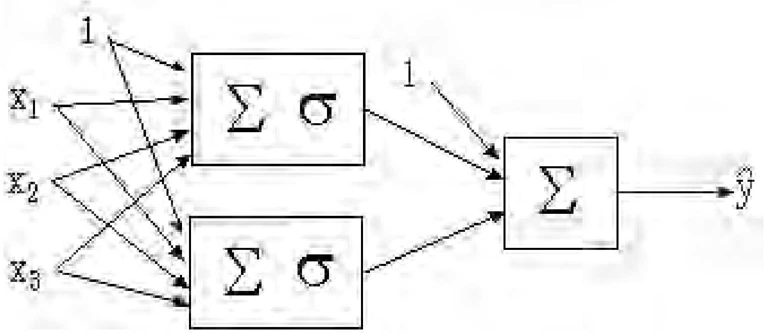


Figure 2.1. A feedforward neural network with three inputs, two hidden neurons, and one output neuron.

Each arrow in Figure 2.1 corresponds to a real-valued *parameter*, or a weight, of the network. The values of these parameters are tuned in the network training.

Generally, a neuron is structured to process multiple inputs, including the unity bias, in a nonlinear way, producing a single output. Specifically, all inputs to a neuron are first augmented by multiplicative weights. These weighted inputs are summed and then transformed via a nonlinear *activation function*, $\sigma$. As indicated in Figure 2.1, the neurons in the first layer of the network are nonlinear. The single output neuron is linear, since no activation function is used.

By inspection of Figure 2.1, the output of the network is given by

$$\begin{aligned}\hat{y} &= b^2 + \sum_{i=1}^{2} w_i^2\, \sigma\left(b_i^1 + \sum_{j=1}^{3} w_{i,j}^1\, x_j\right) \\ &= w_1^2\, \sigma\,(w_{1,1}^1\, x_1 + w_{1,2}^1\, x_2 + w_{1,3}^1\, x_3 + b_1^1) + \\ &\quad w_2^2\, \sigma\,(w_{2,1}^1\, x_1 + w_{2,2}^1\, x_2 + w_{2,3}^1\, x_3 + b_2^1) + b^2\end{aligned} \tag{1}$$

involving the various parameters of the network, the weights $\{w_{i,j}^1, b_{i,j}^1, w_i^2, b^2\}$. The weights are sometimes referred to as *synaptic strengths*.

Equation 2.1 is a nonlinear mapping, $x \to \hat{y}$, specifically representing the neural network in Figure 2.1. In general, this mapping is given in more compact form by

$$\hat{y} = g\,(\theta, x) \tag{2}$$

where the $\theta$ is a real-valued vector whose components are the parameters of the network, namely, the weights. When algorithmic aspects, independent of the exact structure of the neural network, are discussed, then this compact form becomes more convenient to use than an explicit one, such as that of Equation 2.1.

This package supports several types of neural networks from which a user can choose. Upon assigning design parameters to a chosen network, thus specifying its structure $g(\cdot,\cdot)$, the user can begin to train it. The goal of training is to find values of the parameters $\theta$ so that, for any input $x$, the network output $\hat{y}$ is a good approximation of the desired output $y$. Training is carried out via suitable algorithms that tune the parameters $\theta$ so that input training data map well to corresponding desired outputs. These algorithms are iterative in nature, starting at some initial value for the parameter vector $\theta$ and incrementally updating it to improve the performance of the network.

Before the trained network is accepted, it should be validated. Roughly, this means running a number of tests to determine whether the network model meets certain requirements. Probably the simplest way, and often the best, is to test the neural network on a data set that was not used for training, but which was generated under similar conditions. Trained neural networks often fail this validation test, in which case the user will have to choose a better model. Sometimes, however, it might be enough to just repeat the training, starting from different initial parameters $\theta$. Once the neural network is validated, it is ready to be used on new data.

The general purpose of the *Neural Networks* package can be described as function approximation. However, depending on the origin of the data, and the intended use of the obtained neural network model, the function approximation problem may be subdivided into several types of problems. Different types of function approximation problems are described in Section 2.1.1. Section 1.1, Features of This Package, includes a table giving an overview of the supported neural networks and the particular types of problems they are intended to address.

### 2.1.1 Function Approximation

When input data originates from a function with real-valued outputs over a continuous range, the neural network is said to perform a traditional function approximation. An example of an approximation problem could be one where the temperature of an object is to be determined from secondary measurements, such as emission of radiation. Another more trivial example could be to estimate shoe size based on a person's height. These two examples involve models with one input and one output. A more advanced model of the second example might use gender as a second input in order to derive a more accurate estimate of the shoe size.

Pure functions may be approximated with the following two network types:

- Feedforward Neural Networks
- Radial Basis Function Networks

and a basic example can be found in Section 3.4.2, Function Approximation Example.

### 2.1.2 Time Series and Dynamic Systems

A special type of function approximation problem is one where the input data is time dependent. This means that the function at hand has "memory", is thus dynamic, and is referred to as a *dynamic system*. For such systems, past information can be used to predict its future behavior. Two examples of dynamic system problems are: (1) predicting the price of a state bond or that of some other financial instrument; and (2) describing the speed of an engine as a function of the applied voltage and load.

In both of these examples the output signal at some time instant depends on what has happened earlier. The first example is a *time-series* problem modeled as a system involving no inputs. In the second example there are two inputs: the applied voltage and the load. Examples of these kinds can be found in Section 8.2.2, Identifying the Dynamics of a DC Motor, and in Section 12.2, Prediction of Currency Exchange Rate.

The process of finding a model of a system from observed inputs and outputs is generally known as *system identification.* The special case involving time series is more commonly known as *time-series analysis*. This is an applied science field that employs many different models and methods. The *Neural Network* package supports both linear and nonlinear models and methods in the form of neural network structures and associated learning algorithms.

A neural network models a dynamic system by employing memory in its inputs; specifically, storing a number of past input and output data. Such neural network structures are often referred to as *tapped-delay-line neural networks*, or *NFIR*, *NARX*, and *NAR* models.

Dynamic neural networks can be either feedforward in structure or employ radial basis functions, and they must accommodate memory for past information. This is further described in Section 2.6, Dynamic Neural Networks.

The *Neural Networks* package contains many useful *Mathematica* functions for working with dynamic neural networks. These built-in functions facilitate the training and use of the dynamic neural networks for prediction and simulation.

### 2.1.3 Classification and Clustering

In the context of neural networks, classification involves deriving a function that will separate data into categories, or classes, characterized by a distinct set of features. This function is mechanized by a so-called *network classifier*, which is trained using data from the different classes as inputs, and vectors indicating the true class as outputs.

A network classifier typically maps a given input vector to one of a number of classes represented by an equal number of outputs, by producing 1 at the output class and 0 elsewhere. However, the outputs are not always binary (0 or 1); sometimes they may range over {0, 1}, indicating the degrees of participation of a given input over the output classes. The *Neural Networks* package contains some functions especially suited for this kind of constrained approximation.

The following types of neural networks are available for solving classification problems:

- Perceptron
- Vector Quantization Networks
- Feedforward Neural Networks
- Radial Basis Function Networks
- Hopfield Networks

A basic classification example can be found in Section 3.4.1, Classification Problem Example.

When the desired outputs are not specified, a neural network can only operate on input data. As such, the neural network cannot be trained to produce a desired output in a supervised way, but must instead look for hidden structures in the input data without supervision, employing so-called *self-organizing*. Structures in data manifest themselves as constellations of clusters that imply levels of correlation among the raw data and a consequent reduction in dimensionality and increased information in coding efficiency. Specifically, a particular input data vector that falls within a given cluster could be represented by its unique centroid within some squared error. As such, unsupervised networks may be viewed as classifiers, where the classes are the discovered clusters.

An unsupervised network can also employ a *neighbor* feature so that "proximity" among clusters may be preserved in the clustering process. Such networks, known as s*elf-organizing maps* or *Kohonen networks*, may be interpreted loosely as being nonlinear projections of the original data onto a one- or two-dimensional space.

Unsupervised networks and self-organizing maps are described in some detail in Section 2.8, Unsupervised and Vector Quantization Networks.

## 2.2 Data Preprocessing

The *Neural Networks* package offers several algorithms to build models using data. Before applying any of the built-in functions for training, it is important to check that the data is “reasonable.” Naturally, you cannot expect to obtain good models from poor or insufficient data. Unfortunately, there is no standard procedure that can be used to test the quality of the data. Depending on the problem, there might be special features in the data that may be used in testing data quality. Toward this end, some general advice follows.

One way to check for quality is to view graphical representations of the data in question, in the hope of selecting a reasonable subset while eliminating problematic parts. For this purpose, you can use any suitable *Mathematica* plotting function or employ other such functions that come with the *Neural Networks* package especially designed to visualize the data in classification, time series, and dynamic system problems.

In examining the data for a classification problem, some reasonable questions to ask may include the following:

- Are all classes equally represented by the data?

- Are there any outliers, that is, data samples dissimilar from the rest?

For time-dependent data, the following questions might be considered:

- Are there any outliers, that is, data samples very different from neighboring values?

- Does the input signal of the dynamic system lie within the interesting amplitude range?

- Does the input signal of the dynamic system excite the interesting frequency range?

Answers to these questions might reveal potential difficulties in using the given data for training. If so, new data may be needed.

Even if they appear to be quite reasonable, it might be a good idea to consider preprocessing the data before initiating training. Preprocessing is a transformation, or conditioning, of data designed to make modeling easier and more robust. For example, a known nonlinearity in some given data could be removed by an appropriate transformation, producing data that conforms to a linear model that is easier to work with.

Similarly, removing detected trends and outliers in the data will improve the accuracy of the model. Therefore, before training a neural network, you should consider the possibility of transforming the data in some useful way.

You should always make sure that the range of the data is neither too small nor too large so that you stay well within the machine precision of your computer. If this is not possible, you should scale the data. Although *Mathematica* can work with arbitrary accuracy, you gain substantial computational speed if you stay within machine precision. The reason for this is that the *Neural Networks* package achieves substantial computational speed-up using the `Compile` command, which limits subsequent computation to the precision of the machine.

It is also advisable to scale the data so that the different input signals have approximately the same numerical range. This is not necessary for feedforward and Hopfield networks, but is recommended for all other network models. The reason for this is that the other network models rely on Euclidean measures, so that unscaled data could bias or interfere with the training process. Scaling the data so that all inputs have the same range often speeds up the training and improves resulting performance of the derived model.

It is also a good idea to divide the data set into training data and validation data. The validation data should not be used in the training but, instead, be reserved for the quality check of the obtained network.

You may use any of the available *Mathematica* commands to perform the data preprocessing before applying neural network algorithms; therefore, you may consult the standard *Mathematica* reference: Wolfram, Stephen, *The Mathematica Book*, 5th ed. (Wolfram Media, 2003). Some interesting starting points might be Section 1.6.6 Manipulating Numerical Data, Section 1.6.7 Statistics, and Section 1.8.3, Vectors and Matrices, as well as the standard *Mathematica* add-on packages `Statistics`DataManipulation`` and `LinearAlgebra`MatrixManipulation``.

## 2.3 Linear Models

A general modeling principle is to "try simple things first." The idea behind this principle is that there is no reason to make a model more complex than necessary. The simplest type of model is often a linear model. Figure 2.2 illustrates a linear model. Each arrow in the figure symbolizes a parameter in the model.

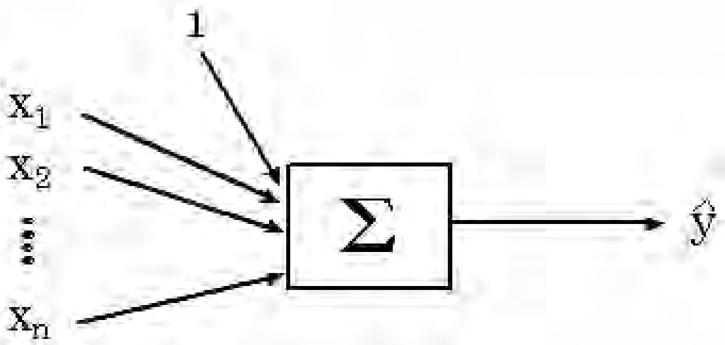


Figure 2.2. A linear model.

Mathematically, the linear model gives rise to the following simple equation for the output

$$\hat{y} = w_1\, x_1 + w_2\, x_2 + \ldots + w_n\, x_n + b \qquad (3)$$

Linear models are called *regression models* in traditional statistics. In this case the output $\hat{y}$ is said to regress on the inputs $x_1,\ldots,x_n$ plus a bias parameter $b$.

Using the *Neural Networks* package, the linear model in Figure 2.2 can be obtained as a feedforward network with one linear output neuron. Section 5.1.1, InitializeFeedForwardNet describes how this is done.

A linear model may have several outputs. Such a model can be described as a network consisting of a bank of linear neurons, as illustrated in Figure 2.3.

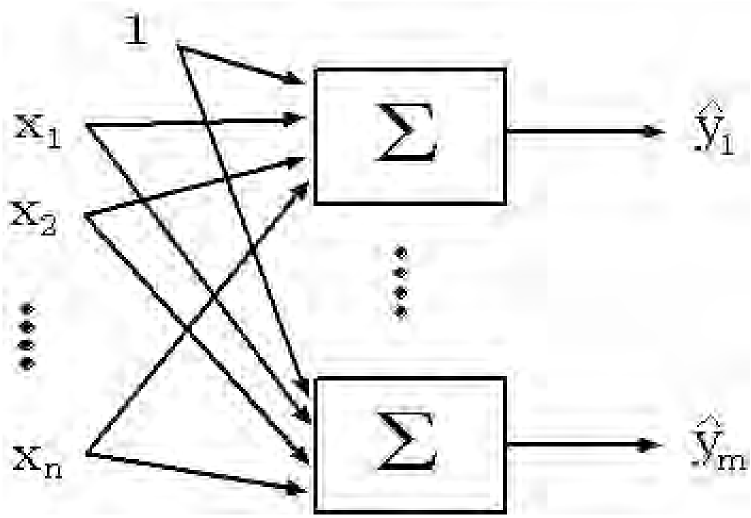


Figure 2.3. A multi-output linear model.

## 2.4 The Perceptron

After the linear networks, the *perceptron* is the simplest type of neural network and it is typically used for classification. In the one-output case it consists of a neuron with a step function. Figure 2.4 is a graphical illustration of a perceptron with inputs $x_1, ..., x_n$ and output $\hat{y}$.

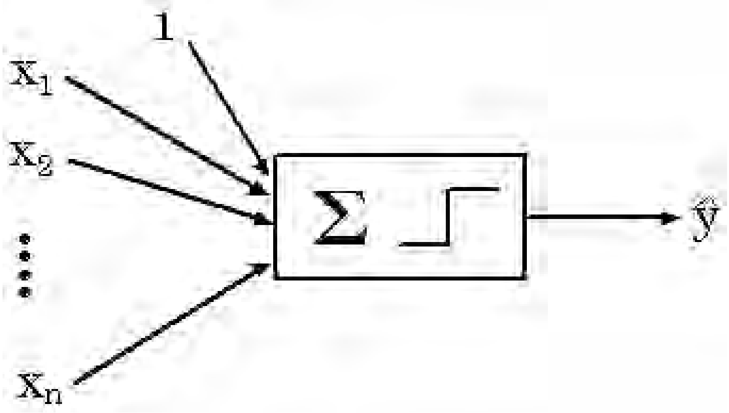


Figure 2.4. A perceptron classifier.

As indicated, the weighted sum of the inputs and the unity bias are first summed and then processed by a step function to yield the output

$$\hat{y}(x, w, b) = \text{UnitStep}[w_1 x_1 + w_2 x_2 + ... + w_n x_n + b] \qquad (4)$$

where $\{w_1, ..., w_n\}$ are the weights applied to the input vector and $b$ is the *bias weight*. Each weight is indicated with an arrow in Figure 2.4. Also, the `UnitStep` function is 0 for arguments less than 0 and 1 elsewhere. So, the output $\hat{y}$ can take values of 0 or 1, depending on the value of the weighted sum. Consequently, the perceptron can indicate two classes corresponding to these two output values. In the training process, the weights (input and bias) are adjusted so that input data is mapped correctly to one of the two classes. An example can be found in Section 4.2.1, Two Classes in Two Dimensions.

The perceptron can be trained to solve any two-class classification problem where the classes are *linearly separable.* In two-dimensional problems (where $x$ is a two-component row vector), the classes may be separated by a straight line, and in higher-dimensional problems, it means that the classes are separable by a hyperplane.

If the classification problem is not linearly separable, then it is impossible to obtain a perceptron that correctly classifies all training data. If some misclassifications can be accepted, then a perceptron could still constitute a good classifier.

Because of its simplicity, the perceptron is often inadequate as a model for many problems. Nevertheless, many classification problems have simple solutions for which it may apply. Also, important insights may be gained from using the perceptron, which may shed some light when considering more complicated neural network models.

Perceptron classifiers are trained with a supervised training algorithm. This presupposes that the true classes of the training data are available and incorporated in the training process. More specifically, as individual inputs are presented to the perceptron, its weights are adjusted iteratively by the training algorithm so as to produce the correct class mapping at the output. This training process continues until the perceptron correctly classifies all the training data or when a maximum number of iterations has been reached. It is possible to choose a judicious initialization of the weight values, which in many cases makes the iterative learning unnecessary. This is described in Section 4.1.1, InitializePerceptron.

Classification problems involving a number of classes greater than two can be handled by a multi-output perceptron that is defined as a number of perceptrons in parallel. It contains one perceptron, as shown in Figure 2.4, for each output, and each output corresponds to a class.

The training process of such a multi-output perceptron structure attempts to map each input of the training data to the correct class by iteratively adjusting the weights to produce 1 at the output of the corresponding perceptron and 0 at the outputs of all the remaining outputs. However, it is quite possible that a number of input vectors may map to multiple classes, indicating that these vectors could belong to several classes. Such cases may require special handling. It may also be that the perceptron classifier cannot make a decision for a subset of input vectors because of the nature of the data or insufficient complexity of the network structure itself. An example with several classes can be found in Section 4.2.2, Several Classes in Two Dimensions.

### Training Algorithm

The training of a one-output perceptron will be described in the following section. In the case of a multi-output perceptron, each of the outputs may be described similarly.

A perceptron is defined parametrically by its weights $\{w, b\}$, where $w$ is a column vector of length equal to the dimension of the input vector $x$ and $b$ is a scalar. Given the input, a row vector, $x = \{x_1, \ldots, x_n\}$, the output of a perceptron is described in compact form by

$$\hat{y}(x, w, b) = \mathrm{UnitStep}[x\, w + b] \tag{5}$$

This description can also be used when a set of input vectors is considered. Let $x$ be a matrix with one input vector in each row. Then $\hat{y}$ in Equation 2.5 becomes a column vector with the corresponding output in its rows.

The weights $\{w, b\}$ are obtained by iteratively training the perceptron with a known data set containing input-output pairs, one input vector in each row of a matrix $x$, and one output in each row of a matrix $y$, as described in Section 3.2.1, Data Format. Given $N$ such pairs in the data set, the training algorithm is defined by

$$\begin{aligned} w_{i+1} &= w_i + \eta\, x^T \epsilon_i \\ b_{i+1} &= b_i + \eta \sum_{j=1}^{N} \epsilon_i[[j]] \end{aligned} \tag{6}$$

where $i$ is the iteration number, $\eta$ is a scalar step size, and $\epsilon_i = y - \hat{y}(x, w_i, b_i)$ is a column vector with $N$-components of classification errors corresponding to the $N$ data samples of the training set. The components of the error vector can only take three values, namely, 0, 1, and –1. At any iteration $i$, values of 0 indicate that the corresponding data samples have been classified correctly, while all the others have been classified incorrectly.

The training algorithm Equation 2.5 begins with initial values for the weights $\{w, b\}$ and $i = 0$, and iteratively updates these weights until all data samples have been classified correctly or the iteration number has reached a maximum value, $i_{max}$.

The step size $\eta$, or *learning rate* as it is often called, has the following default value

$$\eta = \frac{(\mathtt{Max[x]} - \mathtt{Min[x]})}{\mathtt{N}} \tag{7}$$

By compensating for the range of the input data, $x$, and for the number of data samples, $N$, this default value of $\eta$ should be good for many classification problems independent of the number of data samples and their numerical range. It is also possible to use a step size of choice rather than using the default value. However, although larger values of $\eta$ might accelerate the training process, they may induce oscillations that may slow down the convergence.

## 2.5 Feedforward and Radial Basis Function Networks

This section describes feedforward and radial basis function networks, both of which are useful for function approximation. Mathematically, these networks may be viewed as parametric functions, and their training constituting a parameter estimation or fitting process. The *Neural Networks* package provides a common built-in function, `NeuralFit`, for training these networks.

### 2.5.1 Feedforward Neural Networks

Feedforward neural networks (FF networks) are the most popular and most widely used models in many practical applications. They are known by many different names, including “multi-layer perceptrons.”

Figure 2.5 illustrates a one-hidden-layer FF network with inputs $x_1, \ldots, x_n$ and output $\hat{y}$. Each arrow in the figure symbolizes a parameter in the network. The network is divided into *layers*. The input layer consists of just the inputs to the network. Then follows a *hidden layer*, which consists of any number of *neurons*, or *hidden units* placed in parallel. Each neuron performs a weighted summation of the inputs, which then passes a nonlinear *activation function* $\sigma$, also called the *neuron* function.

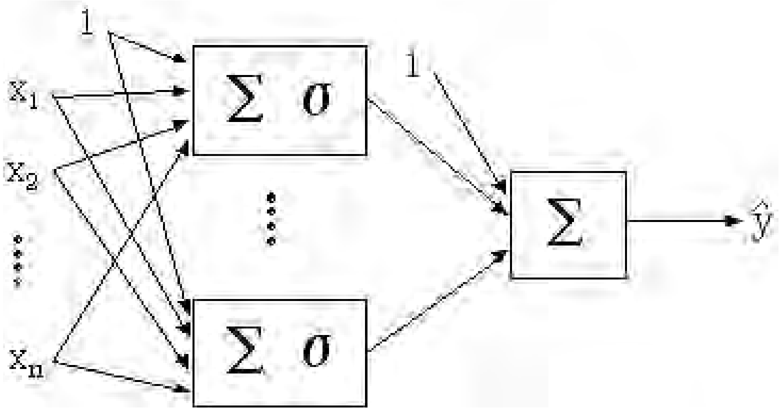


Figure 2.5. A feedforward network with one hidden layer and one output.

Mathematically the functionality of a hidden neuron is described by

$$\sigma\left(\sum_{j=1}^{n} w_j\, x_j + b_j\right)$$

where the weights $\{w_j, b_j\}$ are symbolized with the arrows feeding into the neuron.

The network output is formed by another weighted summation of the outputs of the neurons in the hidden layer. This summation on the output is called the *output layer*. In Figure 2.5 there is only one output in the output layer since it is a single-output problem. Generally, the number of output neurons equals the number of outputs of the approximation problem.

The neurons in the hidden layer of the network in Figure 2.5 are similar in structure to those of the perceptron, with the exception that their activation functions can be any differential function. The output of this network is given by

$$\hat{y}(\theta) = g(\theta, x) = \sum_{i=1}^{nh} w_i^2\, \sigma\left(\sum_{j=1}^{n} w_{i,j}^1\, x_j + b_{j,i}^1\right) + b^2 \qquad (8)$$

where $n$ is the number of inputs and $nh$ is the number of neurons in the hidden layer. The variables $\{w_{i,j}^1, b_{j,i}^1, w_i^2, b^2\}$ are the parameters of the network model that are represented collectively by the parameter vector $\theta$. In general, the neural network model will be represented by the compact notation $g(\theta, x)$ whenever the exact structure of the neural network is not necessary in the context of a discussion.

Some small function approximation examples using an FF network can be found in Section 5.2, Examples.

Note that the size of the input and output layers are defined by the number of inputs and outputs of the network and, therefore, only the number of hidden neurons has to be specified when the network is defined. The network in Figure 2.5 is sometimes referred to as a three-layer network, with input, hidden, and output layers. However, since no processing takes place in the input layer, it is also sometimes called a two-layer network. To avoid confusion this network is called a *one-hidden-layer FF network* throughout this documentation.

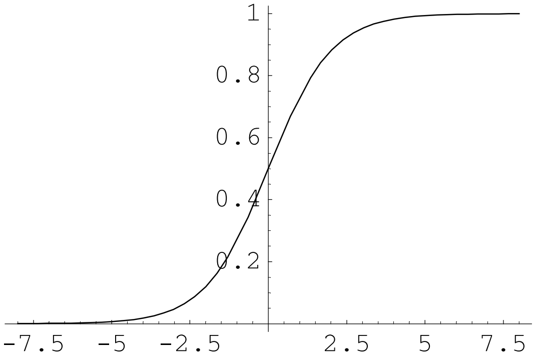
In training the network, its parameters are adjusted incrementally until the training data satisfy the desired mapping as well as possible; that is, until $\hat{y}(\theta)$ matches the desired output $y$ as closely as possible up to a maximum number of iterations. The training process is described in Section 2.5.3, Training Feedforward and Radial Basis Function Networks.

The nonlinear activation function in the neuron is usually chosen to be a smooth step function. The default is the standard sigmoid

$$\mathtt{Sigmoid}[x] = \frac{1}{1+e^{-x}} \tag{9}$$

that looks like this.

```
In[1]:= << NeuralNetworks`
        Plot[Sigmoid[x], {x, -8, 8}]
```



The FF network in Figure 2.5 is just one possible architecture of an FF network. You can modify the architecture in various ways by changing the options. For example, you can change the activation function to any differentiable function you want. This is illustrated in Section 13.3.2, The Neuron Function in a Feedforward Network.

### Multilayer Networks

The package supports FF neural networks with any number of hidden layers and any number of neurons (hidden neurons) in each layer. In Figure 2.6 a multi-output FF network with two hidden layers is shown.

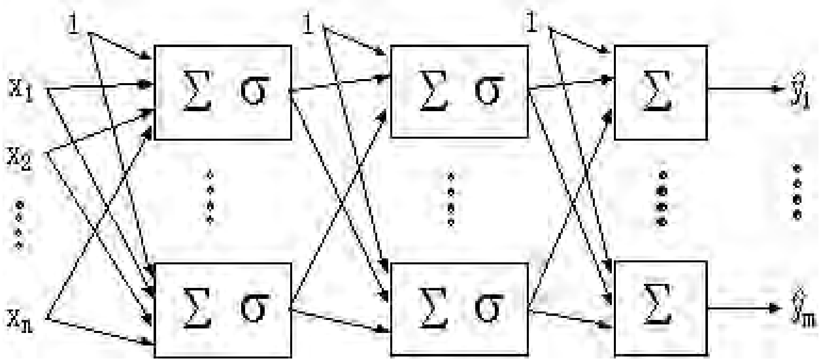


Figure 2.6. A multi-output feedforward network with two hidden layers.

The number of layers and the number of hidden neurons in each hidden layer are user design parameters. The general rule is to choose these design parameters so that the best possible model, with as few parameters as possible, is obtained. This is, of course, not a very useful rule, and in practice you have to experiment with different designs and compare the results, to find the most suitable neural network model for the problem at hand.

For many practical applications, one or two hidden layers will suffice. The recommendation is to start with a linear model; that is, neural networks with no hidden layers, and then go over to networks with one hidden layer but with no more than five to ten neurons. As a last step you should try two hidden layers.

The output neurons in the FF networks in Figures 2.5 and 2.6 are linear; that is, they do not have any nonlinear activation function after the weighted sum. This is normally the best choice if you have a general function, a time series or a dynamical system that you want to model. However, if you are using the FF network for classification, then it is generally advantageous to use nonlinear output neurons. You can do this by using the option `OutputNonlinearity` when using the built-in functions provided with the *Neural Networks* package, as illustrated in the examples offered in Section 5.3, Classification with Feedforward Networks, and Section 12.1, Classification of Paper Quality.

### 2.5.2 Radial Basis Function Networks

After the FF networks, the radial basis function (RBF) network comprises one of the most used network models.

Figure 2.7 illustrates an RBF network with inputs $x_1$, …, $x_n$ and output $\hat{y}$. The arrows in the figure symbolize parameters in the network. The RBF network consists of one hidden layer of basis functions, or neurons. At the input of each neuron, the distance between the neuron center and the input vector is calculated. The output of the neuron is then formed by applying the basis function to this distance. The RBF network output is formed by a weighted sum of the neuron outputs and the unity bias shown.

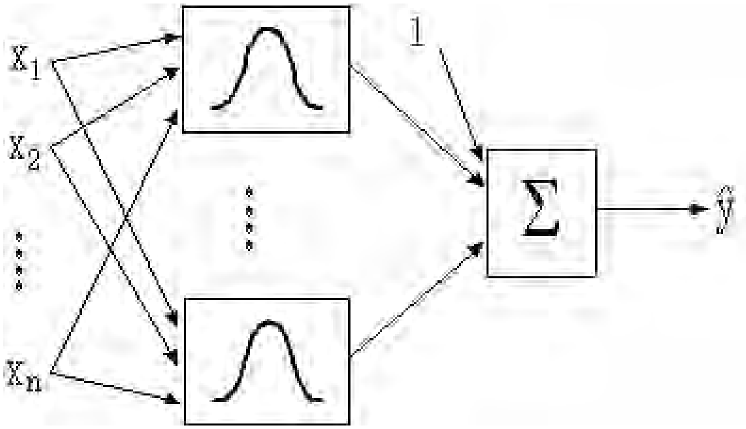


Figure 2.7. An RBF network with one output.

The RBF network in Figure 2.7 is often complemented with a linear part. This corresponds to additional direct connections from the inputs to the output neuron. Mathematically, the RBF network, including a linear part, produces an output given by

$$\hat{y}(\theta) =$$

$$g(\theta, x) = \sum_{i=1}^{nb} w_i^2 \, e^{-\lambda_i^2 (x-w_i^1)^2} + w_{nb+1}^2 + \chi_1 x_1 + \ldots + \chi_n x_n \tag{10}$$

where *nb* is the number of neurons, each containing a basis function. The parameters of the RBF network consist of the positions of the basis functions $w_i^1$, the inverse of the width of the basis functions $\lambda_i$, the weights in output sum $w_i^2$, and the parameters of the linear part $\chi_1$, …, $\chi_n$. In most cases of function approximation, it is advantageous to have the additional linear part, but it can be excluded by using the options.

The parameters are often lumped together in a common variable $\theta$ to make the notation compact. Then you can use the generic description $g(\theta, x)$ of the neural network model, where $g$ is the network function and $x$ is the input to the network.

In training the network, the parameters are tuned so that the training data fits the network model Equation 2.10 as well as possible. This is described in Section 2.5.3, Training Feedforward and Radial Basis Function Networks.

In Equation 2.10 the basis function is chosen to be the Gaussian bell function. Although this function is the most commonly used basis function, other basis functions may be chosen. This is described in Section 13.3, Select Your Own Neuron Function.

Also, RBF networks may be multi-output as illustrated in Figure 2.8.

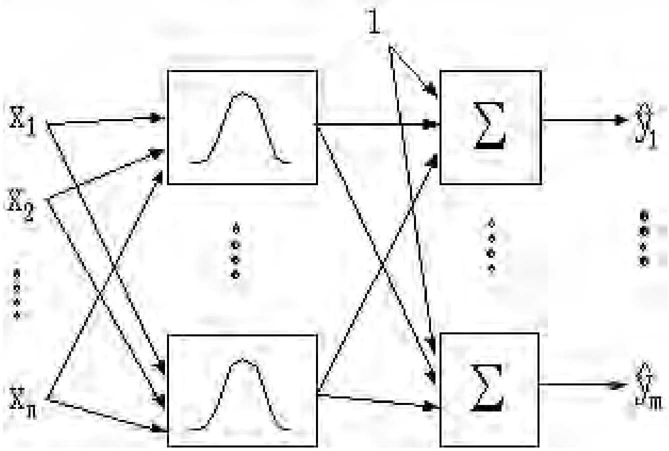


Figure 2.8. A multi-output RBF network.

FF networks and RBF networks can be used to solve a common set of problems. The built-in commands provided by the package and the associated options are very similar. Problems where these networks are useful include:

- Function approximation
- Classification
- Modeling of dynamic systems and time series

### 2.5.3 Training Feedforward and Radial Basis Function Networks

Suppose you have chosen an FF or RBF network and you have already decided on the exact structure, the number of layers, and the number of neurons in the different layers. Denote this network with $\hat{y} = g(\theta, x)$ where $\theta$ is a parameter vector containing all the parametric weights of the network and $x$ is the input. Then it is time to train the network. This means that $\theta$ will be tuned so that the network approximates the unknown function producing your data. The training is done with the command `NeuralFit`, described in Chapter 7, Training Feedforward and Radial Basis Function Networks. Here is a tutorial on the available training algorithms.

Given a fully specified network, it can now be trained using a set of data containing $N$ input-output pairs, $\{x_i, y_i\}_{i=1}^{N}$. With this data the mean square error (MSE) is defined by

$$V_N(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - g(\theta, x_i))^2 \tag{11}$$

Then, a good estimate for the parameter $\theta$ is one that minimizes the MSE; that is,

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\, V_N(\theta) \tag{12}$$

Often it is more convenient to use the root-mean-square error (RMSE)

$$\mathrm{RMSE}(\theta) = \sqrt{V_N(\theta)} \tag{13}$$

when evaluating the quality of a model during and after training, because it can be compared with the output signal directly. It is the RMSE value that is logged and written out during the training and plotted when the training terminates.

The various training algorithms that apply to FF and RBF networks have one thing in common: they are iterative. They both start with an initial parameter vector $\theta^0$, which you set with the command `InitializeFeedForwardNet` or `InitializeRBFNet`. Starting at $\theta^0$, the training algorithm iteratively decreases the MSE in Equation 2.11 by incrementally updating $\theta$ along the negative gradient of the MSE, as follows

$$\theta^{i+1} = \theta^i - \mu\, R\, \nabla_\theta V_N(\theta) \tag{14}$$

Here, the matrix $R$ may change the search direction from the negative gradient direction to a more favorable one. The purpose of parameter $\mu$ is to control the size of the update increment in $\theta$ with each iteration $i$, while decreasing the value of the MSE. It is in the choice of $R$ and $\mu$ that the various training algorithms differ in the *Neural Networks* package.

If $R$ is chosen to be the inverse of the *Hessian* of the MSE function, that is, the inverse of

$$\frac{d^2 V_N(\theta)}{d\theta^2} = \nabla_\theta^2 V_N(\theta) = \frac{2}{N}\sum_{i=1}^{N} \nabla_\theta g(\theta, x_i)\, \nabla_\theta g(\theta, x_i)^T - \frac{2}{N}\sum_{i=1}^{N} (y_i - g(\theta, x_i))\, \nabla_\theta^2 g(\theta, x_i) \tag{15}$$

then Equation 2.14 assumes the form of the Newton algorithm. This search scheme can be motivated by a second-order Taylor expansion of the MSE function at the current parameter estimate $\theta^i$. There are several drawbacks to using Newton's algorithm. For example, if the Hessian is not positive definite, the $\theta$ updates will be in the positive gradient direction, which will increase the MSE value. This possibility may be avoided with a commonly used alternative for $R$, the first part of the Hessian in Equation 2.15:

$$H = \frac{2}{N}\sum_{i=1}^{N} \nabla_\theta g(\theta, x_i)\, \nabla_\theta g(\theta, x_i)^T \tag{16}$$

With $H$ defined, the option `Method` may be used to choose from the following algorithms:

- Levenberg-Marquardt
- Gauss-Newton
- Steepest descent
- Backpropagation
- `FindMinimum`

**Levenberg-Marquardt**

Neural network minimization problems are often very ill-conditioned; that is, the Hessian in Equation 2.15 is often ill-conditioned. This makes the minimization problem harder to solve, and for such problems, the Levenberg-Marquardt algorithm is often a good choice. For this reason, the Levenberg-Marquardt algorithm method is the default training algorithm of the package.

Instead of adapting the step length $\mu$ to guarantee a downhill step in each iteration of Equation 2.14, a diagonal matrix is added to $H$ in Equation 2.16; in other words, $R$ is chosen to be

$$R = (H + e^{\lambda} I)^{-1} \tag{17}$$

and $\mu = 1$.

The value of $\lambda$ is chosen automatically so that a downhill step is produced. At each iteration, the algorithm tries to decrease the value of $\lambda$ by some increment $\Delta\lambda$. If the current value of $\lambda$ does not decrease the MSE in Equation 2.14, then $\lambda$ is increased in steps of $\Delta\lambda$ until it does produce a decrease.

The training is terminated prior to the specified number of iterations if any of the following conditions are satisfied:

- $\lambda > 10\Delta\lambda + \mathrm{Max}[s]$
- $\frac{V_N(\theta^i) - V_N(\theta^{i+1})}{V_N(\theta^i)} < 10^{-\mathrm{PrecisionGoal}}$

Here `PrecisionGoal` is an option of `NeuralFit` and *s* is the largest eigenvalue of *H*.

Large values of $\lambda$ produce parameter update increments primarily along the negative gradient direction, while small values result in updates governed by the Gauss-Newton method. Accordingly, the Levenberg-Marquardt algorithm is a hybrid of the two relaxation methods, which are explained next.

### Gauss-Newton

The Gauss-Newton method is a fast and reliable algorithm that may be used for a large variety of minimization problems. However, this algorithm may not be a good choice for neural network problems if the Hessian is ill-conditioned; that is, if its eigenvalues span a large numerical range. If so, the algorithm will converge poorly, slowing down the training process.

The training algorithm uses the Gauss-Newton method when matrix *R* is chosen to be the inverse of *H* in Equation 2.16; that is,

$$R = H^{-1} \tag{18}$$

At each iteration, the step length parameter is set to unity, $\mu = 1$. This allows the full Gauss-Newton step, which is accepted only if the MSE in Equation 2.11 decreases in value. Otherwise $\mu$ is halved again and again until a downhill step is affected. Then, the algorithm continues with a new iteration.

The training terminates prior to the specified number of iterations if any of the following conditions are satisfied:

- $\frac{V_N(\theta^i) - V_N(\theta^{i+1})}{V_N(\theta^i)} < 10^{-\mathrm{PrecisionGoal}}$
- $\mu < 10^{-15}$

Here PrecisionGoal is an option of NeuralFit.

### Steepest Descent

The training algorithm in Equation 2.14 reduces to the steepest descent form when

$$R = I \tag{19}$$

This means that the parameter vector $\theta$ is updated along the negative gradient direction of the MSE in Equation 2.13 with respect to $\theta$.

The step length parameter $\mu$ in Equation 2.14 is adaptable. At each iteration the value of $\mu$ is doubled. This gives a preliminary parameter update. If the criterion is not decreased by the preliminary parameter update, $\mu$ is halved until a decrease is obtained. The default initial value of the step length is $\mu = 20$, but you can choose another value with the StepLength option.

The training with the steepest descent method will stop prior to the given number of iterations under the same conditions as the Gauss-Newton method.

Compared to the Levenberg-Marquardt and the Gauss-Newton algorithms, the steepest descent algorithm needs fewer computations in each iteration, because there is no matrix to be inverted. However, the steepest descent method is typically much less efficient than the other two methods, so that it is often worth the extra computational load to use the Levenberg-Marquardt or the Gauss-Newton algorithm.

### Backpropagation

The backpropagation algorithm is similar to the steepest descent algorithm, with the difference that the step length $\mu$ is kept fixed during the training. Therefore the backpropagation algorithm is obtained by choosing R=I in the parameter update in Equation 2.14. The step length $\mu$ is set with the option StepLength, which has default $\mu = 0.1$.

The training algorithm in Equation 2.14 may be augmented by using a momentum parameter $\alpha$, which may be set with the Momentum option. The new algorithm is

$$\Delta\theta^{i+1} = -\mu \frac{dV_N(\theta)}{d\theta} + \alpha\Delta\theta^i \tag{20}$$

$$\theta^{i+1} = \theta^i + \Delta\theta^{i+1} \tag{21}$$

Note that the default value of $\alpha$ is 0.

The idea of using momentum is motivated by the need to escape from local minima, which may be effective in certain problems. In general, however, the recommendation is to use one of the other, better, training algorithms and repeat the training a couple of times from different initial parameter initializations.

**FindMinimum**

If you prefer, you can use the built-in *Mathematica* minimization command `FindMinimum` to train FF and RBF networks. This is done by setting the option `Method→FindMinimum` in `NeuralFit`. All other choices for `Method` are algorithms specially written for neural network minimization, which should be superior to `FindMinimum` in most neural network problems. See the documentation on `FindMinimum` for further details.

Examples comparing the performance of the various algorithms discussed here may be found in Chapter 7, Training Feedforward and Radial Basis Function Networks.

## 2.6 Dynamic Neural Networks

Techniques to estimate a system process from observed data fall under the general category of *system identification*. Figure 2.9 illustrates the concept of a system.

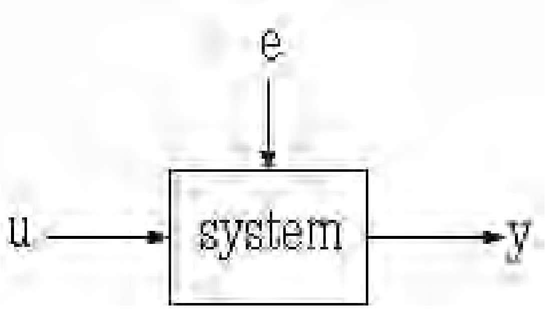


Figure 2.9. A system with input signal u, disturbance signal e, and output signal $y$.

Loosely speaking, a *system* is an object in which different kinds of signals interact and produce an observable output signal. A system may be a real physical entity, such as an engine, or entirely abstract, such as the stock market.

There are three types of signals that characterize a system, as indicated in Figure 2.9. The *output signal y(t)* of the system is an observable/measurable signal, which you want to understand and describe. The *input signal* *u(t)* is an external measurable signal, which influences the system. The *disturbance signal e(t)* also influences the system but, in contrast to the input signal, it is not measurable. All these signals are time dependent.

In a single-input, single-output (SISO) system, these signals are time-dependent scalars. In the multi-input, multi-output (MIMO) systems, they are represented by time-dependent vectors. When the input signal is absent, the system corresponds to a *time-series prediction* problem. This system is then said to be *noise driven*, since the output signal is only influenced by the disturbance *e(t)*.

The *Neural Networks* package supports identification of systems with any number of input and output signals.

A system may be modeled by a dynamic neural network that consists of a combination of neural networks of FF or RBF types, and a specification of the input vector to the network. Both of these two parts have to be specified by the user. The input vector, or *regressor vector*, which it is often called in connection with dynamic systems, contains lagged input and output values of the system specified by three indices: $n_a$, $n_b$, and $n_k$. For a SISO model the input vector looks like this:

$$\begin{aligned} x(t) = [&y(t-1)\ \ldots\ y(t-n_a) \\ &u(t-n_k)\ \ldots\ u(t-n_k-n_b+1)]^T \end{aligned} \qquad (22)$$

Index $n_a$ represents the number of lagged output values; it is often referred to as the *order of the model*. Index $n_k$ is the input delay relative to the output. Index $n_b$ represents the number of lagged input values. In a MIMO case, each individual lagged signal value is a vector of appropriate length. For example, a problem with three outputs and two inputs $n_a = \{1, 2, 1\}$, $n_b = \{2, 1\}$, and $n_k = \{1, 0\}$ gives the following regressor:

$$\begin{aligned} x(t) = [&y_1(t-1)\ y_2(t-1)\ y_2(t-2) \\ &y_3(t-1)\ u_1(t-1)\ u_1(t-2)\ u_2(t)] \end{aligned}$$

For time-series problems, only $n_a$ has to be chosen.

The dynamic part of the neural network defines a mapping from the regressor space to the output space. Denote the neural network model by $g(\theta, x(t))$ where $\theta$ is the parameter vector to be estimated using observed data. Then the prediction $\hat{y}(t)$ can be expressed as

$$\hat{y}(t) = g(\theta, x(t)) \qquad (23)$$

Models with a regressor like Equation 2.22 are called *ARX models*, which stands for *AutoRegressive with eXtra input signal*. When there is no input signal *u(t)*, its lagged valued may be eliminated from Equation 2.22, reducing it to an *AR model*. Because the mapping $g(\theta, x(t))$ is based on neural networks, the dynamic models are called *neural ARX* and *neural AR* models, or *neural AR(X)* as the short form for both of them. Figure 2.10 shows a neural ARX model, based on a one-hidden-layer FF network.

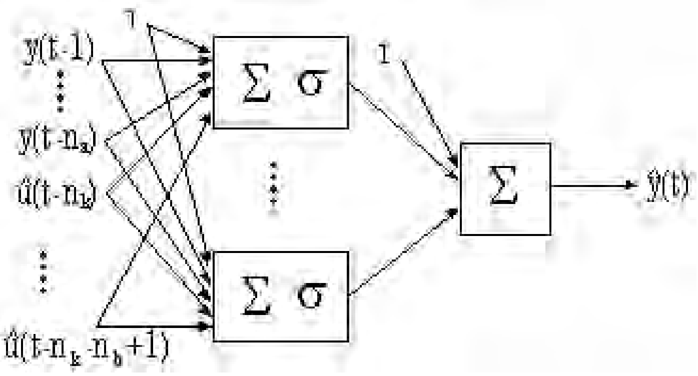


Figure 2.10. A neural ARX model.

The special case of an ARX model, where no lagged outputs are present in the regressor (that is, when $n_a$=*0* in Equation 2.22), is often called a *Finite Impulse Response* (FIR) model.

Depending on the choice of the mapping $g(\theta, x(t))$ you obtain a linear or a nonlinear model using an FF network or an RBF network.

Although the disturbance signal *e(t)* is not measurable, it can be estimated once the model has been trained. This estimate is called the *prediction error* and is defined by

$$\hat{e}(t) = y(t) - \hat{y}(t) \qquad (24)$$

A good model that explains the data well should yield small prediction errors. Therefore, a measure of $\hat{e}(t)$ may be used as a model-quality index.

System identification and time-series prediction examples can be found in Section 8.2, Examples, and Section 12.2, Prediction of Currency Exchange Rate.

## 2.7 Hopfield Network

In the beginning of the 1980s, Hopfield published two scientific papers that attracted much interest. This was the beginning of the new era of neural networks, which continues today.

Hopfield showed that models of physical systems could be used to solve computational problems. Such systems could be implemented in hardware by combining standard components such as capacitors and resistors.

The importance of the different Hopfield networks in practical application is limited due to theoretical limitations of the network structure, but, in certain situations, they may form interesting models. Hopfield networks are typically used for classification problems with binary pattern vectors.

The Hopfield network is created by supplying input data vectors, or pattern vectors, corresponding to the different classes. These patterns are called *class patterns*. In an *n*-dimensional data space the class patterns should have *n* binary components {1, −1}; that is, each class pattern corresponds to a corner of a cube in an *n*-dimensional space. The network is then used to classify distorted patterns into these classes. When a distorted pattern is presented to the network, then it is associated with another pattern. If the network works properly, this associated pattern is one of the class patterns. In some cases (when the different class patterns are correlated), spurious minima can also appear. This means that some patterns are associated with patterns that are not among the pattern vectors.

Hopfield networks are sometimes called *associative networks* because they associate a class pattern to each input pattern.

The *Neural Networks* package supports two types of Hopfield networks, a continuous-time version and a discrete-time version. Both network types have a matrix of weights *W* defined as

$$W = \frac{1}{n} \sum_{i=1}^{D} \xi_i{}^T \xi_i \tag{25}$$

where *D* is the number of class patterns $\{\xi_{1,}\ \xi_2,\ \ldots,\ \xi_D\}$, vectors consisting of $+/-1$ elements, to be stored in the network, and *n* is the number of components, the dimension, of the class pattern vectors.

Discrete-time Hopfield networks have the following dynamics:

$$x(t+1) = \mathrm{Sign}[W\,x(t)] \tag{26}$$

Equation 2.26 is applied to one state, $x(t)$, at a time. At each iteration the state to be updated is chosen randomly. This asynchronous update process is necessary for the network to converge, which means that $x(t) = \mathrm{Sign}[W\, x(t)]$.

A distorted pattern, $x(0)$, is used as initial state for the Equation 2.26, and the associated pattern is the state toward which the difference equation converges. That is, starting with $x(0)$ and then iterating Equation 2.26 gives the associated pattern when the equation converges.

For a discrete-time Hopfield network, the energy of a certain vector $x$ is given by

$$E\,(x) = -xWx^{T} \qquad (27)$$

It can be shown that, given an initial state vector $x(0)$, $x(t)$ in Equation 2.26 will converge to a value having minimum energy. Therefore, the minima of Equation 2.27 constitute possible convergence points of the Hopfield network. Ideally, these minima are identical to the class patterns $\{\xi_{1,}\ \xi_{2},\ \ldots,\ \xi_{D}\}$. Therefore, you can guarantee that the Hopfield network will converge to some pattern, but you cannot guarantee that it will converge to the *correct* pattern.

Note that the energy function can take negative values; this is, however, just a matter of scaling. Adding a sufficiently large constant to the energy expression it can be made positive.

The continuous-time Hopfield network is described by the following differential equation

$$\frac{dx\,(t)}{dt} = -x\,(t) + W\sigma[x\,(t)] \qquad (28)$$

where $x(t)$ is the state vector of the network, $W$ represents the parametric weights, and $\sigma$ is a nonlinearity acting on the states $x(t)$. The weights $W$ are defined in Equation 2.25. The differential equation, Equation 2.28, is solved using an Euler simulation.

To define a continuous-time Hopfield network, you have to choose the nonlinear function $\sigma$. There are two choices supported by the package: `SaturatedLinear` and the default nonlinearity of `Tanh`.

For a continuous-time Hopfield network, defined by the parameters given in Equation 2.25, you can define the *energy* of a particular state vector $x$ as

$$E\,(x) = -\frac{1}{2}\,xWx^{T} + \sum_{i=1}^{m}\int_{0}^{x_i}\sigma^{-1}\,(t)\,dt \qquad (29)$$

As for the discrete-time network, it can be shown that given an initial state vector $x(0)$, the state vector $x(t)$ in Equation 2.28 converges to a local energy minimum. Therefore, the minima of Equation 2.29 constitute the possible convergence points of the Hopfield network. Ideally these minima are identical to the class patterns $\{\xi_{1,}\ \xi_2,\ \ldots,\ \xi_D\}$. However, there is no guarantee that the minima will coincide with this set of class patterns.

Examples with Hopfield nets can be found in Section 9.2, Examples.

## 2.8 Unsupervised and Vector Quantization Networks

Unsupervised algorithms are used to find structures in the data. They can, for instance, be used to find clusters of data points, or to find a one-dimensional relation in the data. If such a structure exists, it can be used to describe the data in a more compact way.

Most network models in the package are trained with *supervised* training algorithms. This means that the desired output must be available for each input vector used in the training. *Unsupervised* networks, or *self-organizing networks*, rely only on input data and try to find structures in the input data space. The training algorithms are therefore called unsupervised.

Since there is no “correct” output, there will also not be any “incorrect” outputs. This fact leaves a lot of responsibility to the user. After an unsupervised network has been trained, it must be tested to show that it makes sense, that is, if the obtained structure is really representing the data. This validation can be very tricky, especially if you work in a high-dimensional space. In two- or three-dimensional problems, you can always plot the data and the obtained structure and simply examine them. Another test that can be applied in any number of dimensions is to check for the mean distance between the data points and the obtained cluster centers. A small mean distance means that the data is well represented by the clusters.

An unsupervised network consists of a number of *codebook vectors,* which constitute cluster centers. The codebook vectors are of the same dimension as the input space, and their components are the parameters of the unsupervised network. The codebook vectors are called the *neurons* of the unsupervised network.

When an unsupervised network is trained, the locations of the codebook vectors are adapted so that the mean Euclidian distance between each data point and its closest codebook vector is minimized. The algorithm, called competitive learning, is described in Section 10.1.2, UnsupervisedNetFit.

An unsupervised network can employ a *neighbor feature*. This gives rise to a s*elf-organizing map* (SOM). For SOM networks, not only is the mean distance between the data and nearest codebook vector minimized, but also the distance between the codebook vectors. In this way it is possible to define one- or two-dimensional relations among the codebook vectors, and the obtained SOM unsupervised network becomes a nonlinear mapping from the original data space to the one- or two-dimensional feature space defined by the codebook vectors. Self-organizing maps are often called *self-organizing feature maps*, or *Kohonen networks*.

When the data set has been mapped by a SOM to a one- or two-dimensional space, it can be plotted and investigated visually.

The training algorithm using the neighbor feature is described in Section 10.1.2, UnsupervisedNetFit.

Another neural network type that has some similarities to the unsupervised one is the *Vector Quantization* (VQ) network, whose intended use is classification. Like unsupervised networks, the VQ network is based on a set of codebook vectors. Each class has a subset of the codebook vectors associated to it, and a *data vector* is classified to be in the class to which the closest codebook vector belongs. In the neural network literature, the codebook vectors are often called the neurons of the VQ network.

Each of the codebook vectors has a part of the space "belonging" to it. These subsets form polygons and are called *Voronoi cells*. In two-dimensional problems you can plot these Voronoi cells.

The positions of the codebook vectors are obtained with a supervised training algorithm, and you have two different ones to choose from. The default one is called *Learning Vector Quantization* (LVQ) and it adjusts the positions of the codebook vectors using both the correct and incorrect classified data. The second training algorithm is the competitive training algorithm, which is also used for unsupervised networks. For VQ networks this training algorithm can be used by considering the data and the codebook vectors of a specific class independently of the rest of the data and the rest of the codebook vectors. In contrast to the unsupervised networks, the output data indicating the correct class is also necessary. They are used to divide the input data among the different classes.

## 2.9 Further Reading

Many fundamental books on neural networks cover neural network structures of interest. Some examples are the following:

M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, Cambridge, MA, The MIT Press, 1995.

S. Haykin, *Neural Networks: A Comprehensive Foundation,* 2nd ed., New York, Macmillan, 1999.

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Reading, MA, Addison-Wesley, 1991.

The following book concerns neural network simulation in *Mathematica*. It does not give very much background information on neural networks and their algorithms, but it contains programs and simulation examples. The book is not based on the *Neural Networks* package. Instead, the book contains the code for some neural network models.

J. A. Freeman, *Simulating Neural Networks with Mathematica*, Reading, MA, Addison-Wesley, 1994.

System identification and time-series prediction are broad and diverse fields, and there are many general and specialized books on these topics. The following list contains just some samples of the vast literature.

The following are good introductory books:

R. Johansson, *System Modeling and Identification*, Englewood Cliffs, NJ, Prentice Hall, 1993.

L. Ljung and T. Glad, *Modeling of Dynamic Systems*, Englewood Cliffs, NJ, Prentice Hall, 1994.

The following books are more thorough, and they are used in graduate courses at several universities:

L. Ljung, *System Identification: Theory for the User*, 2nd ed., Englewood Cliffs, NJ, Prentice Hall, 1999.

T. Söderström and P. Stoica, *System Identification*, Englewood Cliffs, NJ, Prentice Hall, 1989.

The following article discusses possibilities and problems using nonlinear identification methods from a user's perspective:

J. Sjöberg et al., "Non-Linear Black-Box Modeling in System Identification: A Unified Overview", *Automatica*, **31** (12), 1995, pp. 1691–1724.

This book is a standard reference for time-series problems:

G. E. P. Box and G. M. Jenkins, *Time Series Analysis, Forecasting and Control*, Oakland, CA, Holden-Day, 1976.

Many modern approaches to time-series prediction can be found in this book and in the references therein:

A. S. Weigend and N. A. Gershenfeld, "Time Series Prediction: Forecasting the Future and Understanding the Past", in *Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis held in Santa Fe, New Mexico, May 14–17, 1992*, Reading, MA, Addison-Wesley, 1994.

In most cases the neural network training is nothing other than minimization. It is, therefore, a good idea to consult standard books on minimization, such as:

R. Fletcher, *Practical Methods of Optimization*, Chippenham, Great Britain, John Wiley & Sons, 1987.

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

# 3 Getting Started and Basic Examples

The basic steps and conventions common to all neural network models contained in the *Neural Networks* package are explained in this chapter. You will also find some introductory examples.

## 3.1 Palettes and Loading the Package

The *Neural Networks* palettes contain sets of buttons that insert command templates for each type of neural network into a notebook. Section 3.1.2, Palettes, explains the general format of the palettes. Most of the commands described in the documentation can be inserted using a *Neural Networks* palette. To evaluate commands, the package must be loaded, as explained in Section 3.1.1, Loading the Package and Data.

### 3.1.1 Loading the Package and Data

The *Neural Networks* package is one of many available *Mathematica* applications, and it should be installed to the `$BaseDirectory/Applications` or `$UserBaseDirectory/Applications` directory. If this has been done at the installation stage, *Mathematica* should be able to find the application package without further effort on your part. To make all the functionality of the application package available at once, you simply load the package with the `Get`, `<<`, or `Needs` command.

Load the package and make all the functionality of the application available.

```
In[1]:= << NeuralNetworks`
```

If you get a failure message at this stage, it is probably due to a nonstandard location of the application package on your system. You will have to check that the directory enclosing the `NeuralNetworks` directory is included in your `$Path` variable. Commands such as `AppendTo[$Path,` *TheDirectoryNeuralNetworksIsIn*`]` can be used to inform *Mathematica* how to find the application. You may want to add this command to your `Init.m` file so it will execute automatically at the outset of any of your *Mathematica* sessions.

All commands in *Neural Networks* manipulate data in one way or another. If you work on data that was not artificially generated using *Mathematica,* then you have to load the data into *Mathematica*. For all illustrations in the documentation, the data is stored as *Mathematica* expressions and is loaded in the following way. Here `twoclasses.dat` is the name of a data file that comes with the *Neural Networks* package.

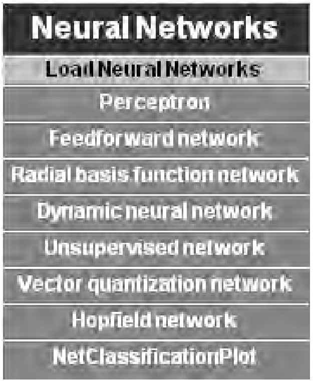
Load a data file.

```
In[2]:= << twoclasses.dat;
```

### 3.1.2 Palettes

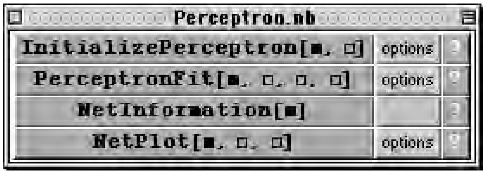
The *Neural Networks* package contains several palettes. As with the standard palettes that come with *Mathematica,* the palettes are available from the front end menu via the File ▹ Palettes command. These palettes provide you with an overview of the available commands, and their options and an easy method for inserting a function template into a notebook.

The main palette, shown in the following screen shot, is called the Neural Networks palette. This palette contains one button for each neural network type supported by the package.



The Neural Networks palette.

Clicking a button for a network type opens a subpalette that contains buttons for all functions of the chosen network. Function buttons on a subpalette are accompanied by buttons that open palettes of function options. Clicking a question-mark button opens online documentation. The following is a screen shot of the Perceptron palette.



The Perceptron palette.

You can also access the palettes from the online documentation in the Palettes subcategory.

## 3.2 Package Conventions

This section describes conventions common to all neural network types supported by *Neural Networks*. These conventions for function names, data format, and trained neural network storage make it easy to work with new types of networks once you have learned how to use one network type.

### 3.2.1 Data Format

To train a network, you need a set of data $\{x_i, y_i\}_{i=1}^{N}$ containing $N$ input-output pairs. All of the functions in the package require the same data format. The input and output data are each arranged in the form of a *Mathematica* matrix. Each individual input vector, $x_i$, is a vector on row $i$ of the input data matrix, and each $y_i$ is a vector on row $i$ of the output data matrix. An exception to this rule is when a neural network is applied to a single data item, in which case the data can be written as a vector rather than as a matrix.

Consider the sample data file `one2twodimfunc.dat` that is packaged with *Neural Networks*. This data item has $N = 20$ input-output pairs. Each $x_i$ is a vector of length 1, and each output item is a vector of length 2. To view the data, first load the package and then load the data file.

Load the *Neural Networks* package and the data file.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << one2twodimfunc.dat;
```

In this data file, the input and output matrices are assigned to the variable names `x` and `y`, respectively. Once the data set has been loaded, you can query the data using *Mathematica* commands. To better understand the data format and variable name assignment, you may also want to open the data file itself.

Show the contents of the input and output matrices.

```
In[3]:= x

Out[3]= {{0.}, {0.5}, {1.}, {1.5}, {2.}, {2.5}, {3.}, {3.5}, {4.}, {4.5},
          {5.}, {5.5}, {6.}, {6.5}, {7.}, {7.5}, {8.}, {8.5}, {9.}, {9.5}}
```

```
In[4]:= y

Out[4]= {{0., 1.}, {0.479426, 0.877583}, {0.841471, 0.540302}, {0.997495, 0.0707372},
         {0.909297, -0.416147}, {0.598472, -0.801144}, {0.14112, -0.989992},
         {-0.350783, -0.936457}, {-0.756802, -0.653644}, {-0.97753, -0.210796},
         {-0.958924, 0.283662}, {-0.70554, 0.70867}, {-0.279415, 0.96017},
         {0.21512, 0.976588}, {0.656987, 0.753902}, {0.938, 0.346635}, {0.989358, -0.1455},
         {0.798487, -0.602012}, {0.412118, -0.91113}, {-0.0751511, -0.997172}}
```

Check the number of data items and the number of inputs and outputs for each data item.

```
In[5]:= Dimensions[x]
        Dimensions[y]

Out[5]= {20, 1}

Out[6]= {20, 2}
```

The data set contains 20 data items with one input and two outputs per item.

Look at input and output of data item 14.

```
In[7]:= x[[14]]
        y[[14]]

Out[7]= {6.5}

Out[8]= {0.21512, 0.976588}
```

The next example demonstrates the data format for a classification problem. A classification problem is a special type of function approximation: the output of the classifier has discrete values corresponding to the different classes. You can work with classification as you would with any other function approximation, but it is recommended that you follow the standard described here so that you can use the special command options specifically designed for classification problems.

In classification problems input data is often called *pattern vectors*. Each row of the input data $x$ contains one pattern vector, and the corresponding row in the output data $y$ specifies the correct class of that pattern vector. The output data matrix $y$ should have one column for each class. On each row the correct class is indicated with a 1 in the correct class position, with the rest of the positions containing 0. If the classification problem has only two classes, then you can choose to have only one column in $y$ and indicate the classes with 1 or 0.

Consider the following example with three classes. The data is stored in `threeclasses.dat` in which the input matrix has been named `x` and the output data is assigned to matrix `y`. Although this is an artificially generated data set, imagine that the input data contains the age and weight of several children, and that these children are in three different school classes.

Load a data set.

```
In[9]:=  << threeclasses.dat;
```

Look at the 25th input data sample.

```
In[10]:=  x[[25]]

Out[10]=  {2.23524, 2.15257}
```

The children are from three different groups. The group is indicated by the position of the `1` in each row of the output `y`.

Verify in which class child 25 belongs.

```
In[11]:=  y[[25]]

Out[11]=  {0, 1, 0}
```

Since there is a 1 in the second column, the 25th child belongs to the second class.

Examples of classification problems can be found in Chapter 4, The Perceptron; Chapter 11, Vector Quantization; Section 12.1, Classification of Paper Quality; and Chapter 9, Hopfield Networks.

### 3.2.2 Function Names

Most neural network types rely on the following five commands, in which * is replaced by the name of the network type.

| | |
|---|---|
| `Initialize*` | initializes a neural network of indicated type |
| `*Fit` | trains a neural network of indicated type |
| `NetPlot` | illustrates a neural network<br>in a way that depends on the options |
| `NetInformation` | gives a string of information about the neural network |
| `NeuronDelete` | deletes a neuron from an existing network |

Common command structures used in the *Neural Networks* package.

`Initialize*` creates a neural network object with head equal to the name of the network type. The output of the training commands, `*Fit`, is a list with two elements. The first element is a trained version of the network, and the second is an object with head `*Record` containing logged information about the training. `NetPlot` can take `*Record` or the trained version of the network as an argument to return illustrations of the training process or the trained network. If `NetInformation` is applied to the network, a string of information about the network is given. These commands are illustrated in the examples in Section 3.4.1, Classification Problem Example and Section 3.4.2, Function Approximation Example. More examples can also be found in the sections describing the different neural network types.

In addition to these four commands, special commands for each neural network type are discussed in the chapter that focuses on the particular network.

### 3.2.3 Network Format

A trained network is identified by its head and list elements in the following manner.

- The head of the list identifies the type of network.
- The first component of the list contains the parameters, or weights, of the network.
- The second component of the list contains a list of rules indicating different network properties.

Consider this structure in a simple example.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

Create a perceptron network.

```
In[2]:= net = InitializePerceptron[{{1., -1.}, {-1., 1.}}, {1, 0}]

Out[2]= Perceptron[{w, b},
          {CreationDate → {2002, 4, 3, 13, 13, 5}, AccumulatedIterations → 0}]
```

The head is `Perceptron`. The first component contains the parameters of the neural network model, indicated by the symbol `{w, b}` for perceptrons. You obtain the parameters of a network by extracting the first element.

There are two replacement rules in the perceptron object. The one with left side `CreationDate` indicates when the network was created, and the other one, `AccumulatedIterations`, indicates the number of training iterations that have been applied to the network. In this case it is zero; that is, the network has not been trained at all.

Look at the parameters.

```
In[3]:= net[[1]]

Out[3]= {{{0.958944}, {0.313997}}, {-0.816685}}
```

You can store more information about a network model by adding more rules to the second component. The following example inserts the rule `NetworkName → JimsFavoriteModel` as the first element of the list in the second component of the neural network model.

Add a name to the network.

```
In[4]:= Insert[net, (NetworkName → JimsFavoriteModel), {2, 1}]

Out[4]= Perceptron[{w, b}, {NetworkName → JimsFavoriteModel,
           CreationDate → {2002, 4, 3, 13, 13, 5}, AccumulatedIterations → 0}]
```

## 3.3 NetClassificationPlot

The command `NetClassificationPlot` displays classification data in a plot. The format of the input data *x* and the output data *y* must follow the general format described in Section 3.2.1, Data Format.

| | |
|---|---|
| `NetClassificationPlot[x, y]` | plots data vectors *x* with the correct class indicated in *y* |
| `NetClassificationPlot[x]` | plots data vectors *x* without any information about class |

Display of classification data in a plot.

`NetClassificationPlot` has one option, which influences the way the data is plotted.

| *option* | *default value* | |
|---|---|---|
| `DataFormat` | `Automatic` | indicates how data should be plotted |

Option of `NetClassificationPlot`.

`DataFormat` can have one of the following two values:

- `DataMap` produces a two-dimensional plot based on `MultipleListPlot`. If the class information is submitted, different plot symbols will indicate the class to which the input belongs. This option is the default for two-dimensional data.
- `BarChart` produces a bar chart illustrating the distribution of the data over the different classes using the command `BarChart`. If the dimension of the input data is larger than two, then the default `DataFormat` is a bar chart. This type of plot allows you to see how the data is distributed over the classes.

You can influence the style of the plotting with any options of the commands `MultipleListPlot` and `BarChart`.

The next set of examples shows typical plots using two-dimensional data.

Load the *Neural Networks* package and demonstration data.

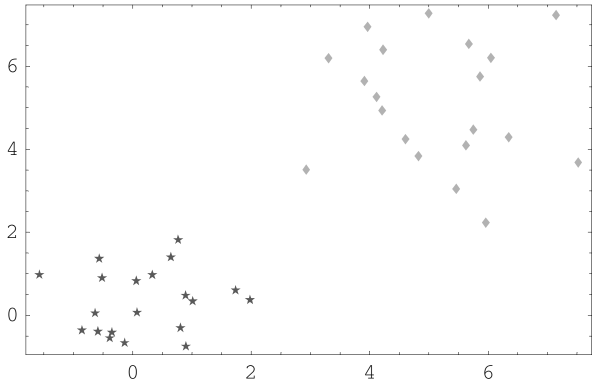
```
In[1]:= <<NeuralNetworks`
```

```
In[2]:=  << twoclasses.dat;
```

The input data is stored in *x* and the output data in *y*.

Make a two-dimensional plot of the data.

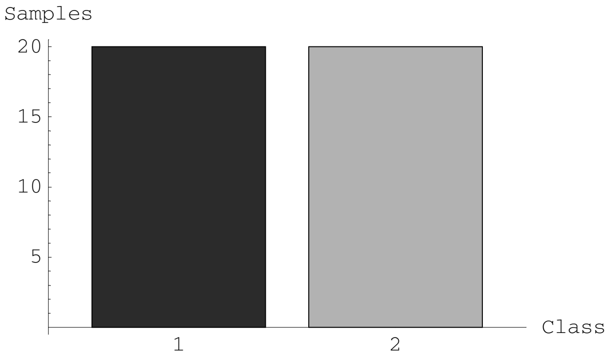
```
In[3]:=  NetClassificationPlot[x, y, SymbolStyle → {Hue[0.5], Hue[0.8]}]
```



The color of the data was changed using the option `SymbolStyle` of `MultipleListPlot`.

Illustrate the distribution of the data over the classes in a bar chart.

```
In[4]:=  NetClassificationPlot[x, y, DataFormat → BarChart, BarStyle → {Hue[0.7], Hue[0.5]}]
```



There are, therefore, 20 data samples in each of the two classes. The data is distributed evenly between the classes. The option `BarStyle` was used to change the colors of the bars.

For higher-dimensional data, you can also try to plot projections of the data. The next data set has three classes, but there are also three input dimensions.

Load new data.

```
In[5]:= << vqthreeclasses3D.dat;
```

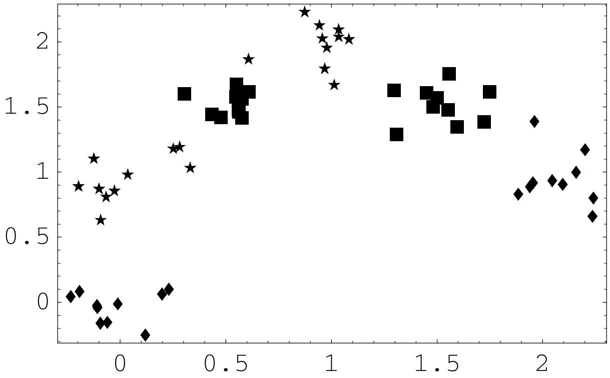
Check the dimensionality of the input space.

```
In[6]:= Dimensions[x]

Out[6]= {60, 3}
```

Three input dimensions cannot be plotted in a two-dimensional plot. Instead take the scalar product of x with a 3 × 2 matrix and then plot the resulting two-dimensional projection of the input data. In the following example, the first two dimensions of the input data are plotted. The plot symbols indicate the class y.

Project and plot the data.

```
In[7]:= NetClassificationPlot[x . {{1, 0}, {0, 1}, {0, 0}}, y]
```



## 3.4 Basic Examples

Each neural network problem follows the same basic steps: (1) load the data set; (2) initialize the network; (3) train the network; and (4) validate the model. Here the basic steps common to all neural network problems are elucidated with two examples: a classification problem and a function approximation problem. The options that alter or go beyond the basic steps are particular to the neural network model and so are discussed in later chapters. A detailed discussion of the meaning and manipulation of `NeuralFit` output can be found in the chapters that discuss individual network types.

### 3.4.1 Classification Problem Example

This subsection describes a small classification problem. This two-dimensional example is instructive because the data and the classifier can be visualized very easily. The data used in this example is stored in the file `twoclasses.dat` in which the input is stored in the matrix `x` and the output in `y`. To help understand the problem, assume that the input data is the age and weight of several children, and the output data represents the class to which each child belongs. There are two possible classes in this example.

Load the *Neural Networks* package and the data.

```
In[1]:= <<NeuralNetworks`

In[2]:=  << twoclasses.dat;
```

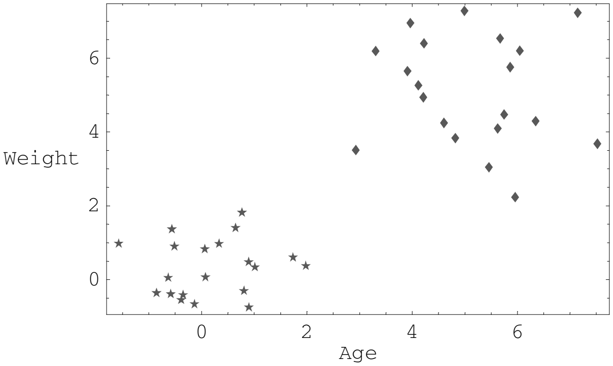
Because there are two possible classes, the output can be stored in one column, with a 1 or 0 indicating the class to which a child belongs. For a general discussion of data format, see Section 3.2.1, Data Format.

View the output data.

```
In[3]:= y

Out[3]= {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
          1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Plot the data, setting the axes labels according to the names of the measured values.

```
In[4]:= NetClassificationPlot[x,y,FrameLabel→{"Age","Weight"},SymbolStyle→
        {Hue[0.6],Hue[0.8]},RotateLabel→False]
```



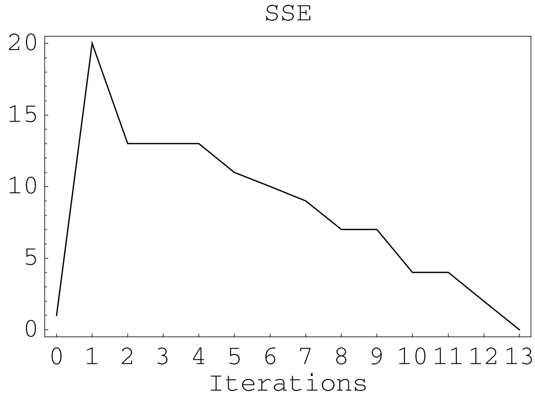
The plot clearly shows that the data is divided into two classes, and that it should be possible to divide the groups with a curve found during the training process. The successfully trained neural network classifier will return the correct group to which a child belongs, given the child's weight and age.

A *measure of fit*, or *performance index*, to be minimized by the training algorithm, must be chosen for the training to proceed. For classification problems, the criterion is set to the number of incorrectly classified data samples. The classifier has correctly classified all data when the criterion is zero.

A perceptron is the simplest type of neural network classifier. It is used to illustrate the training in this example. You can initialize a perceptron with `InitializePerceptron` and then use `PerceptronFit` to train the perceptron. However, perceptron initialization will take place automatically if you start with `PerceptronFit`.

Initialize and train a perceptron classifier using the data.

```
In[5]:= {per, fitrecord} = PerceptronFit[x, y];
```



Note that you will usually obtain slightly different results if you repeat the training command. This is due to the random initialization of the perceptron, which is described in Section 4.1.1, InitializePerceptron. As a result of this, the parametric weights of the perceptron will also be different for each evaluation, and you will obtain different classifiers.

During the evaluation of `PerceptronFit`, a separate notebook opens and displays the progress of the training. At the end of the training, a summary of the training process is shown in the plot of summed squared error (SSE) versus iteration number. The preceding plot is the summary of the training process for this example. You can see that the SSE tends toward 0 as the training goes through more iterations.

The first output argument of `PerceptronFit` is the trained perceptron `per` in this case. The second output argument, equal to `fitrecord` in this example, is a training record that contains information about the training procedure. See Section 7.8, The Training Record, for a description of how the training record can be used to analyze the quality of training. See Chapter 4, The Perceptron, for options that change the training parameters and plot.

The perceptron’s training was successful, and it can now be used to classify new input data.

Classify a child of age six and weight seven.

```
In[6]:= per[{6., 7.}]

Out[6]= {1}
```

You can also evaluate the perceptron on symbolic inputs to obtain a *Mathematica* expression describing the perceptron function. Then you can combine the perceptron function with any *Mathematica* commands to illustrate the classification and the classifier.

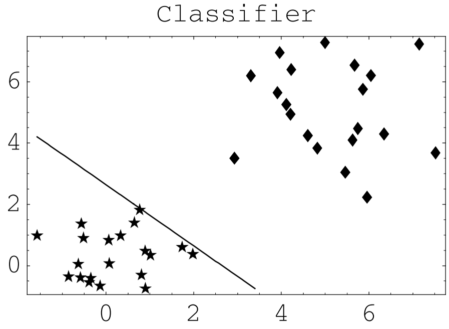
Obtain a *Mathematica* expression describing the perceptron.

```
In[7]:= Clear[a, b];
        per[{a, b}]

Out[8]= {UnitStep[-168.948 + 63.7735 a + 64.0374 b]}
```

Note that the numerical values of the parameter weights will be different when you repeat the example.

NetPlot can be used to illustrate the trained network in various ways, depending on the options given. The trained classifier can, for example, be visualized together with the data. This type of plot is illustrated using the results from the two-dimensional classifier problem. For this example, a successful classifier divides the two classes with a line. The exact position of this line depends on what particular solution was found in the training. All lines that separate the two clusters are possible results in the training.

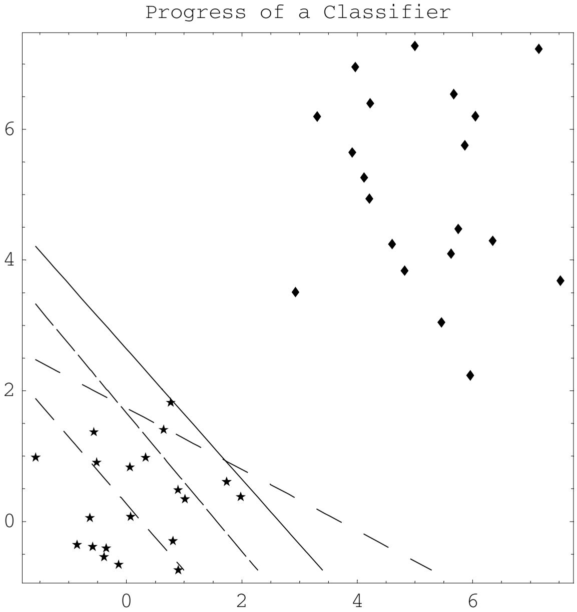
```
In[9]:= NetPlot[per, x, y]
```



NetPlot can also be used to illustrate the training process by applying it to the training record, the second argument of PerceptronFit.

Illustrate the training of the perceptron.

```
In[10]:= NetPlot[fitrecord, x, y]
```



The plot shows the classification of the initial perceptron and its improvement during the training.

The perceptron is described further in Chapter 4, The Perceptron, and you can find a list of neural networks that can be used for classification problems in Section 2.1, Introduction to Neural Networks.

### 3.4.2 Function Approximation Example

This subsection contains a one-dimensional approximation problem solved with a FF network. Higher-dimensional problems, except for the data plots, can be handled in a similar manner.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

Load data and *Mathematica*'s matrix add-on package.

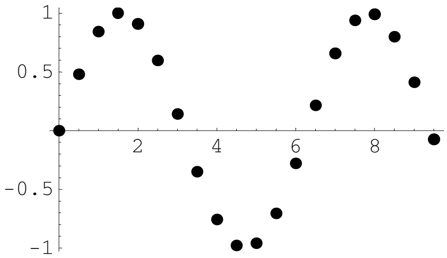
```
In[2]:= <<onedimfunc.dat;
        << LinearAlgebra`MatrixManipulation`
```

The file `onedimfunc.dat` contains the input and output data in the matrices defined as `x` and `y`, respectively. It is assumed that the input-output pairs are related by $y = f(x)$, where $f$ is an unspecified function. The data will be used to train a feedforward network that will be an approximation to the actual function $f$.

To motivate the use of a neural network, imagine that both $x$ and $y$ are measured values of some product in a factory, but $y$ can be measured only by destroying the product. Several samples of the product were destroyed to obtain the data set. If a neural network model can find a relationship between $x$ and $y$ based on this data, then future values of $y$ could be computed from a measurement of $x$ without destroying the product.

Plot the data.

```
In[4]:= ListPlot[AppendRows[x, y], PlotStyle → PointSize[0.03]]
```



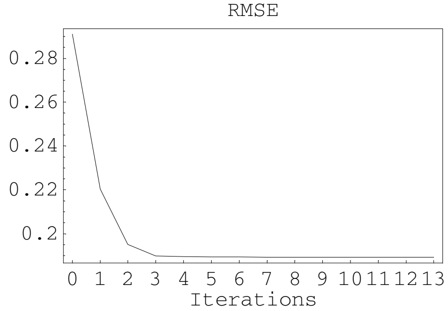
This is a very trivial example; the data was generated with a sinusoid. A feedforward network will be trained to find such an approximation. More information on this kind of neural network can be found in Chapter 5, The Feedforward Neural Network.

Initialize a feedforward network with three neurons.

```
In[5]:= fdfrwrd = InitializeFeedForwardNet[x, y, {3}]

Out[5]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 16, 51},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

Train the initialized network.

```
In[6]:=  {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y];
```



Note that you will usually obtain slightly different results if you repeat the initialization and the training command. This is due to the partly random initialization of the feedforward network, which is described in Section 5.1.1, InitializeFeedForwardNet.

As with the previous example, the improvement of the fit is displayed in a separate notebook during the training process. At the end of the training, the fit improvement is summarized in a plot of the RMSE in the neural network prediction versus iterations. `NeuralFit` options allow you to change the training and plot features. At the end of the training, you will often receive a warning that the training did not converge. It is usually best to visually inspect the RMSE decrease in the plot to decide if more training iterations are needed. How this can be done is illustrated in Section 5.2.1, Function Approximation in One Dimension. For now, assume that the network has been successfully trained, though later you can plot the model to compare it to the data.

The first output argument of `NeuralFit` is the trained feedforward network. The second argument is a training record containing information about the training procedure. See Section 7.8, The Training Record, for a discussion of how to use the training record to analyze the quality of training.

The trained neural network can now be applied to a value *x* to estimate *y* = *f(x)*.

Produce an estimate for `y` when `x=3`.

```
In[7]:=  fdfrwrd2[{3}]

Out[7]=  {0.241326}
```

To obtain a *Mathematica* expression describing the network, apply the network to symbolic input. Then you can use *Mathematica* commands to plot and manipulate the network function.

Obtain a *Mathematica* expression describing the network.

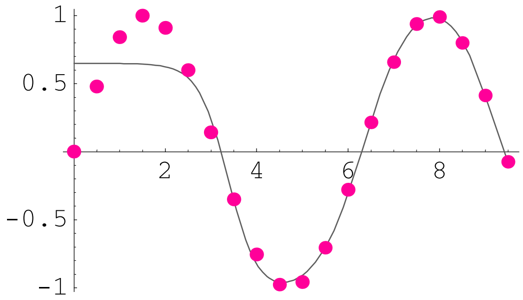
```
In[8]:=  Clear[a];
         fdfrwrd2[{a}]
```

$$\text{Out[9]=}\ \left\{0.648702-\frac{1.7225}{1+e^{10.3493-3.06279\,a}}-\frac{1.77352}{1+e^{21.1303-2.31796\,a}}+\frac{2.26213}{1+e^{12.3109-1.93971\,a}}\right\}$$

The special command `NetPlot` illustrates the trained neural network in a way indicated with the option `DataFormat`. For one- and two-dimensional problems you can use it to plot the neural network function.

Plot the function estimate together with the data.

```
In[10]:= NetPlot[fdfrwrd2, x, y, DataFormat → FunctionPlot,
          PlotStyle → {Hue[0.9], PointSize[0.03]}]
```



Depending on the option `DataFormat`, `NetPlot` uses different *Mathematica* plot commands, and you may submit any options of these commands to change the way the result is displayed. For example, in the illustrated plot the color was changed.

# 4 The Perceptron

The *perceptron* is the simplest type of neural network, and it is typically used for classification.

Perceptron classifiers are trained with a supervised training algorithm. This means that the true class of the training data must be available so that they can be submitted to the training function. If a data sample is incorrectly classified then the algorithm modifies the network weights so that performance improves. This is an iterative training process that continues until the perceptron correctly classifies all data or the upper limit of training iterations is reached.

It is often possible to use a smart initialization of the weights of the perceptron so that the extent of the iterative training may be minimized or entirely circumvented.

A short tutorial about the perceptron is given in Section 2.4, The Perceptron. Section 4.1, Perceptron Network Functions and Options, defines the commands and the options to work with perceptron networks, and in Section 4.2, Examples, you can find examples illustrating the commands. A small introductory example also appears in Section 3.4.1, Classification Problem Example.

## 4.1 Perceptron Network Functions and Options

This section describes the functions to initialize, train, and use perceptrons. Examples can be found in Section 4.2, Examples.

### 4.1.1 InitializePerceptron

A perceptron model can be initialized with `InitializePerceptron` and then trained, or fitted, to the data with `PerceptronFit`. You can also call `PerceptronFit` without an initialized perceptron model. In this case, the initialization is done internally.

`InitializePerceptron` is called in the following way.

`InitializePerceptron[`$x$, $y$`]`
initializes a perceptron of the appropriate dimensionality according to the supplied data, $x$-input data vectors, and $y$-outputs

Initializing a perceptron network.

The result is placed in an object of type `Perceptron`. In the first position, you find the parametric weights {$w$, $b$}, as described in Section 2.4, The Perceptron. In the multi-output case, there is one column in $w$ and one component in $b$ for each of the outputs.

The perceptron automatically assumes as many inputs as the dimensionality of the input data $x$, and as many outputs as the dimensionality of the output data $y$.

`InitializePerceptron` takes one option.

| *options* | *default value* | |
|---|---|---|
| `RandomInitialization` | `True` | indicates random initialization |

Option of `InitializePerceptron`.

If the option is set to `False`, a smart initialization is used, which sets the parametric weights so that the initial classifier is placed between the data clusters. In the multi-output case, the weights of each output are initialized in this way. This often gives such a good initial perceptron model that no training is necessary.

### 4.1.2 PerceptronFit

The command `PerceptronFit` is used to train a perceptron. It can be called with an already-existing perceptron model that is then trained, or it can be called with only data, in which case a perceptron will first be initialized. If the number of iterations is not specified, it will assume a default value of 100 or stop sooner when all patterns are correctly classified.

| | |
|---|---|
| `PerceptronFit[`*x*`,` *y*`]` | initializes and trains a perceptron with the default number of iterations |
| `PerceptronFit[`*x*`,` *y*`,` *iterations*`]` | initializes and trains a perceptron with the indicated number of iterations |
| `PerceptronFit[`*x*`,` *y*`,` *perceptron*`]` | trains the submitted perceptron with the default number of iterations |
| `PerceptronFit[`*x*`,` *y*`,` *perceptron, iterations*`]` | trains the submitted perceptron with the indicated number of iterations |

Training a perceptron network.

`PerceptronFit` returns a list containing two elements. The first element is the trained perceptron and the second is an object of type `PerceptronRecord`, which contains information of the training session. It can be used to evaluate the quality of the training using the command `NetPlot`. The structure of the training record is explained in Section 7.8, The Training Record.

An existing perceptron can be submitted for more training by setting *perceptron* equal to the perceptron or its training record. The advantage of submitting the training record is that the information about the earlier training is combined with the additional training.

During the training, intermediate results are displayed in a separate notebook, which is created automatically. After each training iteration the number of misclassified training patterns is displayed together with the iteration number. At the end of the training, this information is shown in a plot. By changing the options, as described in Section 7.7, Options Controlling Training Results Presentation, you can change or switch off this information about the training.

`PerceptronFit` takes the following options.

| *options* | *default values* | |
|---|---|---|
| `RandomInitialization` | `True` | indicates that random initialization should be used |
| `StepLength` | `Automatic` | sets the step length $\eta$ for the training algorithm to any nonnegative numerical value |
| `CriterionPlot` | `True` | plots the improvement of the classifier as a function of iteration number after the training |

| | | |
|---|---|---|
| `CriterionLog` | `True` | writes out the intermediate classification result during training |
| `CriterionLogExtN` | `True` | writes out the intermediate classification result in an external notebook |
| `ReportFrequency` | `1` | logs the intermediate results with the indicated interval |
| `MoreTrainingPrompt` | `False` | prompts for more training iterations if set to `True` |

Options of `PerceptronFit`.

If `StepLength→Automatic` then it is set according to Equation 2.6 in Section 2.4, The Perceptron. That default value is good for a large variety of problems. You can, however, supply any other positive numerical value.

The options `CriterionPlot`, `CriterionLog`, `CriterionLogExtN`, `ReportFrequency`, and `MoreTrainingPrompt` influence the way the results of `PerceptronFit` are presented and they are similar to the other training commands in the *Neural Networks* package. They are described in Section 7.7, Options Controlling Training Results Presentation.

A derived perceptron can be applied to new input vectors using function evaluation.

| | |
|---|---|
| *per* [*x*] | evaluates the perceptron *per* on the input vector *x* |

Function evaluation of a perceptron network.

The input argument *x* can be a vector containing one input vector or a matrix containing one input vector on each row.

### 4.1.3 NetInformation

Some information about a perceptron is presented in a string by the function `NetInformation`.

| | |
|---|---|
| `NetInformation` [*per*] | writes out information about a perceptron |

The `NetInformation` function.

### 4.1.4 NetPlot

`NetPlot` can be used to illustrate a perceptron, its classification, and its training. Depending on how the option `DataFormat` is set, you can illustrate the classification and the training in different ways.

| | |
|---|---|
| `NetPlot[`*per*`, `*x*`, `*y*`]` | illustrates the classification result for the given perceptron model with the submitted $x$ input $y$ output data |
| `NetPlot[`*per*`, `*x*`]` | illustrates the classification result for the given perceptron model with only input data |
| `NetPlot[`*fitrecord*`, `*x*`, `*y*`]` | illustrates the classifier during training |
| `NetPlot[`*fitrecord*`, `*x*`]` | illustrates the classifier during training with only input data |

Display of perceptrons and training of perceptrons.

If the output data $y$ is submitted then the correct class and the class according to the perceptron model can be indicated. For two-dimensional problems the default is to plot the data and the border line(s) of the classifier. For higher-dimensional data the result is represented with bar charts.

`NetPlot` takes the following options when it is applied to a perceptron network.

| *options* | *default values* | |
|---|---|---|
| `DataFormat` | `Automatic` | indicates the way in which the data should be illustrated; the default depends on the dimension of the data (possibilities are described in the following paragraphs) |
| `Intervals` | `5` | number of iterations between displayed plots |
| `BoundaryDash` | `True` | specifies if the boundaries at different stages of the training should be illustrated with longer dashes as training proceeds |
| `Compiled` | `True` | use compiled version |

Options of `NetPlot`.

The option DataFormat takes the following values:

- DataMap is the default for two-dimensional problems, producing a plot of data along with the class discriminant. If a training record is submitted, then the class discriminant for the intermediate classifiers during training are also plotted. The option Intervals can be used to set the frequency of the intermediate classifiers.
- BarChart illustrates the classification result with a bar chart.
- ClassPerformance is the default if a training record is supplied instead of a perceptron and when the input dimension is larger than two. The classification performance is plotted against training iterations with one plot for each class. Each plot indicates the classification performance versus training iterations for one specific class. The solid line indicates the number of correctly classified data samples to that class, and the dashed line indicates the number of incorrectly classified data samples to that class.
- DataMapArray gives a graphics array of the progress of the classification during training. This applies only to two-dimensional problems and when a training record is submitted.

The last two possibilities are encountered only when NetPlot is applied to a training record. The option BoundaryDash is active only when DataFormat→DataMap and when NetPlot is applied to a training record.

Depending on the value of DataFormat, NetPlot uses the *Mathematica* command MultipleListPlot, BarChart, or BarChart3D, and you can modify the plot by submitting the related options.

With DataFormat→BarChart the classification result is illustrated with a three-dimensional bar chart indicating class according to output data and class according to the perceptron model. The height of the bar at position {$n$, $m$} illustrates the number of data objects belonging to class $n$ according to the supplied output, and classified to class $m$ by the perceptron. Therefore, the bars on the diagonal correspond to correctly classified data.

If no output is submitted, then only the classification according to the model can be given, and the plot illustrates how the input data is distributed over the classes according to the classification of the perceptron.

## 4.2 Examples

This subsection gives some simple examples where perceptrons are used to classify data from different classes. Remember that the perceptron is a very simple type of classifier and in most real problems you might need more advanced models.

The first two examples use two-dimensional input data. This is helpful because the data and the classifier can be depicted graphically. In the first example there are only two classes, which can be separated using a perceptron with one output. The second example handles three classes, and you need a perceptron with three outputs. This is equivalent to three single-output perceptrons in parallel.

The third example considers classification in a three-dimensional space. The main difference from the two-dimensional classification is that you cannot properly plot the data.

Notice that if you re-evaluate the examples you will not receive exactly the same results due to the random initialization used in training perceptron networks. See Section 4.1.1, InitializePerceptron.

### 4.2.1 Two Classes in Two Dimensions

Consider a two-dimensional classification problem. First the package and a data set are loaded.

Load the *Neural Networks* package and a data set.

```
In[1]:= <<NeuralNetworks`
```

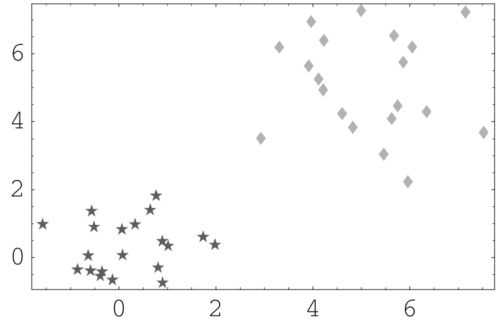
```
In[2]:=  << twoclasses.dat;
```

The input data is stored in $x$ and the output in $y$. The data format follows the standard form of the package as described in Section 3.2, Package Conventions. The data set is the same as the one described in Section 3.4.1, Classification Problem Example.

The starting point should always be to look at the data.

Look at the data.

```
In[3]:= NetClassificationPlot[x,y,SymbolStyle→{Hue[.5],Hue[.9]}]
```

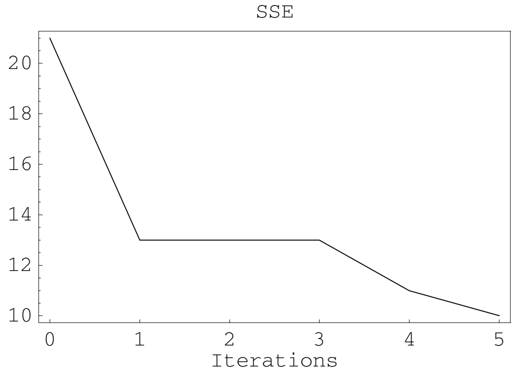


The two classes are distinguished by different labels according to the information in the output *y*. The positions of the different patterns are given by the input *x*.

You can now fit a perceptron classifier to the data. In other words, you train a perceptron to divide the data in the correct manner.

Initialize and train a perceptron using five iterations.

```
In[4]:= {per, fitrecord} = PerceptronFit[x, y, 5];
```



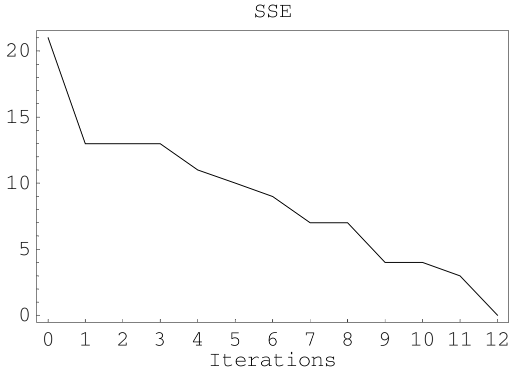
```
PerceptronFit::NotConverged :
 No solution found. The problem might not be linearly
   separable or you may need more iterations.
```

Since the perceptron did not learn to classify all the training data correctly, a warning is given at the end of the training. The cause of the failure might be that the perceptron is too simple a classifier for the problem at hand, or that it needs more training iterations. You can repeat the previous command with an increased number of iterations, or you can submit the trained perceptron for more training.

More training can be applied to an existing perceptron in two ways: by submitting the perceptron or by submitting its training record. The advantage of submitting the training record is that the information about the new training will be combined with the earlier training and added in the new training record. In this way it is possible to examine all the training more easily. This is illustrated here.

Train the perceptron with another 10 iterations.

```
In[5]:= {per, fitrecord2} = PerceptronFit[x, y, fitrecord, 10];
```



Now the perceptron has learned to classify all of the training data. Not all of the specified training iterations were necessary (in addition to the first five iterations). Notice that these results may turn out differently if you re-evaluate the commands due to the random initialization. You may need more iterations.

The SSE is computed at each iteration and displayed in a separate notebook. At the end of the training, a plot is produced in the main notebook showing the SSE reduction as a function of iteration number. These features may be changed by setting different options in the call to `PerceptronFit`. This is described in Section 7.7, Options Controlling Training Results Presentation.

Provide some information about the perceptron.

```
In[6]:= NetInformation[per]

Out[6]= Perceptron model with 2 inputs and 1 output. Created 2002-4-3 at 13:19.
```

The trained perceptron can be used to classify data input vectors by just applying the perceptron object to these vectors. The output is 0 or 1, depending on the class to which the input vectors belong.

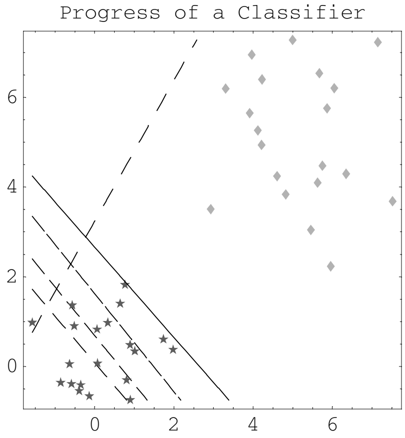
Classify two new input vectors.

```
In[7]:= per[{{1., 1.2}, {5.1, 5.3}}]

Out[7]= {{0}, {1}}
```

The training progress of the perceptron can be illustrated with the command `NetPlot` in different ways, depending on the option `DataFormat`. The default for two-dimensional problems is to plot the data together with the classification boundary. The length of the dashes illustrates the stage of the training. The final classification boundary is indicated with a solid line.

Display the training progress of the classification.

```
In[8]:= NetPlot[fitrecord2, x, y, Intervals → 3, SymbolStyle → {Hue[.5], Hue[.9]}]
```



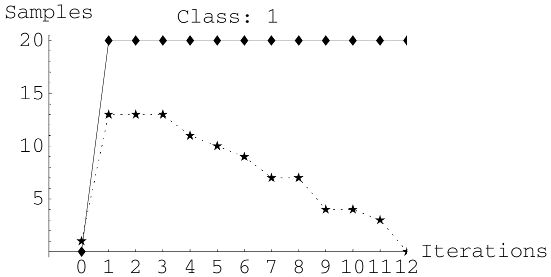
It is only in two-dimensional problems that the classifier can be illustrated with plots showing the data points and the class boundary. In higher-dimensional problems you can instead plot the number of correctly and incorrectly classified data samples over all the classes as a function of the number of training iterations. Of course, this can also be done in two-dimensional problems.

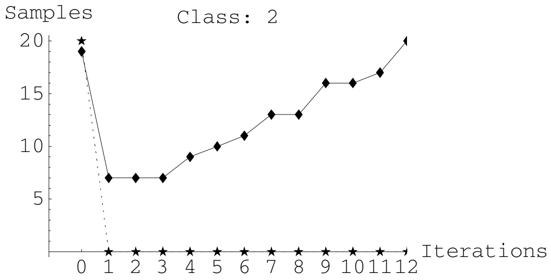
Notice that the final classification boundary barely cleared the last element of the star class displayed in the plot. Intuitively it seems better to have the classification boundary in the middle between the classes. It is, however, a feature of the perceptron training algorithm that the training terminates as soon as all the training data is classified correctly, and the classification boundary may then be anywhere in the gap between the classes.

Look at the number of correctly and incorrectly classified data objects to each class versus training iterations.

```
In[9]:= NetPlot[fitrecord2, x, y, DataFormat → ClassPerformance]
```

```
Correctly/incorrectly classified data
```



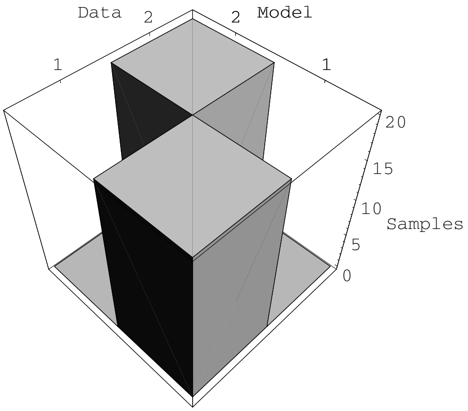


Each plot indicates the classification performance versus training iterations for one specific class. The solid line indicates the number of correctly classified data objects assigned to that class, and the dashed line indicates the number of samples from other classes that are incorrectly classified to the class.

The classification result can also be illustrated with bar charts. On the $x$ and $y$ axes you have the class of the samples according to the output data and according to the perceptron classifier, and on the $z$ axis you have the number of samples. For example, in the bin (2, 1) is the number of data samples from the second class, according to the supplied output data, that were classified into the first class by the network. Therefore, the diagonal represents correctly classified data and the off-diagonal bars represent incorrectly classified data.

Display the classification of the trained perceptron with bar charts.

```
In[10]:= NetPlot[per, x, y, DataFormat → BarChart]
```

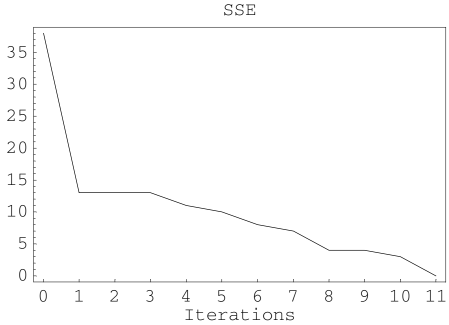


The success of the classifier depends on the data and on the random initialization of the perceptron. Each time `PerceptronFit` is called, without an already-initialized perceptron, a new random initialization is obtained. Repeat the last command a couple of times to see how the training will evolve differently because of the different initial weights.

You can change the step length, described in Equation 2.6 in Section 2.4, The Perceptron, by setting the option `StepLength` to any positive value.

Train a new perceptron with a different step length.

```
In[11]:= {per, fitrecord} = PerceptronFit[x, y, StepLength → 2.];
```



The warning message at the end of the training informs you that the perceptron did not succeed to learn to classify the training data correctly. You could continue with more training iterations, change the step length, or just repeat the command, which then gives a new initialization from which the training might be more successful.

It is easy to extract the weights of a perceptron since they are stored in the first element. In the same manner you can insert new values of the weights. This is further described in Section 13.1, Change the Parameter Values of an Existing Network.

Extract the perceptron weights.

```
In[12]:= w = MatrixForm[per[[1, 1]]]
         b = MatrixForm[per[[1, 2]]]

  Out[12]//MatrixForm=
        ( 1.27812 )
        (0.963722 )

  Out[13]//MatrixForm=
        ( -3.2454 )
```

### Intelligent Initialization

The classification problem considered in this example is not very hard. Actually, most problems where the perceptron can be used are fairly easy “toy” problems. To illustrate this, the data can be used for the initialization of the perceptron as described in connection with Section 4.1.1, InitializePerceptron. You can use the smart initialization by setting the option `RandomInitialization→False`, and often there is no need for any additional training when this initialization is used.

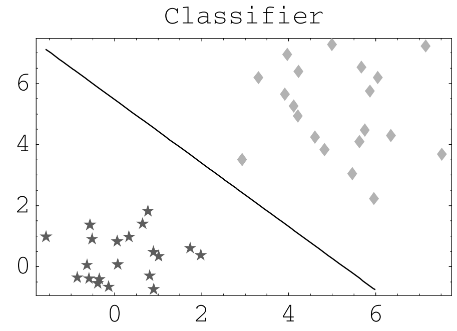
Initialize and train a perceptron using the smart initialization.

```
In[14]:= {per, fitrecord} =
           PerceptronFit[x, y, 5, CriterionLog → False, RandomInitialization → False];

       PerceptronFit::InitialPerfect : The initialized perceptron
          classifies all data correctly, no iterative learning is necessary.
```

The message given tells you that the initialized perceptron correctly classifies all training data and, therefore, no iterative training is necessary. This can also be seen clearly if you plot the result.

Plot the perceptron classifier together with the data.

```
In[15]:= NetPlot[per,x,y,SymbolStyle→{Hue[.5],Hue[.9]}]
```



You can also check the performance by evaluating the perceptron on the training data and comparing the output with the true output. If you subtract the classification result from the true output, the correctly classified data samples are indicated with zeros.

Check the performance of the initialized perceptron.

```
In[16]:= Flatten[per[x]] - y

Out[16]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Are there any incorrect ones?

### 4.2.2 Several Classes in Two Dimensions

There were only two classes in the previous example. Therefore, one output was sufficient to classify an input pattern vector to one of two classes, identified by 0 and 1. Consider now an example with more than two classes. To keep things simple a problem with three classes is chosen. A perceptron needs one output for each class, that is, you need a perceptron with three outputs. This is equivalent to having three single-output perceptrons in parallel. Each output indicates by taking the value 1 or 0 whether a pattern belongs to that class or not. Notice that there is no a priori problem for a pattern to belong to no class at all or to several classes simultaneously. It is up to you as a user to decide what to do in such cases.

Load the *Neural Networks* package and the data.

```
In[1]:= <<NeuralNetworks`

In[2]:=  << threeclasses.dat;
```

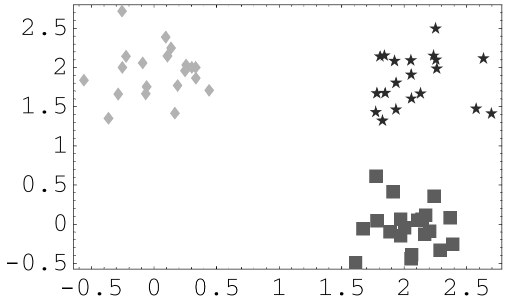
Check the dimensions of the data.

```
In[3]:=  Dimensions[x]
         Dimensions[y]

Out[3]=  {60, 2}

Out[4]=  {60, 3}
```

There are 60 data samples, two inputs, and three outputs.

Look at the data.

```
In[5]:= NetClassificationPlot[x,y,SymbolStyle→{Hue[.5],Hue[.7],Hue[.9]}]
```



By analogy to the data in Section 3.4.1, Classification Problem Example, this data could correspond to scaled values of age and weight for children from three different groups.

Check the age, weight, and class of the 25th child.

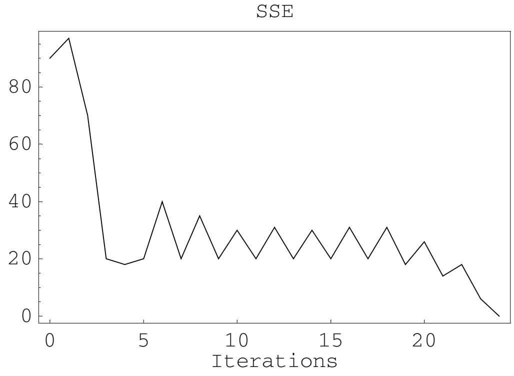
```
In[6]:=  x[[25]]
         y[[25]]

Out[6]=  {2.23524, 2.15257}

Out[7]=  {0, 1, 0}
```

It belongs to the second class since there is a 1 in the second column of $y$.

The number of perceptron inputs and outputs do not have to be specified in the function call. They are implied by the number of columns in $x$ and $y$. Therefore, the command to initialize and train the perceptron is given as before.

Initialize and train a perceptron.

```
In[8]:=  {per, fitrecord} = PerceptronFit[x, y];
```



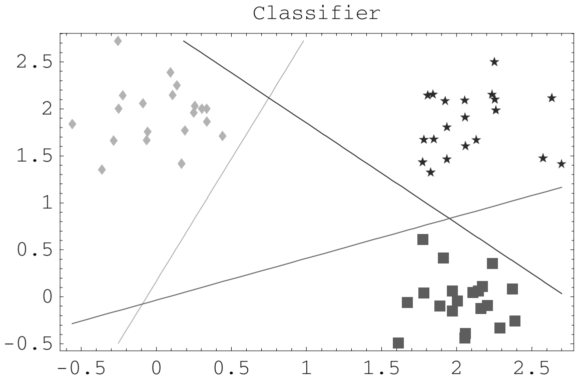
The perceptron can be used to classify any input vector by using the evaluation rule of the perceptron object.

Classify some data vectors using the trained perceptron.

```
In[9]:= per[x[[{1,5,10,15,20,30,40,50,60}]]]

Out[9]= {{1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
         {1, 0, 0}, {0, 1, 0}, {0, 1, 0}, {0, 0, 1}, {0, 0, 1}}
```

Check how the perceptron performs on the data.

```
In[10]:= NetPlot[per,x,y,SymbolStyle→{Hue[.5],Hue[.7],Hue[.9]},ContourStyle→
         {Hue[.5],Hue[.7],Hue[.9]}]
```

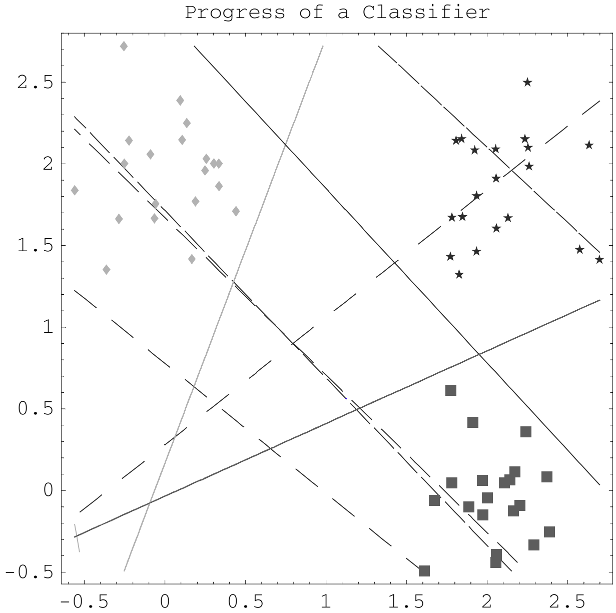


Notice that each output of the perceptron gives a boundary line, indicating one class. There are some areas in the plot that belong to two classes, and the area in the center does not belong to any of the classes. There is no a priori way to handle these ambiguities. Instead, they are artifacts of the perceptron classifier, and it is up to you as a user to decide how to handle these areas. However, it does not necessarily have to be a big problem. On this data, for example, there are no data items in the ambiguous areas where no clear decision can be taken.

As in the previous example, if you want to see how the perceptron classifier evolved during the training, you submit the training record instead of the trained perceptron in the call to `NetPlot`.

Display the training progress of the classification.

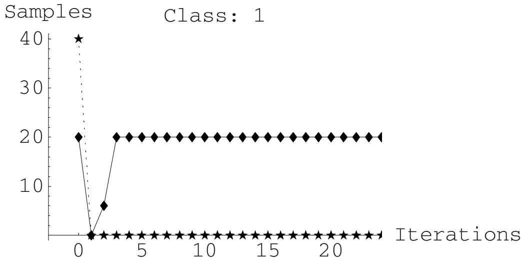
```
In[11]:= NetPlot[fitrecord, x, y, Intervals → 3, SymbolStyle → {Hue[.5], Hue[.7], Hue[.9]},
           ContourStyle → {Hue[.5], Hue[.7], Hue[.9]}]
```

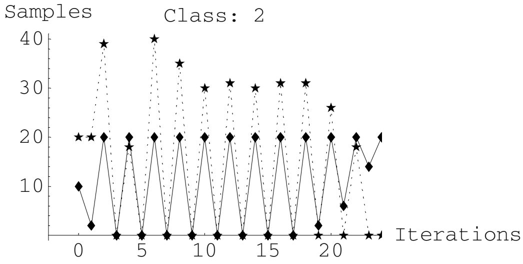


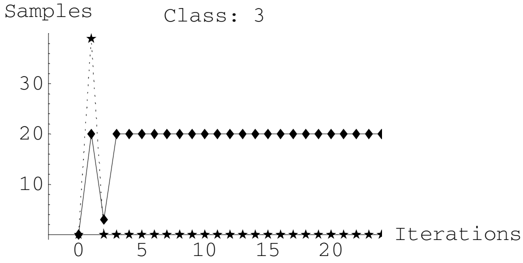
The plot shows the classification lines between the classes at different stages of the training. Since the plot is quite messy, it might be more interesting to look at the development of the classification result for each class.

Look at the number of correctly and incorrectly classified samples assigned to each class versus training iteration.

```
In[12]:= NetPlot[fitrecord, x, y, DataFormat → ClassPerformance]

         Correctly/incorrectly classified data
```







You have one plot for each class. In each plot, a solid line indicates the number of samples of this class that are correctly classified, and a dashed line indicates the number of samples incorrectly classified to this class.

### 4.2.3 Higher-Dimensional Classification

Classification in higher-dimensional problems, that is, when the dimension of the input pattern $x$ is higher than two, can be done in the same way as the two-dimensional problems. The main difference is that you can no longer illustrate the result with nice plots. Instead, you can view the data at various two-dimensional projections. It is also possible to look at how the data is distributed among the classes. This may be done using the commands in the *Neural Networks* package as illustrated next.

Load the *Neural Networks* package and the data.

```
In[1]:= <<NeuralNetworks`
```

```
In[2]:=  << threeclasses3D.dat;
```

The input patterns are placed in the matrix $x$ and the output in $y$.

Check the dimensions of the data.

```
In[3]:= Dimensions[x]
        Dimensions[y]

Out[3]= {40, 3}

Out[4]= {40, 3}
```

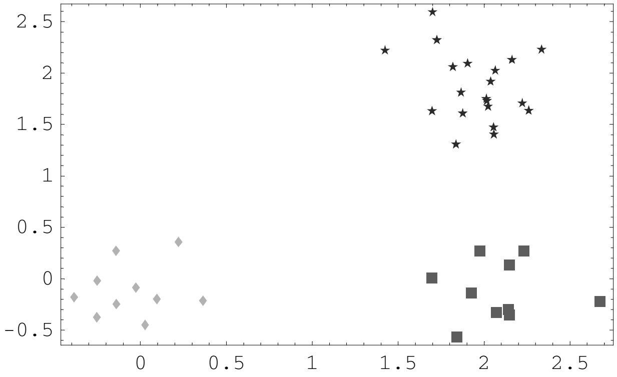
There are 40 data samples. The input matrix $x$ has three columns, which means that the data is in a three-dimensional space. The output $y$ also consists of three columns, which means that the perceptron should have three outputs, one for each class.

By analogy to the data in Section 3.4.1, Classification Problem Example, this data could correspond to (scaled values of) the age, weight, and height of children from three different groups.

The main difference compared to two-dimensional problems is that you cannot look at the data in the same way. It is, however, possible to look at projections of the data. To do that, you need a projection matrix of dimensions #inputs × 2.

Look at a projection of the data.

```
In[5]:= NetClassificationPlot[x . {{1,0},{0,1},{0,0}},y,SymbolStyle→
          {Hue[.5],Hue[.7],Hue[.9]}]
```

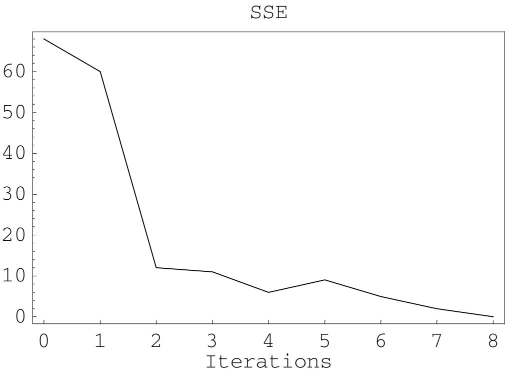


There are obviously 20 data samples of class two and ten of classes one and three.

You can now train a perceptron with this data set.

Train the perceptron.

```
In[6]:= {per, fitrecord} = PerceptronFit[x, y];
```



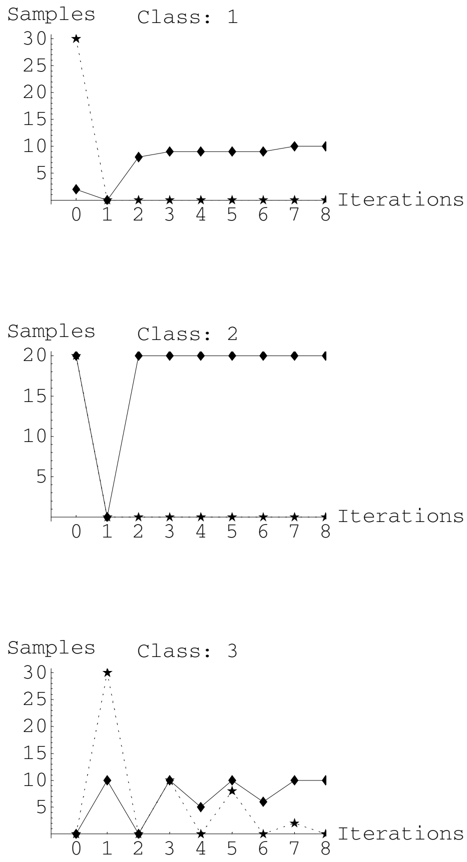
Success of the training depends on the initial weights of the perceptron. If you repeat the command, you will likely obtain slightly different results.

You can use the training record to investigate the training process.

Plot the number of correctly and incorrectly classified data vectors of each class.

```
In[7]:=  NetPlot[fitrecord, x, y, Intervals → 3]
```

```
Correctly/incorrectly classified data
```

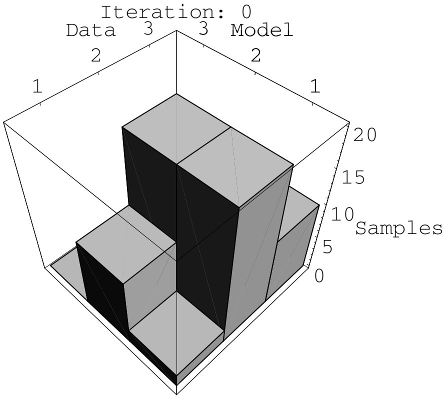


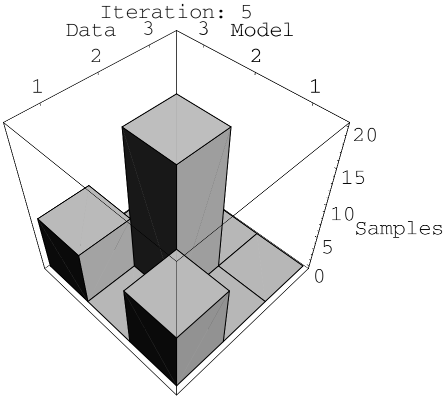
You can also illustrate the classification during training with a bar chart. The result is a graphics array.

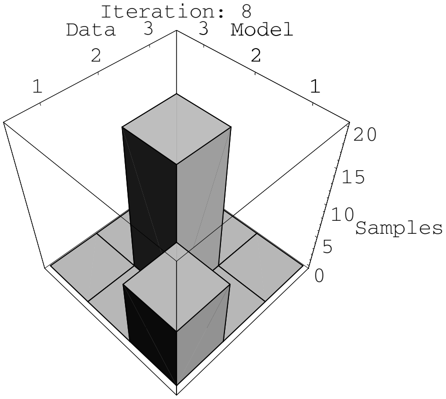
Check the evolvement of the classifier during the training.

```
In[8]:= NetPlot[fitrecord, x, y, Intervals → 5, DataFormat → BarChart]
```

```
Classification after
```





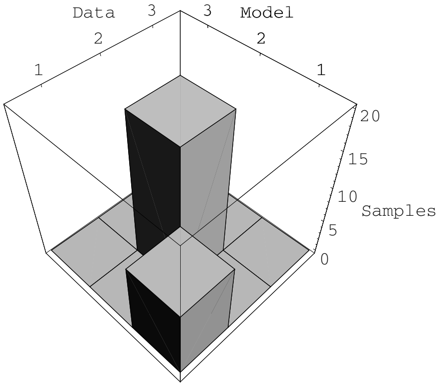


If you prefer an animation of the training progress, you can load `<<Graphics`Animation`` and then change the command to `Apply[ShowAnimation,NetPlot[fitrecord,x,y,Intervals→1, DataFormat→BarChart,DisplayFunction→Identity]]`.

If you are interested only in the end result, you submit the perceptron instead of the training record.

Look at only the end result.

```
In[9]:= NetPlot[per,x,y]
```

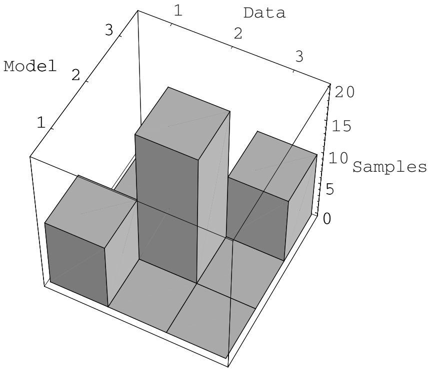


If the classification is perfect, then all samples should be on the diagonal of the three-dimensional bar chart. The size of the off-diagonal bars corresponds to the number of misclassified samples.

If you cannot see all the bars properly, you can repeat the command and change the viewpoint. This is most easily done by using the menu command 3D ViewPoint Selector.

Change the viewpoint.

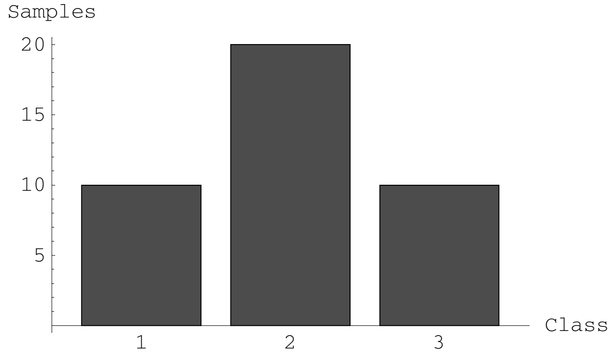
```
In[10]:= NetPlot[per,x,y,ViewPoint→{2.354, -4.532, 6.530}]
```



If the output $y$ is not supplied, the distribution between the classes according to the perceptron model is given.

Illustrate the classification without any output.

```
In[11]:= NetPlot[per, x, DataFormat → BarChart]
```



However, if you do not supply any output data, the graph cannot indicate which data samples are correctly and incorrectly classified. Instead, you can see only the distribution over the classes according to the perceptron.

## 4.3 Further Reading

The perceptron is covered in most books on neural networks, especially the following:

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, Macmillan, 1999.

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Reading, MA, Addison-Wesley, 1991.

# 5 The Feedforward Neural Network

This chapter describes FF neural networks, also known as backpropagation networks and multilayer perceptrons. Definitions, commands, and options are discussed in Section 5.1, Feedforward Network Functions and Options, and examples may be found in Section 5.2, Examples. A short tutorial introducing FF networks can be found in Section 2.5.1, Feedforward Neural Networks. Chapter 13, Changing the Neural Network Structure, describes how you can use the options and other ways to define more advanced network structures.

FF networks have a lot in common with those in Chapter 6, The Radial Basis Function Network. They are used for the same types of problems, and they use the same training algorithms (see Section 2.5.3, Training Feedforward and Radial Basis Function Networks).

The *Neural Networks* package supports the use of FF networks in three special types of problems, as follows:

- Function approximation
- Classification
- Modeling of dynamic systems and time series

This section illustrates the first two applications. Dynamic neural network models are described in Chapter 8, Dynamic Neural Networks. However, because the dynamic neural network models are based on FF networks, they will also be examined here.

The *Neural Networks* package offers several important features for FF networks, most of which are uncommon in other neural network software products. These features are listed here with links to places where more detailed descriptions are given.

**Initialization:** There are special initialization algorithms that give well-initialized neural networks. You can obtain an initialization with better performance from these than from one derived from a linear model. After initialization the performance is improved by the training.

**Fixed parameters:** You do not have to train all parameters. By keeping some of them fixed to values of your choice, you can obtain special model structures that are, for example, linear in some parameters. This is described in Section 13.2, Fixed Parameters.

**Different neuron activation function:** You can specify any nonlinear activation function for the neuron. This is described in Section 13.3, Select Your Own Neuron Function.

**Regularization and stopped search:** These techniques help you to obtain models that generalize better on new data. This is covered in Section 7.5, Regularization and Stopped Search.

**Linear models:** You can obtain linear models by specifying an FF network without hidden layers. The subsection Section 2.5.1, Feedforward Neural Networks, discusses why this might be a good choice.

**Linear model in parallel to the network:** You can choose to have a linear model in parallel to the neural network by setting the option `LinearPart` to `True` in `InitializeFeedForwardNet` of the FF network.

Several of these features make use of a combination of numeric and symbolic capabilities of *Mathematica*.

## 5.1 Feedforward Network Functions and Options

This subsection introduces the different commands to initialize, train, and evaluate FF networks. Examples using the commands can be found in Section 5.2, Examples.

### 5.1.1 InitializeFeedForwardNet

You initialize an FF network with `InitializeFeedForwardNet`.

| | |
|---|---|
| `InitializeFeedForwardNet[`*x*, *y*, *nh*, *opts*`]` | initializes an FF network based on the input data *x* and the output data *y* with the number of hidden neurons given by the list *nh* |

Initializing an FF network.

The returned network is an object with head `FeedForwardNet`, following the general format described in Section 3.2.3, Network Format. `FeedForwardNet` and `RBFNet` have one more replacement rule than the other network models. Its name is `NumberOfInputs` and it indicates how many inputs the network takes.

The number of inputs and outputs of the network do not need to be specified explicitly. They are instead extracted from the number of columns in the input and output data.

The argument *nh* should be a list of integers. The length of the list indicates the number of hidden layers in the network, and the integers indicate the number of neurons in each hidden layer. A linear model is obtained by setting it to an empty list, *nh*={}.

`InitializeFeedForwardNet` takes the following options.

| *option* | *default value* | |
|---|---|---|
| `LinearPart` | `False` | indicates if a linear model should be placed in parallel to the network |
| `Neuron` | `Sigmoid` | activation function in the neurons |
| `BiasParameters` | `True` | indicates if bias parameters should be included |
| `RandomInitialization` | `False` | indicates if the parameters should be randomly initialized; the default is to use a smart initialization |
| `Regularization` | `None` | indicates regularization in the least-squares fit of the linear parameters |
| `FixedParameters` | `None` | indicates if some parameters should be fixed and, therefore, excluded from the training |
| `InitialRange` | `1` | range of the uniform probability function if the parameters are initialized randomly |
| `OutputNonlinearity` | `None` | indicates if the output neuron should be nonlinear; for classification problems, `OutputNonlinearity→ Sigmoid` is recommended |
| `Compiled` | `True` | use compiled version |

Options of `InitializeFeedForwardNet`.

These are almost the same options for `InitializationRBFNet`, which is used to initialize RBF networks. The difference is the option `BiasParameters` with which you can obtain an FF network without the bias parameters indicated by *b* in Equation 2.7 in Section 2.5.1, Feedforward Neural Networks. Normally, you should include these parameters.

Another difference compared to InitializationRBFNet is that some of the default values of the options are different.

The parameters of the network can be initialized in three different ways, depending on the option RandomInitialization:

- False, which is the default. Then the parameters are initialized so that the slopes of neurons are placed within the domain of the input data. The exact initialization is still random, but the input data is used to choose a good range from which the parameters are chosen randomly. If there is no neuron in the output layer, then the linear parameters of the network are fitted to the output data using the least-squares algorithm. For most problems this gives a good initialization. If the model is overparameterized, that is, if it has more parameters than necessary, the least-squares step may give very large parameter values, which can give problems in the training. In such cases LinearParameters is an alternative to False.
- True, by which the parameters are initialized randomly from a uniform distribution. The range of the distribution can be set using the option InitialRange.
- LinearParameters, by which the positions of the neurons are initialized in the same way as if False is used. The linear parameters are randomly chosen. This can be a good alternative if the model is overparameterized and if you intend to use regularization or stopped search in the training.

The options Regularization and FixedParameters can be set at the initialization of the network or when the network is trained with NeuralFit. In Section 7.5, Regularization and Stopped Search, and Section 13.2, Fixed Parameters, you can learn how to use these options.

The default neuron function is Sigmoid, but you can use the option Neuron to change it to any other differentiable function. How this is done is shown in Section 13.3, Select Your Own Neuron Function.

Depending on the initial parameter values of the FF network, it will converge to different local minima in the training. Therefore, it is best to repeat the training a couple of times with the same neural network type but with different initializations. You get a new parameter initialization by repeating the command InitializeFeedForwardNet. Also, it should be noted that if you use the default option RandomInitialization→False, you get a partly random initialization.

### 5.1.2 NeuralFit

The initialized FF network is trained with `NeuralFit`. This command, which is also used for RBF networks and for dynamic networks, is described here with all its variants. Section 2.5.3, Training Feedforward and Radial Basis Function Networks, describes the algorithms in some detail.

| | |
|---|---|
| `NeuralFit[`*net*`,` *x*`,` *y*`]` | trains the model *net* using input data *x* and output data *y* with a default number of training iterations (30) |
| `NeuralFit[`*net*`,` *x*`,` *y*`,` *xv*`,` *yv*`]` | trains the model *net* using input data *x* and output data *y* with validation data *xv*, *yv* submitted |
| `NeuralFit[`*net*`,` *x*`,` *y*`,` *iterations*`]` | normal training but with the number of training iterations indicated |
| `NeuralFit[`*net*`,` *x*`,` *y*`,` *xv*`,` *yv*`,` *iterations*`]` | normal training but with the number of training iterations indicated and with submitted validation data |

Training an FF network.

`NeuralFit` returns a list of two variables. The first one is the trained FF network, and the second is a record containing information about the training.

An existing network can be submitted for more training by setting *net* equal to the network or its training record. The advantage of submitting the training record is that the information about the earlier training is combined with the additional training.

During the training, intermediate results are displayed in an automatically created notebook. After each training iteration the following information is displayed:

- Training iteration number
- The value of the RMSE
- For validation data submitted in the call, the RMSE value for this second data set is also displayed
- The step length control parameter of the minimization algorithm ($\lambda$ or $\mu$), which is described in Section 2.5.3, Training Feedforward and Radial Basis Function Networks

At the end of the training, a plot is displayed with RMSE as a function of iteration.

Using the options of `NeuralFit`, as described in Section 7.7, Options Controlling Training Results Presentation, you can change the way the training results are presented.

At the end of the training you often receive different warning messages. These give you information on how the training performed. By looking at the performance plot, you can usually tell whether more training is required. If the plot has not flattened out toward the end of the training, then you should consider applying more training iterations. This can be done by resubmitting the trained network to `NeuralFit` so that you do not have to initiate training anew.

There are many options to `NeuralFit` that can be used to modify its behavior. They are described in Section 7.1, NeuralFit.

A derived FF network can be applied to new inputs using function evaluation. The result given by the network is its estimate of the output.

| | |
|---|---|
| *net* `[` *x* `]` | evaluates *net* on the input vector *x* |

Function evaluation of a feedforward network.

The input argument *x* can be a vector containing one input sample, or a matrix containing one input sample on each row.

The function evaluation has one option.

| *option* | *default value* | |
|---|---|---|
| `Compiled` | `True` | indicates that a compiled version of the evaluation rule should be used |

Option of the network evaluation rule.

### 5.1.3 NetInformation

Information about an FF network is presented in a string by the function `NetInformation`.

| | |
|---|---|
| `NetInformation[`*fdfrwrd*`]` | gives information about an FF network |

The `NetInformation` function.

### 5.1.4 NetPlot

The command `NetPlot` can be used to illustrate the derived FF network or the evolution of the training. Depending on how the option `DataFormat` is set, the command can be used in very different ways.

| | |
|---|---|
| `NetPlot[`*fdfrwrd*`,`*x*`,`*y*`]` | evaluates the network on the supplied data |
| `NetPlot[`*fitrecord*`,`*x*`,`*y*`]` | evaluates the training of a net using supplied data |

Illustrate models and training of models.

If the input dimension is one or two, the default is to plot the estimated function in the range of the supplied data. In the one-dimensional case, the data is also shown.

When `NetPlot` is applied to an FF network, it takes the following options.

| *option* | *default value* | |
|---|---|---|
| `OutputNonlinearity` | `Automatic` | this option only takes effect when `DataFormat→` `BarChart;` it can be used to change the nonlinearity in the output layer; for classification problems `OutputNonlinearity→` `UnitStep` is recommended |
| `DataFormat` | `Automatic` | if a model is submitted, gives the values of the hidden neurons when the model is evaluated on the data; if a training record is submitted, gives a plot of the parameters of the network versus training iterations |
| `Intervals` | `5` | intervals between plots if a training record is submitted |
| `Compiled` | `True` | use compiled version |

Options of NetPlot.

In addition to these, you can submit options to modify the graphical output. Depending on the chosen option for DataFormat, the graphic is created by BarChart, BarChart3D, MultipleListPlot, ListPlot, Plot3D, or Histogram.

If the input dimension is higher than two, then the default is to plot the numerical values of the hidden neurons versus the data. This can be obtained also in one- and two-dimensional problems by choosing DataFormat→HiddenNeurons. Notice, however, that plotting the neurons versus data only makes sense if the input signal vectors are placed in some kind of order.

The option DataFormat takes any of the following values:

- FunctionPlot: plots the mapping using the range of the supplied data. It can only be used if the model has one or two inputs.
- NetOutput: plots the network output versus the given output. A perfect fit corresponds to a straight line with slope 1 through the origin.
- ErrorDistribution: gives a histogram of the errors when the model is applied to submitted data. You can modify the presentation of the result using any of the options applicable to Histogram.
- HiddenNeurons: gives the values of the hidden neurons when the model is evaluated on the data. This function makes most sense when it is applied to dynamic models.
- ParameterValues: plots the parameters versus the training iterations. This is only possible for the training record.
- LinearParameters: plots the parameters of the linearization versus data. This function makes most sense when it is applied to dynamic models.

The following three possibilities are primarily intended for classification models:

- Classifier: shows the borders between different classes. It can only be used with two-input models.
- BarChart: illustrates the classification result with bar charts.
- ClassPerformance: plots the improvement of the classification for each class versus the number of training iterations. Correctly classified samples are marked with diamonds and solid lines, while incorrectly classified samples are indicated with stars and dashed lines. This is only possible for the training record.

If you submit a training record instead of an FF network, then depending on which option is given, you obtain a graphic array of the corresponding results as a function of the number of training iterations. For a large number of iterations, it is advisable to set the option `Intervals` to a larger integer, thus controlling the size of the length of the graphic array.

Examples where `NetPlot` is used to evaluate FF networks are given in Section 5.3, Classification with Feedforward Networks and Section 5.2, Examples.

### 5.1.5 LinearizeNet and NeuronDelete

The commands `LinearizeNet` and `NeuronDelete` modify the structure of an existing network.

In many situations, it is interesting to linearize a nonlinear function at some point. FF networks can be linearized using `LinearizeNet`.

| | |
|---|---|
| `LinearizeNet[`*fdfrwrd*`,` *x*`]` | linearizes the FF network at *x* |

Linearizing a feedforward network.

`LinearizeNet` returns a linear model in the form of an FF network without any hidden neurons as described in Section 2.5.1, Feedforward Neural Networks.

The point of the linearization, *x*, should be a list of real numbers of length equal to the number of inputs of the neural network.

The linear network corresponds to a first-order Taylor expansion of the original network in the linearization point.

Sometimes it might be of interest to remove parts of an existing network. `NeuronDelete` can be used to remove outputs, inputs, hidden neurons, or a linear submodel.

You can also remove individual parameters by setting their numerical values to zero and excluding them from the training, as described in Section 13.2, Fixed Parameters.

| | |
|---|---|
| `NeuronDelete[`*net*, *pos*`]` | deletes the neurons indicated<br>with *pos* in an existing network *net* |
| `NeuronDelete[`*net*, *pos*, *x*`]` | deletes the neurons indicated with *pos* in an<br>existing network *net* where input data is supplied<br>and the remaining network parameters are adjusted |

Deleting neurons from an existing network.

The argument *pos* indicates which part of the network should be deleted in the following way:

{0, 0}: removes the linear submodel.

{0, *m*}: removes input *m.*

{*n*, *m*}: removes neuron *m* in hidden layer *n.*

{*n*, *m*}: removes output *m* if *n* == number of hidden layers + 1.

The argument *pos* can also be a list where each element follows these rules.

`NeuronDelete` can be used to obtain the values of the hidden neurons of a network; if all outputs are removed, then a network is returned with outputs equal to the last hidden layer of the initial network. The output nonlinearity is set to the neuron function used in the initial network.

If input data is submitted, then the parameters of the layer following the removed neuron are adjusted so that the new network approximates the original one as well as possible. The least-squares algorithm is used for this.

There is no adjustment of the parameters if an output is removed.

If a neuron in the last hidden layer is removed, then the parameters in the linear submodel are also included in the parameter adjustment. If the linear submodel is removed, then the parameters in the last layer are adjusted.

### 5.1.6 SetNeuralD, NeuralD, and NNModelInfo

The commands `SetNeuralD`, `NeuralD`, and `NNModelInfo` are intended primarily for internal use in the package, but they might be useful if you want to perform more special operations.

`SetNeuralD` and `NeuralD` help you to compute the derivative of an FF or RBF network and they might be useful if you write your own training algorithm. `SetNeuralD` does not return any object but produces optimal code for the specific network structure, which then is used by `NeuralD`. Therefore, each time the network structure is changed, `SetNeuralD` has to be called prior to `NeuralD`.

| | |
|---|---|
| `SetNeuralD[`*net*, *opts*`]` | sets `NeuralD` to a function to compute the derivative of *net* with respect to its parameters |

Generating optimal code for `NeuralD`, which is used to compute the derivative of a network.

| *option* | *default* | |
|---|---|---|
| `Compiled` | `True` | indicates if `NeuralD` should be compiled |
| `FixedParameters` | `None` | points out parameters to be excluded in the same way as for `NeuralFit` |

Options of `SetNeuralD`.

The numerical derivative of a network is obtained using `NeuralD`.

| | |
|---|---|
| `NeuralD[`*net*, *x*`]` | computes the derivative of *net* at the input vectors *x* |

Computation of the derivative of a network.

`NeuralD` can be applied to a matrix with one numerical input vector on each row. The output is a three-dimensional list: the first level indicates the data, the second level has one component for each network output, and the third level has one component for each parameter.

Notice that you only have to call `SetNeuralD` once for a specific network. The current parameter values, submitted in the argument *net,* are used each time `NeuralD` is called.

You may use `SetNeuralD` and `NeuralD` in any of your own training algorithms in the following way. First the network structure is determined. Then `SetNeuralD` is called to obtain the optimized code. The actual training often consists of a loop containing the computation of the derivative and a parameter update. You use `NeuralD` to obtain the derivative at the parameter values given in the network in the call. Section 7.9, Writing Your Own Training Algorithms, illustrates the use of `SetNeuralD` and `NeuralD`.

To save computer time, and since `NeuralD` is intended to be used inside a loop, there is no security check of the input arguments. Therefore, you have to include these yourself, preferably outside the loop.

`NNModelInfo` gives you exactly the specification needed to create a new neural network with the same structure. This specifies the network with the exception of the parameter values.

| | |
|---|---|
| `NNModelInfo[`*fdfrwrd*`]` | extracts the specification of the FF network |

Structure specification about an FF network.

`NNModelInfo` returns a list of four variables that contains the following: number of inputs, number of outputs, number of hidden neurons, and a list of options used when the model was defined.

## 5.2 Examples

This section contains some simple examples that illustrate the use of FF networks. Since many commands and options are identical for FF and RBF networks, more advanced examples that illustrate common features are found in Chapter 7, Training Feedforward and Radial Basis Function Networks; Chapter 8, Dynamic Neural Networks; and Chapter 12, Application Examples.

### 5.2.1 Function Approximation in One Dimension

Consider a function with one input and one output. First, output data is generated by evaluating the given function for a given set of input data. Then, the FF network will be trained with the input-output data set to approximate the given function. You can run the example on different data sets by modifying the commands generating the data.

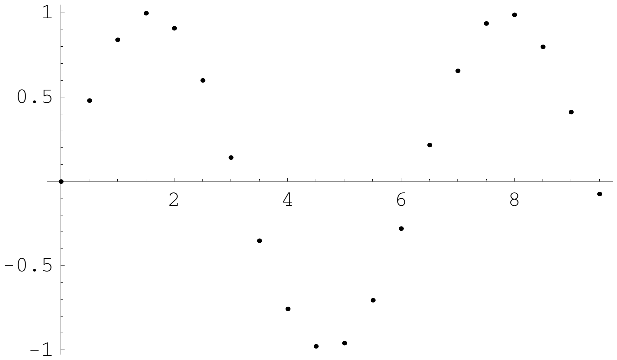
Read in the *Neural Networks* package and a standard add-on package.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

This example will involve an input-output one-dimensional data set of a sinusoidal function. The variable *Ndata* indicates the number of training data generated. Change the following data generating commands if you want to rerun the example on other data. It is always a good idea to look at the data. This is especially easy for one-dimensional problems.

Generate and look at the data.

```
In[3]:= Ndata = 20;
        x = Table[10 N[{i / Ndata}], {i, 0, Ndata - 1}];
        y = Sin[x];
        ListPlot[AppendRows[x, y]]
```



The training data consist of the input data *x* and the output *y*.

Consider first a randomly initialized FF network, with four hidden neurons in one hidden layer. Although random initialization is not generally recommended, it is used here only for purposes of illustration.

Initialize an FF network with four neurons.

```
In[7]:= fdfrwrd = InitializeFeedForwardNet[x, y, {4}, RandomInitialization → True]

Out[7]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 25, 46},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

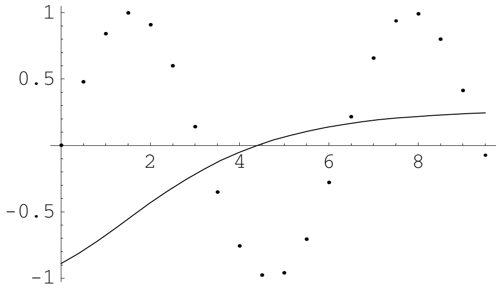
Find some information about the network.

```
In[8]:= NetInformation[fdfrwrd]

Out[8]= FeedForward network created 2002-4-3 at 13:25.
          The network has 1 input and 1 output. It consists of 1 hidden
          layer with 4 neurons with activation function of Sigmoid type.
```

The randomly initialized network is a description of a function, and you can look at it before it is trained. This can be done using `NetPlot`.

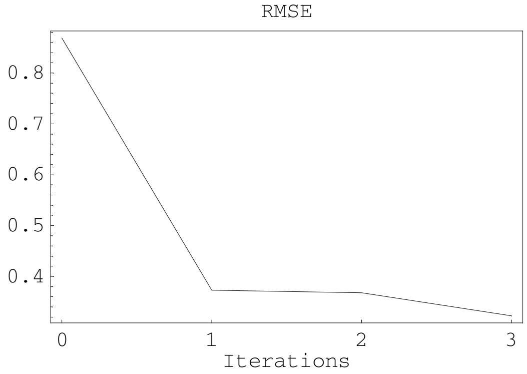
Look at the initialized FF network.

```
In[9]:= NetPlot[fdfrwrd, x, y]
```



Fit the network to the data.

```
In[10]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 3];
```



Often, a warning appears stating that training was not completed for the iterations specified. This is equivalent to saying that the parametric weights did not converge to a point minimizing the performance index, RMSE. This is not an uncommon occurrence, especially for network models involving a large number of parameters. In such situations, by looking at the performance plot you can decide whether additional training would improve performance.

The trained FF network can now be used in any way you would like. For example, you can apply it to new input data.

Evaluate the FF model at a new input value.

```
In[11]:= fdfrwrd2[{1.5}]

Out[11]= {1.21137}
```

Evaluate the FF model at several new input values.

```
In[12]:= fdfrwrd2[{{1.5}, {0.3}, {2.5}}]

Out[12]= {{1.21137}, {0.0976301}, {0.478028}}
```

You can also obtain a *Mathematica* expression of the FF network by applying the network to a list of symbols. The list should have one component for each input of the network.

Obtain an expression for the FF network in the symbol *xx*.

```
In[13]:= Clear[xx]
         fdfrwrd2[{xx}]
```

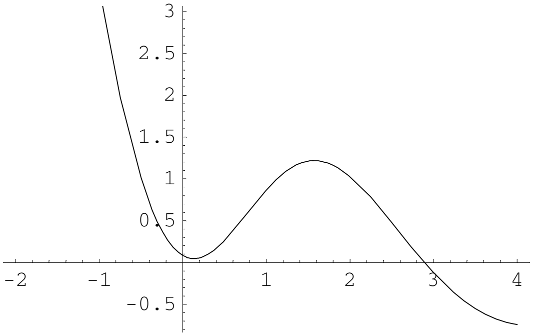
Out[14]=

$$\left\{-412.23+\frac{162.243}{1+e^{0.693347-0.954841\,\mathrm{xx}}}+\frac{250.898}{1+e^{0.722195-0.737868\,\mathrm{xx}}}+\frac{82.5833}{1+e^{0.80038+0.780475\,\mathrm{xx}}}+\frac{356.662}{1+e^{-0.860144+0.831306\,\mathrm{xx}}}\right\}$$

You can plot the function of the FF network on any interval of your choice.

Plot the FF network on the interval {−2, 4}.

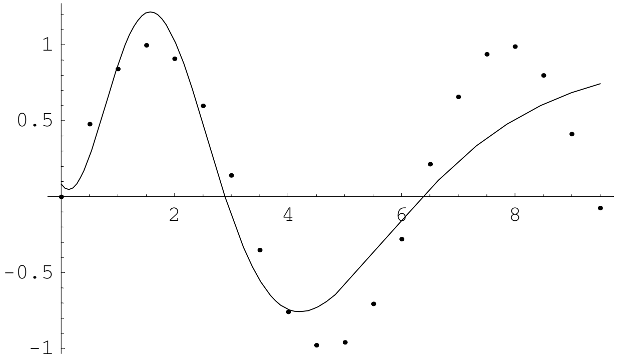
```
In[15]:= Plot[fdfrwrd2[{x}], {x, -2, 4}]
```



If you use NetPlot then you automatically get the plot in the range of your training data. The command relies on the *Mathematica* command Plot, and any supported option may be used.

Plot the estimated function and pass on some options to Plot.

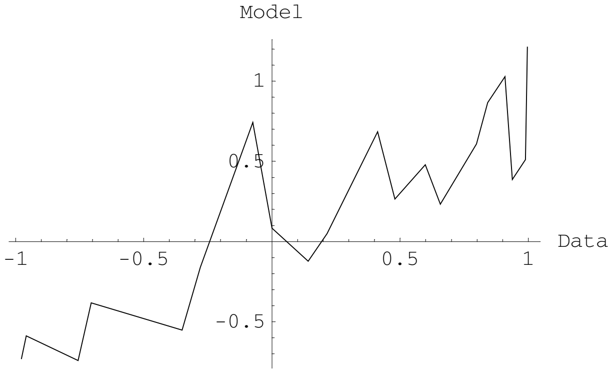
```
In[16]:= NetPlot[fdfrwrd2, x, y, PlotPoints → 5, PlotDivision → 20]
```



By giving the option DataFormat→NetOutput you obtain a plot of the model output as a function of the given output. If the network fits the data exactly, then this plot shows a straight line with unit slope through the origin. In real applications you always have noise on your measurement, and you can only expect an approximate straight line if the model is good.

Plot the model output versus the data output.

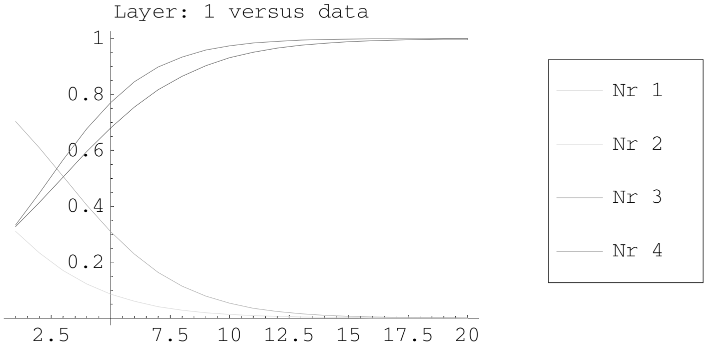
```
In[17]:= NetPlot[fdfrwrd2, x, y, DataFormat → NetOutput]
```



By giving the option `DataFormat→HiddenNeurons` in the call to `NetPlot`, you obtain a plot of the values of the hidden neurons versus the data. Such a plot may indicate if the applied network is unnecessarily large. If some hidden neurons have very similar responses, then it is likely that the network may be reduced to a smaller one with fewer neurons.

Look at the values of the hidden neurons and specify colors.

```
In[18]:= NetPlot[fdfrwrd2, x, y, DataFormat → HiddenNeurons,
          PlotStyle → {Hue[.0], Hue[.2], Hue[.4], Hue[.6]},
          PlotLegend → Map["Nr " <> ToString[#] &, Range[Length[fdfrwrd2[[1, 1, 1, 1]]]]]]
```

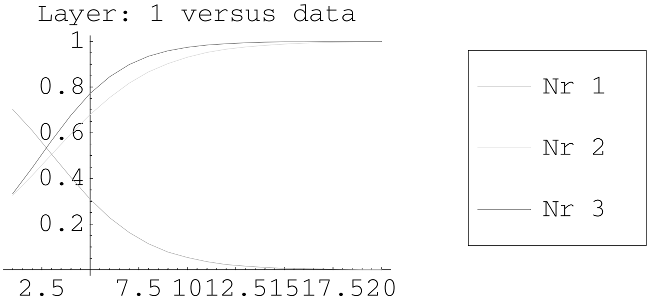


If some hidden neurons give similar outputs, or if there is a linear relation between them, then you may remove some of them, keeping the approximation quality at the same level. The number of any such neurons can be identified using the legend. This might be of interest in a bias-variance perspective as described in Section 7.5, Regularization and Stopped Search.

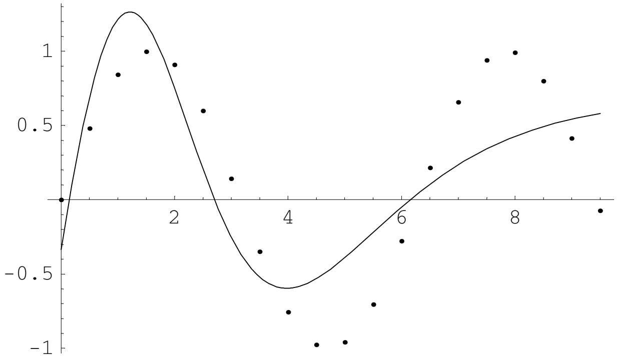
Remove the second hidden neuron, and look at the neurons and the approximation of the function.

```
In[19]:= fdfrwrd3 = NeuronDelete[fdfrwrd2, {1, 2}, x];
```

```
In[20]:= NetPlot[fdfrwrd3, x, y, DataFormat → HiddenNeurons,
          PlotStyle → {Hue[.2], Hue[.4], Hue[.6]},
          PlotLegend → Map["Nr " <> ToString[#] &, Range[Length[fdfrwrd3[[1, 1, 1, 1]]]]]]
```



```
In[21]:= NetPlot[fdfrwrd3, x, y]
```



Note that if you re-evaluate the example, then you may have to delete a different neuron due to the randomness in the initialization.

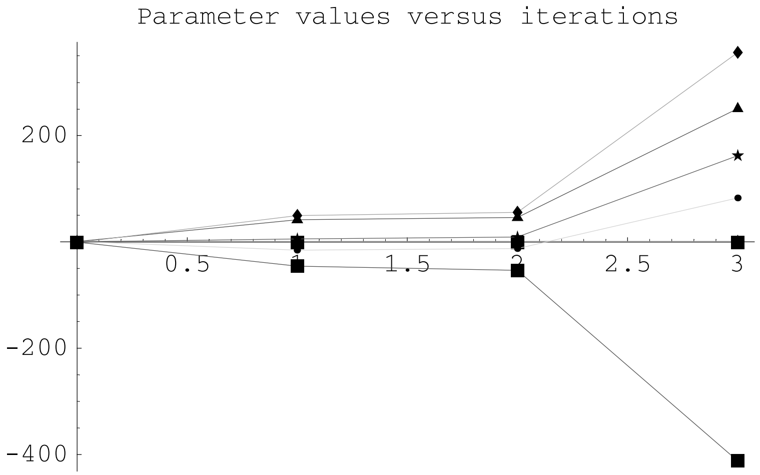
By removing the output of the network you obtain a new network with outputs equal to the hidden neurons of the original network.

```
In[22]:= NeuronDelete[fdfrwrd2, {2, 1}]

       NeuronDelete::NewOutputLayer :
        All outputs have been deleted. The second-to-last layer becomes the new output.

Out[22]= FeedForwardNet[{{w1}}, {AccumulatedIterations → 3,
           CreationDate → {2002, 4, 3, 13, 25, 46}, Neuron → Sigmoid,
           FixedParameters → None, OutputNonlinearity → Sigmoid, NumberOfInputs → 1}]
```

You can use `NetPlot` to evaluate the training of the network. This is done by applying it to the training record, which was the second argument of `NeuralFit`. Depending on the option `DataFormat`, the result is presented differently.

Look at how the parameter values change during the training.

```
In[23]:= NetPlot[fitrecord, x, y, PlotStyle → {Hue[.0], Hue[.2], Hue[.4], Hue[.6]}]
```

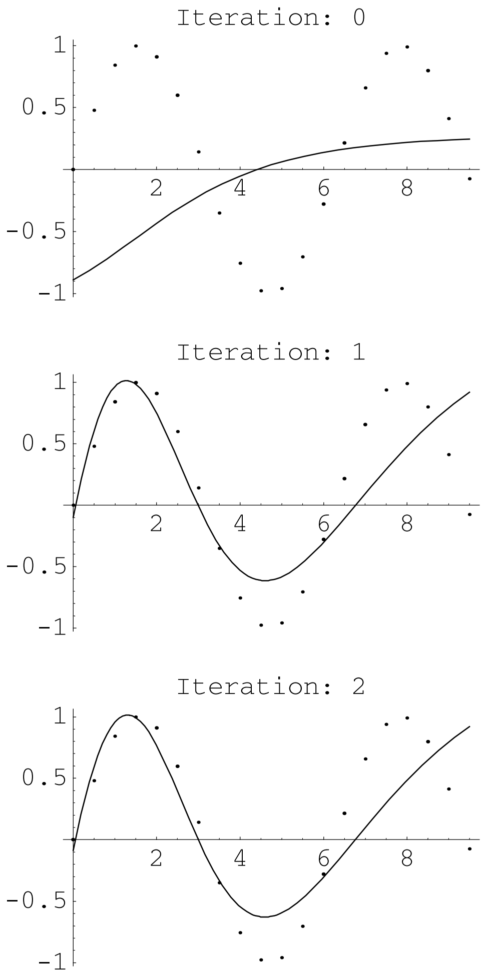


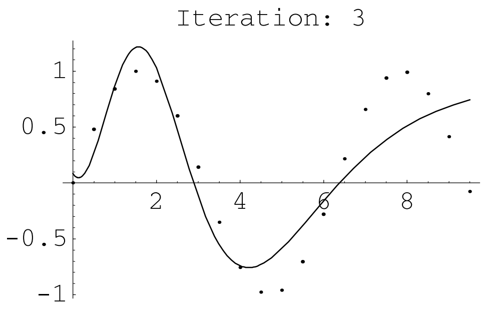
Often the parameter values increase during the training. From such a plot you may get some insights about why the parameter values did not converge in the training, although the derived network performs well.

Look at the function approximation after each training iteration.

```
In[24]:= NetPlot[fitrecord, x, y, Intervals → 1, DataFormat → FunctionPlot]
```

```
Function estimate after
```





If you prefer an animation of the training progress, you can load <<Graphics`Animation` and then change the command to Apply[ShowAnimation,NetPlot[fitrecord,x,y,Intervals→1, DataFormat→FunctionPlot,DisplayFunction→Identity]].

### 5.2.2 Function Approximation from One to Two Dimensions

In this example a function with one input and two outputs will be considered. The only difference from the previous example is that there are two outputs instead of one.

Read in the *Neural Networks* package and a standard add-on package.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

Load the data.

```
In[3]:= << one2twodimfunc.dat;
```

The input is placed in *x* and the output in *y*.

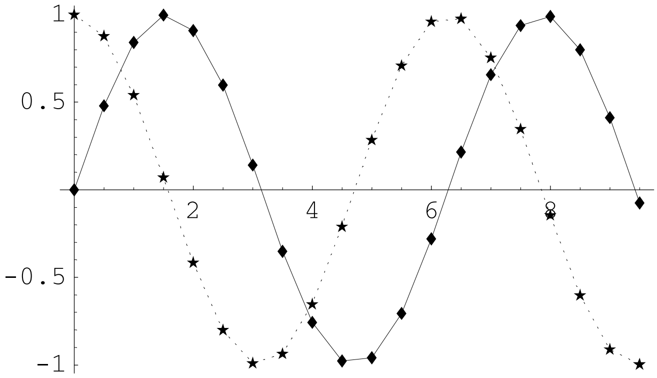
Check the dimensions of the data.

```
In[4]:= Dimensions[x]
        Dimensions[y]

Out[4]= {20, 1}

Out[5]= {20, 2}
```

There are 20 data samples, one input, and two outputs.

Look at the data; some transformation is necessary.

```
In[6]:= << Graphics`MultipleListPlot`;
        temp = Map[AppendRows[x, Transpose[{#}]] &, Transpose[y]];
        Apply[MultipleListPlot, Flatten[{temp, PlotJoined → True}, 1]]
```



The plot shows the two outputs versus the input.

The origin of this data is artificial; however, you can imagine a problem setting like in Section 3.4.2, Function Approximation Example, with the change that two variables (the outputs) depend on the variable *x* (the input).

Initialize and train an FF network with two outputs to approximate the two-dimensional output. The number of inputs and outputs does not need to be specified, since this information is extracted from the dimensions of the supplied data matrices.

Initialize an FF network with four neurons.

```
In[9]:= fdfrwrd = InitializeFeedForwardNet[x, y, {4}]

Out[9]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 27, 25},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

Find some information about the network.

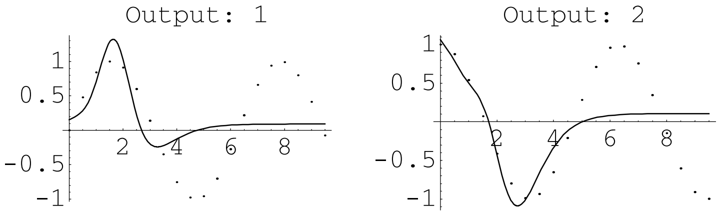
```
In[10]:= NetInformation[fdfrwrd]
```

```
Out[10]= FeedForward network created 2002-4-3 at 13:27. The
           network has 1 input and 2 outputs. It consists of 1 hidden
           layer with 4 neurons with activation function of Sigmoid type.
```

So far, the network has only been initialized. It can be interesting to look at the initialization before the training.

Look at the initialized FF network.

```
In[11]:= NetPlot[fdfrwrd, x, y]
```

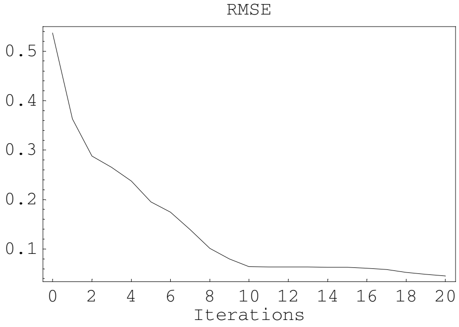


Notice that already the initialization is quite good, something not too unusual for a default initialization. You can repeat the initialization setting `RandomInitialization→True` to see the difference.

Now train the initialized FF network.

Fit the network to the data.

```
In[12]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 20];
```



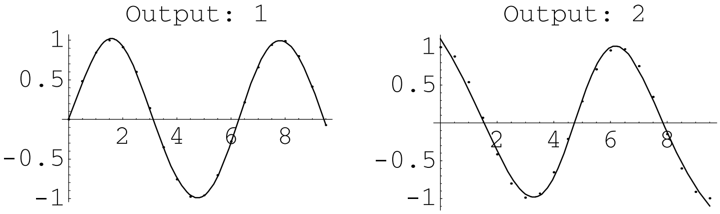
The FF network with two outputs can be evaluated on data in the same way as a network with only one output. The difference is that you obtain two output values now.

Evaluate the FF network on one input data sample.

```
In[13]:= fdfrwrd2[{2.}]

Out[13]= {0.920756, -0.369678}
```

Look at the result with the fitted FF network.

```
In[14]:= NetPlot[fdfrwrd2, x, y]
```



### 5.2.3 Function Approximation in Two Dimensions

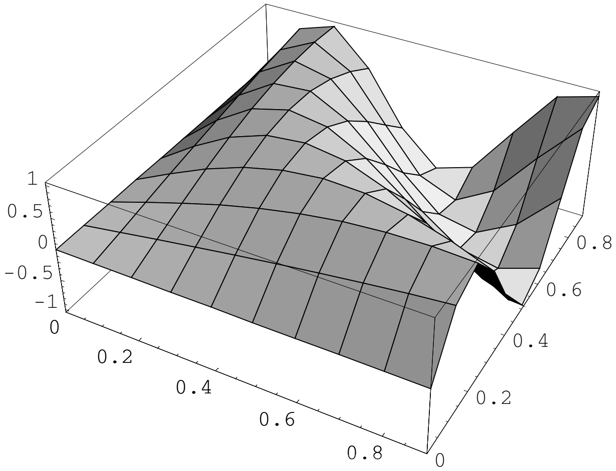
An FF network can have any number of inputs and outputs. In the previous two examples there was only one input. Here is an example with two inputs.

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

Generate data and look at the function.

```
In[2]:= Ndata = 10;
        x = Table[N[{i / Ndata, j / Ndata}], {i, 0, Ndata - 1}, {j, 0, Ndata - 1}];
        y = Map[Sin[10. #[[1]] #[[2]]] &, x, {2}];
        ListPlot3D[y, MeshRange → {{0, 0.9}, {0, 0.9}}];
        x = Flatten[x, 1];
        y = Transpose[{Flatten[y]}];
```



The training data is placed in *x* and *y*, where *x* is the input data and *y* is the output data.

You can modify the example by changing the function generating the data and by changing the number of neuron basis functions in the following initialization. Notice that you will obtain slightly different results even if you repeat the example without any changes at all. This is due to the randomness in the initialization algorithm of the FF network.

Check the dimensions of the data.

```
In[8]:= Dimensions[x]
        Dimensions[y]

Out[8]= {100, 2}

Out[9]= {100, 1}
```

There are 100 input-output data pairs with two-dimensional inputs and one-dimensional outputs.

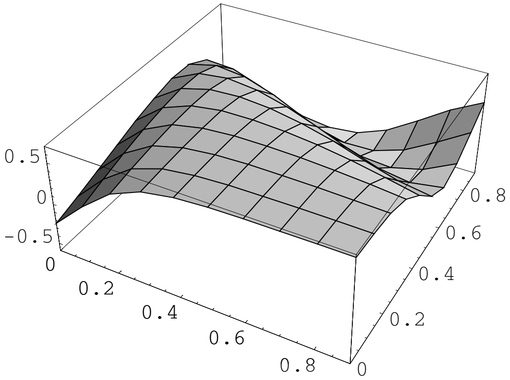
Initialize an FF network with four neurons.

```
In[10]:= fdfrwrd = InitializeFeedForwardNet[x, y, {4}]

Out[10]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 27, 54},
           OutputNonlinearity → None, NumberOfInputs → 2}]
```

You can apply the initialized network to the input data and plot the network output. Compare the result with the true function in the previous plot.

Look at the initialized FF network.

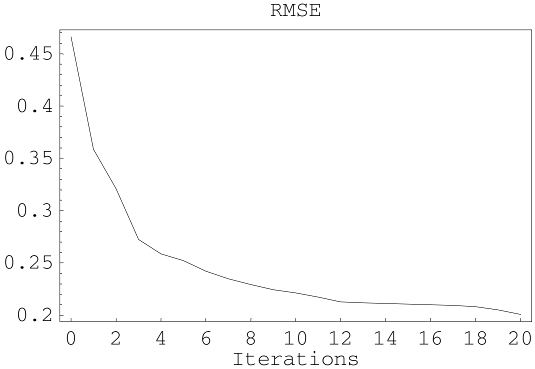
```
In[11]:= y2 = fdfrwrd[x];
         ListPlot3D[Partition[Flatten[y2], Ndata], MeshRange → {{0, 0.9}, {0, 0.9}}]
```



So far the network has only been initialized. Now it is time to train it.

Fit the FF network to the data.

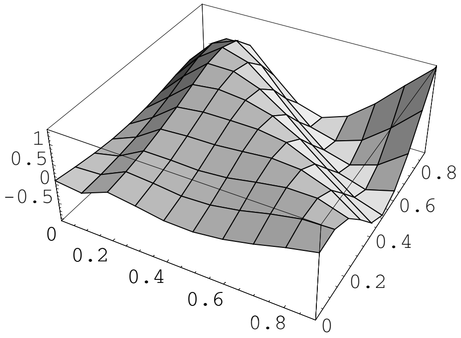
```
In[13]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 20];
```



You can look at the result by evaluating the trained network at the input data points as follows.

Look at the result with the fitted FF network.

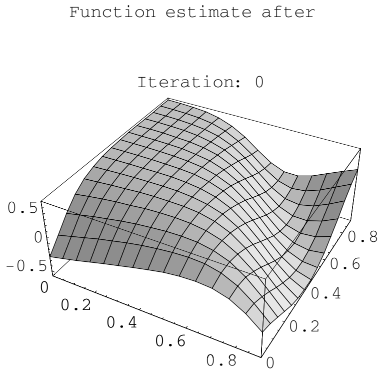
```
In[14]:= y2 = fdfrwrd2[x];
         ListPlot3D[Partition[Flatten[y2], Ndata], MeshRange → {{0, 0.9}, {0, 0.9}}]
```

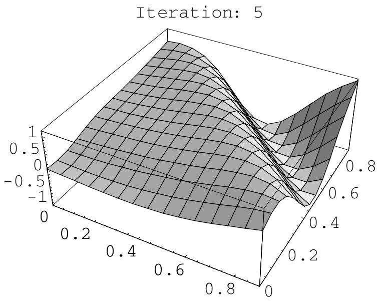


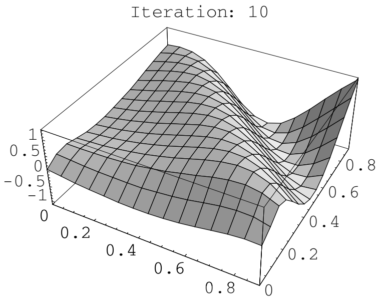
Notice that there are usually several local minima. If you repeat the initialization and the training, you obtain different results.

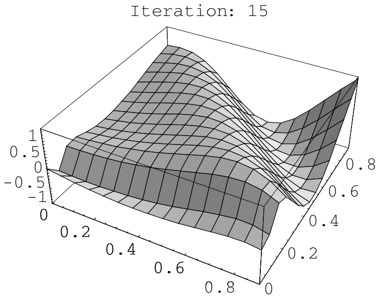
The plot does not, however, show how the function looks in between the data points. By using `NetPlot`, which gives a plot based on the *Mathematica* command, `Plot3D`, you obtain a plot with any number of evaluation points. If you apply `NetPlot` to the training record, you obtain a graphics array showing the evolution of the function approximation during the training.

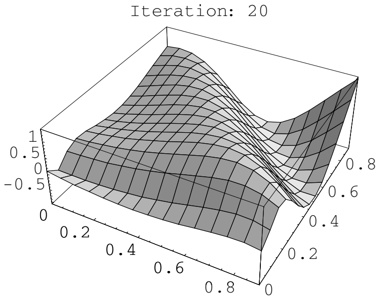
```
In[16]:= NetPlot[fitrecord, x, y, DataFormat → FunctionPlot]
```




Iteration: 5
1
0.5
0
-0.5
-1
0
0.2
0.4
0.6
0.8
0
0.2
0.4
0.6
0.8


Iteration: 10
1
0.5
0
-0.5
-1
0
0.2
0.4
0.6
0.8
0
0.2
0.4
0.6
0.8

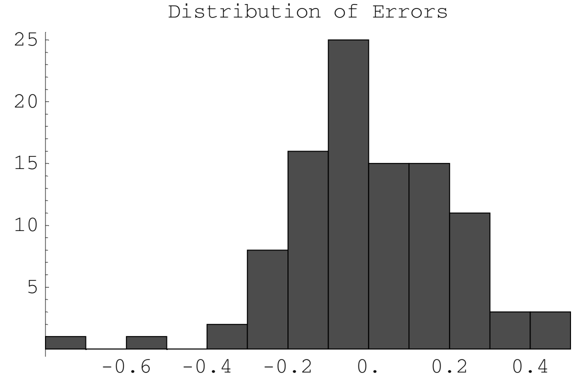

Iteration: 15
1
0.5
0
-0.5
-1
0
0.2
0.4
0.6
0.8
0
0.2
0.4
0.6
0.8


Iteration: 20
1
0.5
0
-0.5
0
0.2
0.4
0.6
0.8
0
0.2
0.4
0.6
0.8

If you prefer an animation of the training progress, you can load `<<Graphics`Animation`` and then change the command to `Apply[ShowAnimation,NetPlot[fitrecord,x,y,Intervals→1, DataFormat→FunctionPlot,DisplayFunction→Identity]]`.

You can obtain a histogram of the errors between the network output and the true output with `NetPlot` by choosing `DataFormat→ErrorDistribution`. This might help you to find outliers in the data and to explain if something goes wrong.

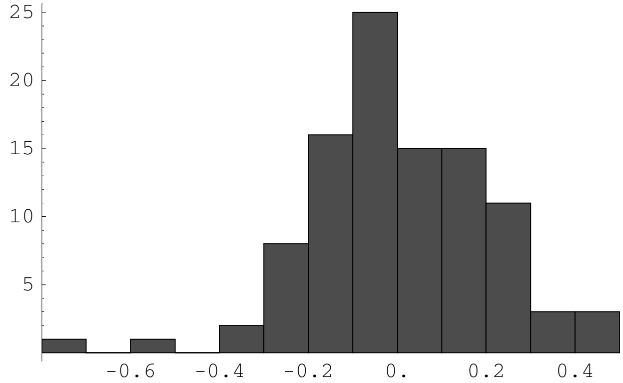
```
In[17]:=  NetPlot[fdfrwrd2, x, y, DataFormat → ErrorDistribution]
```



Each bar shows the number of samples given an estimation error within the borders of the bars.

Of course, you can also obtain the previous plot by using the command `Histogram` in the following way.

```
In[18]:=  Histogram[Flatten[y - fdfrwrd2[x]]]
```



## 5.3 Classification with Feedforward Networks

In this section a small example is given that shows how FF networks can be used for classification.

Read the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

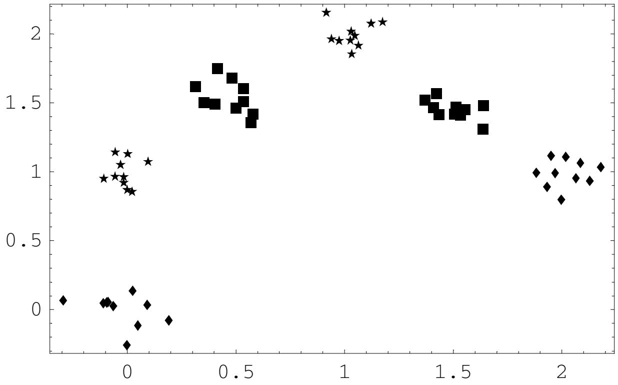
Load the data consisting of three classes divided into two clusters each. The data distribution is contained in *x* (input data), and the class indication is in *y* (output data). The data format is described in Section 3.2, Package Conventions.

Load the data vectors and output indicating class.

```
In[2]:= << vqthreeclasses.dat;
```

Look at the data.

```
In[3]:= NetClassificationPlot[x, y]
```



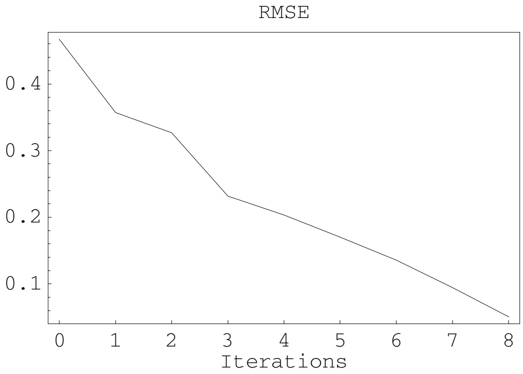
In classification problems it is important to have a differentiable nonlinearity at the output of the FF network model. The purpose of the nonlinearity is to ensure that the output values stay within the range of the different classes. That is done using the option `OutputNonlinearity`. Its default is `None`. Set it to `Sigmoid` so that its saturating values are 0 and 1, exactly as the output data of the classes. Note that the sigmoid never reaches exactly 0 and 1; this is in most problems of no practical importance.

Initialize an FF network.

```
In[4]:=  fdfrwrd = InitializeFeedForwardNet[x, y, {6}, OutputNonlinearity → Sigmoid]

Out[4]=  FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 28, 54},
           OutputNonlinearity → Sigmoid, NumberOfInputs → 2}]
```

Train the initialized FF network.

```
In[5]:=  {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 8];
```



The trained classifier can now be used on the input data vectors.

Classify two data vectors.

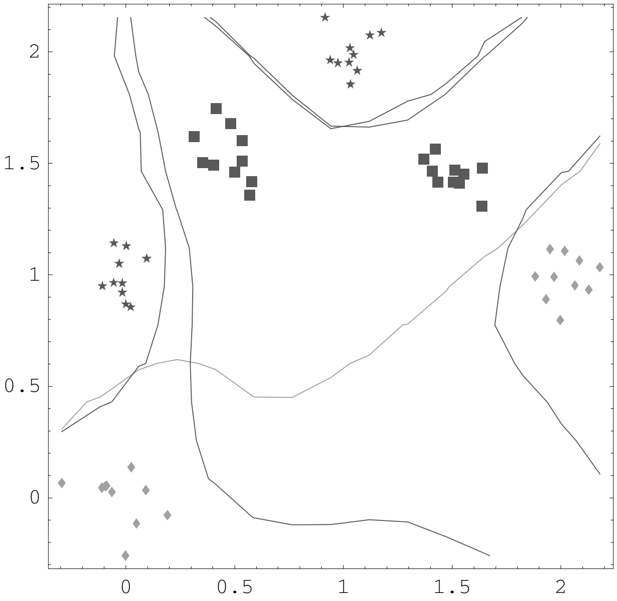
```
In[6]:=  fdfrwrd2[{{0, 0.1}, {2, 1}}]

Out[6]=  {{0.975454, 0.0430855, 0.0193727}, {0.958409, 0.0196418, 0.0360199}}
```

The data vectors are assigned to the class with the largest output value. If several outputs give large values, or if none of them do, then the classifier is considered to be highly unreliable for the data used.

The performance of the derived classifier can be illustrated in different ways using `NetPlot`. By the choice of the option `DataFormat`, you can indicate the type of plot you want. If the data vectors are of dimension two, as in this example, nice plots of the classification boundaries can be obtained.

Plot classification borders together with the data.

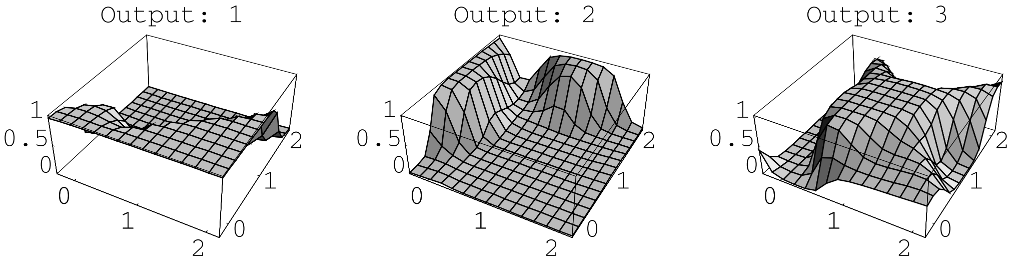
```
In[7]:= NetPlot[fdfrwrd2, x, y, DataFormat → Classifier,
         ContourStyle → {Hue[.4], Hue[.6], Hue[.8]},
         SymbolStyle → {Hue[.4], Hue[.6], Hue[.8]}]
```



The previous plot showed the classification boundary for each class. It is also possible to look at the classification function as a function plot. Since there are three outputs of the network, you obtain three plots. The boundaries indicated in the previous plot are the level curves where the function output equals 0.5 in the function plot shown here.

Look at the function.

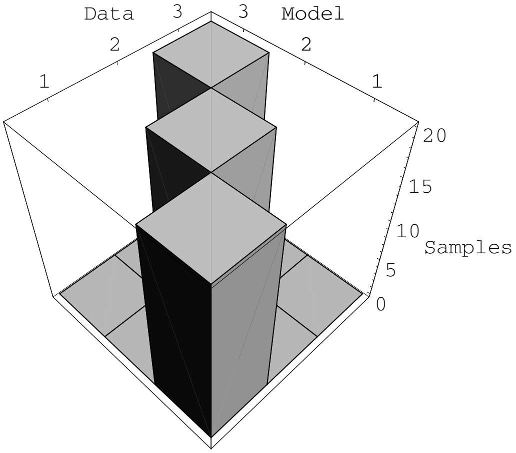
```
In[8]:= NetPlot[fdfrwrd2, x, y, DataFormat → FunctionPlot,
          Ticks → {{0, 1, 2}, {0, 1, 2}, {0, 0.5, 1}}]
```



This option can be used for problems with one or two input signals.

By giving the option `BarChart`, you obtain bar charts showing the classification performance. Correctly classified data is found on the diagonal and the misclassified data corresponds to the off-diagonal bars. Notice that, since the outputs of the FF network take values in the range {0, 1}, you do not obtain precise classifications but, rather, a "degree" of membership. This situation may be corrected by using a `UnitStep` output neuron with the option setting `OutputNonlinearity→UnitStep`. Then the outputs will be either 0 or 1, as desired.

```
In[9]:= NetPlot[fdfrwrd2, x, y, DataFormat → BarChart, OutputNonlinearity → UnitStep]
```



On the $x$ and $y$ axes, you have the class of the samples according to the output data and according to the network classifier. On the $z$ axis is the number of samples. For example, in the bin (2, 3) is the number of data samples from the second class, according to the supplied output data, but classified into the third class by the network. Therefore, the diagonal bins correspond to correctly classified samples, that is, the network assigns these samples to the same class as indicated in the output data.

In contrast to `FunctionPlot` and `Classifier`, the `BarChart` option can be used to visualize the performance of classifiers of any input dimensions.

So far you have evaluated the end result of the training—the derived FF network. It is also possible to display the same plots but as a function of the training iterations. Consider the training record.

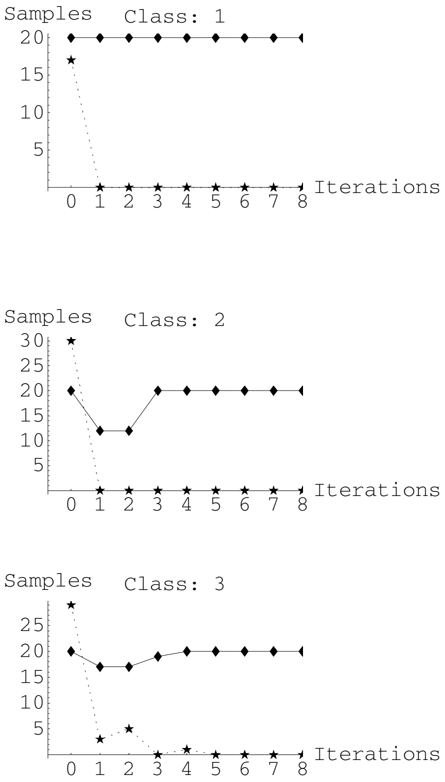
```
In[10]:= fitrecord

Out[10]= NeuralFitRecord[FeedForwardNet,
          ReportFrequency → 1, CriterionValues → {-values-},
          CriterionValidationValues → {-values-}, ParameterRecord → {-parameters-}]
```

The first component is just a copy of the FF network model. The second component contains several information items about the training. Section 7.8, The Training Record, shows you how to extract the information from the training record. Here, you will see how this information can be illustrated in different ways by using `NetPlot` and by choosing a different `DataFormat` option.

Look at the classification performance for each class during training. Correctly classified samples are marked with diamonds and a solid line, incorrectly assigned samples are indicated with stars and a dashed line.

```
In[11]:= NetPlot[fitrecord, x, y,
           DataFormat → ClassPerformance, OutputNonlinearity → UnitStep]
```

```
Correctly/incorrectly classified data
```

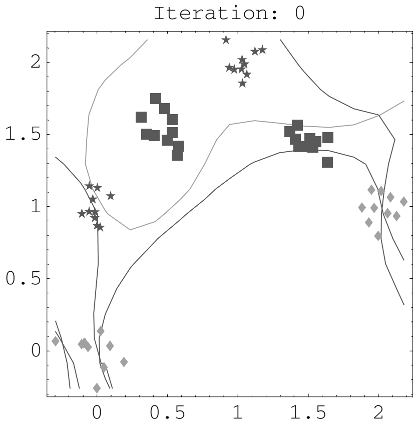


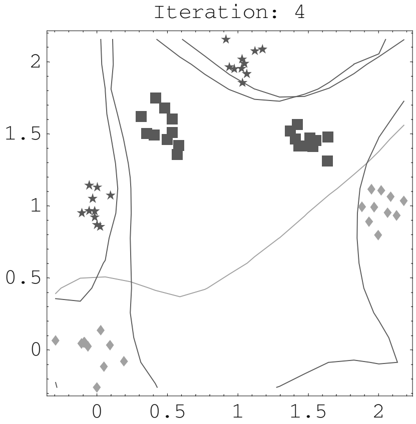
The training progress of the classifier may be viewed as a function of iteration using the option setting `DataFormat→Classifier`. By default, the display shows the evolving boundaries at every (5 × report frequency) iterations, where the report frequency is determined by the option `ReportFrequency` of `NeuralFit`. The display frequency may be changed from 5 to any other positive integer by explicitly setting `Intervals` to a desired value, such as 4 in the present example.

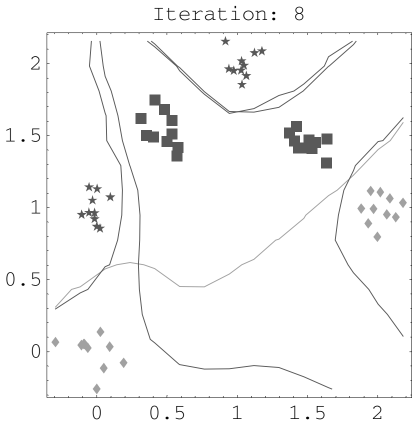
Plot the classifier at every four training iterations.

```
In[12]:= NetPlot[fitrecord, x, y, DataFormat → Classifier,
           Intervals → 4, ContourStyle → {Hue[.4], Hue[.6], Hue[.8]},
           SymbolStyle → {Hue[.4], Hue[.6], Hue[.8]}]
```

```
Classification  Boundaries
```







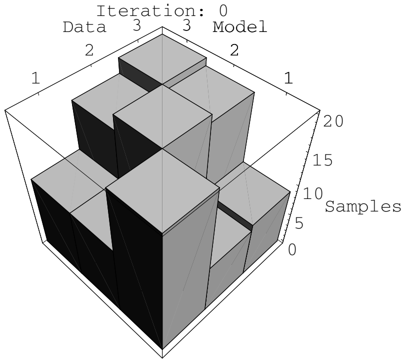
If you prefer, the progress can be animated as described in Section 5.2.1, Function Approximation in One Dimension, instead of being given in a graphics array.

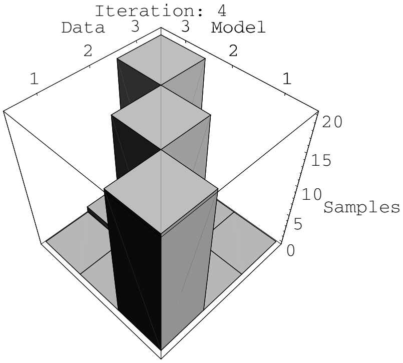
Also the `BarChart` option can be used to evaluate the progress of the training. Changing the nonlinearity at the output from the smooth sigmoid to a discrete step makes the output take only the values 0 and 1.

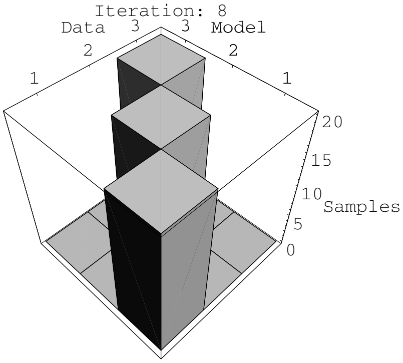
Illustrate the classification result every four iterations of the training.

```
In[13]:= NetPlot[fitrecord, x, y, DataFormat → BarChart,
          OutputNonlinearity → UnitStep, Intervals → 4]
```

```
Classification after
```







As seen in the plots, in the beginning of the training the network model classifies a lot of samples incorrectly. These incorrectly classified samples are illustrated with the non-diagonal bins. As the training proceeds, more of the samples are classified correctly; at the end you can see all of the samples are correctly classified because all samples are at the diagonal.

This result can easily be animated as described in Section 5.2.1, Function Approximation in One Dimension.

## 5.4 Further Reading

FF neural networks are covered in the following textbooks, among others:

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, Macmillan, 1999.

M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, Cambridge, MA, The MIT Press, 1995.

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Redwood City, CA, Addison-Wesley, 1991.

# 6 The Radial Basis Function Network

Compared to the FF network, the RBF network is the next-most-used network model. As the name implies, this network makes use of *radial* functions. Section 2.5.2, Radial Basis Function Networks, gives a tutorial introduction to this network model.

FF and RBF networks can be used for the same types of problems, and the commands and their options are very similar. Therefore, instead of repeating a lot of information, the presentation here will be brief, giving references to the corresponding place in the section on FF networks where possible.

Using the symbolic computing capability of *Mathematica,* the structure of the RBF network may be modified much more naturally than is possible in other neural network software tools. Some of the features are listed in Chapter 5, The Feedforward Neural Network.

## 6.1 RBF Network Functions and Options

This section introduces the commands to initialize, train, and use RBF networks. Since most of these commands are the same as those used for FF models, references will be given to the chapter on FF networks instead of repeating the material here. Examples using the commands can be found in Section 6.2, Examples.

### 6.1.1 InitializeRBFNet

RBF networks are initialized with `InitializeRBFNet`.

| | |
|---|---|
| `InitializeRBFNet[`*x*, *y*, *nb*, *opts*`]` | |
| | initializes an RBF network based on the input data *x* and the output data *y* with the number of neurons given by the integer *nb* |

Initializing an RBF network.

The returned network is an object with head `RBFNet`, following the general format described in Section 3.2.3, Network Format. `FeedForwardNet` and `RBFNet` have one replacement rule more then the other network models. Its name is `NumberOfInputs`, and it indicates how many inputs the network takes.

The number of inputs and outputs in the network are extracted from the number of columns in the input and output data, so only the number of neurons needs to be specified.

InitializeRBFNet takes almost the same options as InitializationFeedforwardNet. However, some of the default values for the options are different, as indicated in the table that follows.

| *option* | *default value* | |
|---|---|---|
| LinearPart | True | indicates whether a linear model should be placed in parallel to the net |
| Neuron | Exp | neuron activation function |
| RandomInitialization | False | indicates whether the parameters should be randomly initialized; the default is to use a smart initialization |
| Regularization | None | indicates regularization in the least-squares fit of the linear parameters |
| FixedParameters | None | indicates whether some parameters should be fixed and, therefore, excluded from the training |
| InitialRange | 1 | indicates the range of the uniform probability function if the parameters are initialized randomly |
| OutputNonlinearity | None | indicates whether the output neuron should be nonlinear; for classification problems OutputNonlinearity→ Sigmoid is recommended |
| Compiled | True | uses the compiled version |

Options of InitializeRBFNet.

The parameters of the network can be initialized in three different ways depending on the option RandomInitialization: False, which is the default; True; and LinearParameters. The default initialization is usually the best. It gives a random distribution of the basis centers over the input data range. This makes sense since only basis functions that cover the data range can be used in the function approximation. Also the widths of the basis functions are scaled using the input data range. The linear parameters are fitted with the least-squares algorithm to the output data. The meanings of the options are the same as for FF networks, and they are further described in Section 5.1.1, InitializeFeedForwardNet. You can also define your own initialization algorithm and insert the parameters in an `RBFNet` object as described in Section 13.1, Change the Parameter Values of an Existing Network.

The options `Regularization` and `FixedParameters` can be set at the initialization of the network or when the network is trained with `NeuralFit`. You can learn how to use these options in Section 7.5, Regularization and Stopped Search, and Section 13.2, Fixed Parameters.

The default value of the option `Neuron` is `Exp`. You then obtain an RBF network with the Gaussian bell function as the basis function, which is the most commonly used choice. Section 13.3, Select Your Own Neuron Function, describes how you can use other basis functions.

### 6.1.2 NeuralFit

The initialized RBF network is trained with `NeuralFit`. This command is also used for FF networks and for dynamic networks. An introduction to using it is given in Section 5.1.2, NeuralFit. Chapter 7, Training Feedforward and Radial Basis Function Networks, describes the command and its options in detail.

A derived RBF network can be applied to new inputs using function evaluation. The result given by the network is its estimate of the output.

| | |
|---|---|
| *net* [ *x* ] | evaluates *net* on the input vector *x* |

Function evaluation of an RBF network.

The input argument *x* can be a vector containing one input sample or a matrix containing one input sample on each row.

The function evaluation has one option.

| *option* | *default value* | |
|---|---|---|
| `Compiled` | `True` | indicates that a compiled version of the evaluation rule should be used |

Option of the network evaluation rule.

### 6.1.3 NetInformation

Information about an RBF network is presented in a string by the function `NetInformation`.

| | |
|---|---|
| `NetInformation[`*rbf*`]` | writes out information about an RBF network |

The `NetInformation` function.

### 6.1.4 NetPlot

The command `NetPlot` can be used to evaluate the obtained RBF network and the training. It is used in the same way as it was used for the FF networks described in Section 5.1.4, NetPlot.

### 6.1.5 LinearizeNet and NeuronDelete

The commands `LinearizeNet` and `NeuronDelete` modify the structure of an existing network.

You can linearize an RBF network at any input signal point using `LinearizeNet`.

| | |
|---|---|
| `LinearizeNet[`*rbf*`, `*x*`]` | linearizes the RBF network at *x* |

Linearize an RBF network.

`LinearizeNet` returns a linear model in the form of an FF network without any hidden neurons as described in Section 2.5.1, Feedforward Neural Networks.

The point of the linearization *x* should be a list of real numbers of length equal to the number of inputs of the neural network.

Sometimes it may be of interest to remove parts of an existing network. `NeuronDelete` can be used to remove outputs, inputs, hidden neurons, or a linear submodel.

You can also remove individual parameters by setting their numerical values to zero and excluding them from the training, as described in Section 13.2, Fixed Parameters.

| | |
|---|---|
| `NeuronDelete[`*net*,*pos*`]` | deletes the neurons indicated with *pos* in an existing network *net* |
| `NeuronDelete[`*net*,*pos*,*x*`]` | deletes the neurons indicated with *pos* in an existing network *net* with the input data supplied and the remaining network parameters adjusted |

Deleting the neurons from an existing network.

The argument *pos* indicates which part of the network should be deleted in the following ways:

{0, 0}: removes the linear submodel

{0, *m*}: removes input *m*

{1, *m*}: removes neuron *m*

{2, *m*}: removes output *m*

The argument *pos* can also be a list where each element follows these rules.

If input data is submitted, then the parameters of the output layer and the linear submodel are adjusted so that the new network approximates the original one as well as possible. The least-squares algorithm is used for this.

There is no adjustment of the parameters if an input or an output is removed.

### 6.1.6 SetNeuralD, NeuralD, and NNModelInfo

The commands `SetNeuralD`, `NeuralD`, and `NNModelInfo` are primarily for internal use in the *Neural Networks* package, but they may be useful if you want to perform more special operations. They are used in the same way for RBF as for FF networks and an explanation can be found in Section 5.1.6, SetNeuralD, NeuralD, and NNModelInfo.

## 6.2 Examples

This section gives some simple function approximation examples with one and two input variables and one and two output variables. The examples are analogous to the examples given on FF networks in Section 5.2, Examples. The reason for this similarity is that RBF networks and FF networks are used for the same type of problems. The best choice between these two alternative models depends on the specific problem. You can seldom tell which one is best without trying them both.

Notice that if you re-evaluate the examples, you will not receive exactly the same results. The reason for this indeterminacy is that the initial network parameters are partly random. See Section 6.1.1, InitializeRBFNet.

### 6.2.1 Function Approximation in One Dimension

The following example illustrates a one-dimensional approximation problem. You can change the example by modifying the function that generates the data and then re-evaluating the entire example. The result depends not only on the data and the number of chosen neuron basis functions, but also on the initialization of the parameters of the RBF network. Since the initialization is partly random, you should expect that the answers will differ each time the network is trained.

Read in the *Neural Networks* package and a standard add-on package.

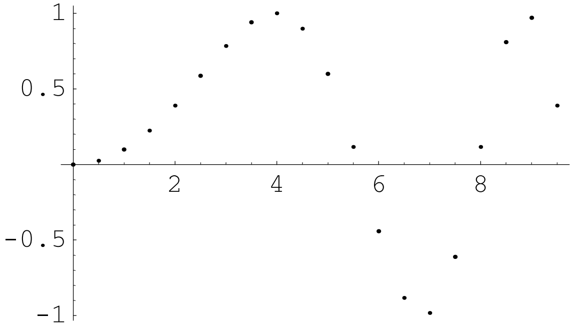
```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << LinearAlgebra`MatrixManipulation`
```

It is always a good idea to look at the data. In one-dimensional problems this is especially easy. Here you can also modify the number of data samples and the function that generates the data.

Generate and look at data.

```
In[3]:= Ndata = 20;
        x = Table[10 N[{i / Ndata}], {i, 0, Ndata - 1}];
        y = Sin[0.1 x^2];
        ListPlot[AppendRows[x, y]]
```



The training data consists of the input data *x* and the output *y*.

Initialize an RBF network with four neurons.

```
In[7]:= rbf = InitializeRBFNet[x, y, 4]

Out[7]= RBFNet[{{w1, λ, w2}, χ}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 36, 27},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

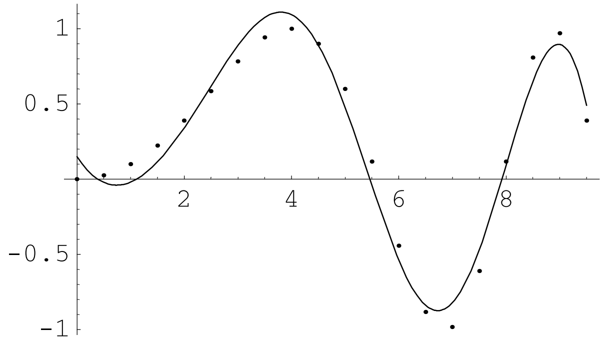
Find some information about the network.

```
In[8]:= NetInformation[rbf]

Out[8]= Radial Basis Function network. Created 2002-4-3 at
          13:36. The network has 1 input and 1 output.  It consists of 4
          basis functions of Exp type. The network has a linear submodel.
```

You can check the performance of the RBF network before the training. Often the initialization gives a fairly good result.

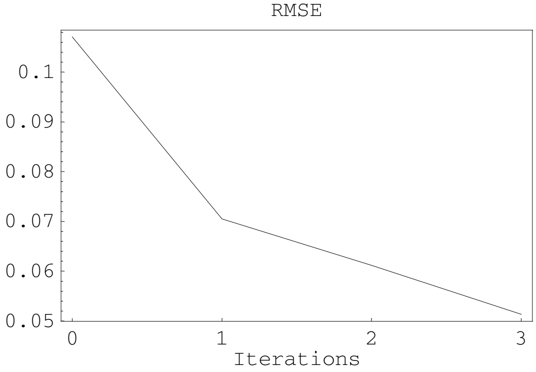
Look at the initialized RBF network on the data domain.

```
In[9]:=  NetPlot[rbf, x, y]
```



Fit the network to the data applying three iterations.

```
In[10]:=  {rbf2, fitrecord} = NeuralFit[rbf, x, y, 3];
```



Often a warning is given at the end of the training that indicates that the training was not complete. This is quite common in connection with neural network models, especially when they involve a large number of parameters. Usually, the best thing to do is to look at the decrease in the performance index to decide whether more training iterations are necessary. In this example where only three iterations were applied, you normally would apply more so that you see that the criterion flattened out.

The trained RBF network can now be used to process new input data. The output is the approximation of the unknown true function at the point corresponding to the input.

Evaluate the RBF model at a new input value.

```
In[11]:= rbf2[{1.5}]

Out[11]= {0.222969}
```

Evaluate the RBF model at several new input values.

```
In[12]:= rbf2[{{1.5}, {0.3}, {2.5}}]

Out[12]= {{0.222969}, {0.0298877}, {0.52558}}
```

By applying the network to a list of symbols you obtain a *Mathematica* expression describing the network.

```
In[13]:= rbf2[{a}]
```

Out[13]=

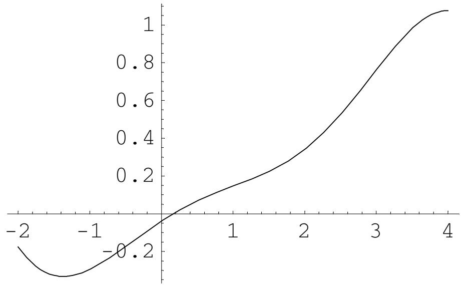
$$\left\{-81.4446 - 14.1937\,a + 940.804\,e^{-0.0194935\,(-7.77038+a)^2} - 755.36\,e^{-0.0242975\,(-7.28812+a)^2} + 9.54363\,e^{-0.146868\,(-4.84531+a)^2} - 1.55335\,e^{-0.0308604\,(-3.54432+a)^2}\right\}$$

You can then go on and manipulate this expression as you would any other *Mathematica* expression.

You can plot the function of the RBF network on any interval of your choice.

Plot the RBF network on the interval {−2, 4}.

```
In[14]:= Plot[rbf2[{x}], {x, -2, 4}]
```

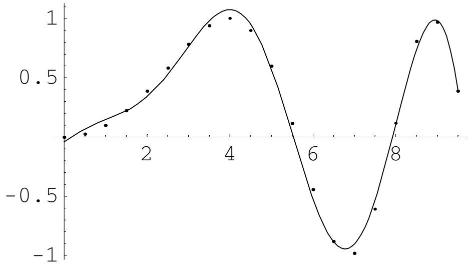


Usually, a function estimate cannot be used outside the domain of the training data. In this case, this means that the RBF network can only be used on the interval {0, 10}. Go back and check that this was the interval of the training data.

You can also use the command NetPlot to plot the function approximation together with the data. The range of the plot is set to the range of the supplied input data.

Plot the estimated function.

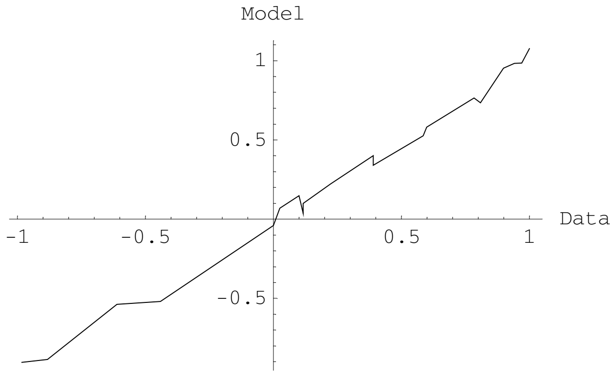
```
In[15]:= NetPlot[rbf2, x, y]
```



NetPlot can be used in many ways to illustrate the quality of the trained neural network. For example, by giving the option DataFormat→NetOutput, you obtain a plot of the model output as a function of the given output. If the network fits the data exactly, then this plot shows a straight line with unit slope through the origin. In real applications you always have noise on your measurement, and you can only expect an approximate straight line if the model is good.

Plot the model output versus the data output.

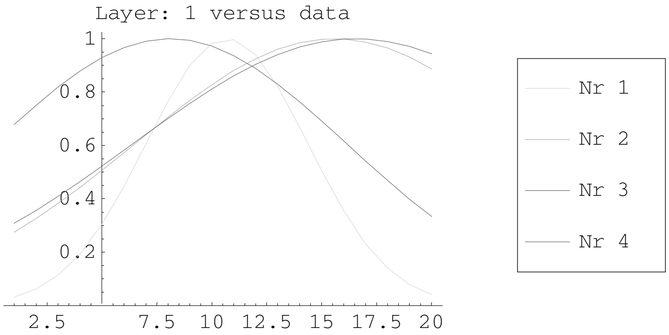
```
In[16]:= NetPlot[rbf2, x, y, DataFormat → NetOutput]
```



It might be interesting to look at the position of the basis functions. If several of them are placed on the same position or close to one another, this indicates that the number of neurons was higher than necessary, or that the initial parameters were unfavorable. In that case you can go back and initialize a new RBF network with fewer neurons. By giving the option DataFormat→HiddenNeurons, you display the output values of the individual neurons versus the data.

Look at the basis functions.

```
In[17]:= NetPlot[rbf2, x, y, DataFormat → HiddenNeurons,
           PlotStyle → {Hue[.2], Hue[.4], Hue[.6], Hue[.8]},
           PlotLegend → Map["Nr " <> ToString[#] &, Range[Length[rbf2[[1, 1, 1, 1]]]]]]
```

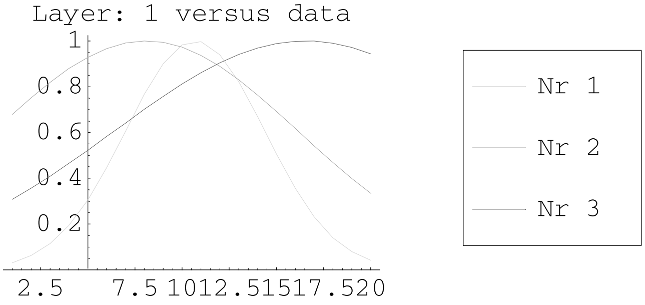


If you see fewer than four basis functions in the plot, then two or more of the basis functions are identical and placed on top of each other. These functions can be identified using the legend. Remember that the result will be slightly different each time you evaluate the example. If some basis functions are identical, then one of them may actually be removed without compromising the approximation quality. Assume that you want to delete the second basis function.

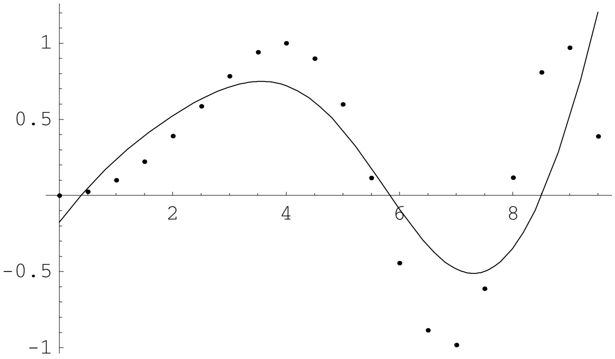
Remove the second neuron, plot the basis function, and plot the approximation of the new network.

```
In[18]:= rbf3 = NeuronDelete[rbf2, {1, 2}, x]

Out[18]= RBFNet[{{w1, λ, w2}, χ}, {AccumulatedIterations → 3,
           CreationDate → {2002, 4, 3, 13, 36, 27}, Neuron → Exp,
           FixedParameters → None, OutputNonlinearity → None, NumberOfInputs → 1}]
```

```
In[19]:= NetPlot[rbf3, x, y, DataFormat → HiddenNeurons,
          PlotStyle → {Hue[.2], Hue[.4], Hue[.6]},
          PlotLegend → Map["Nr " <> ToString[#] &, Range[Length[rbf2[[1, 1, 1, 1]]]]]]
```

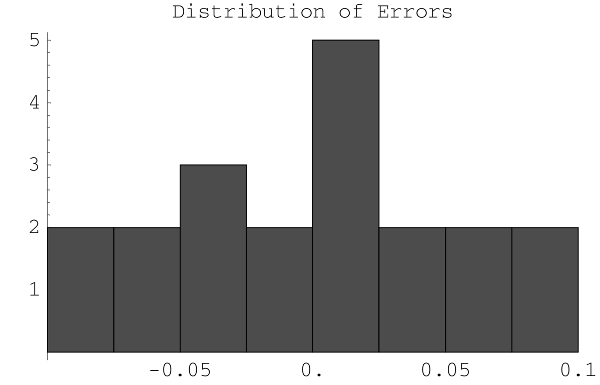


```
In[20]:= NetPlot[rbf3, x, y]
```



By giving the option DataFormat→ErrorDistribution, you obtain a histogram showing the distribution of the error of the fit. This plot may indicate if you have problems with outliers.

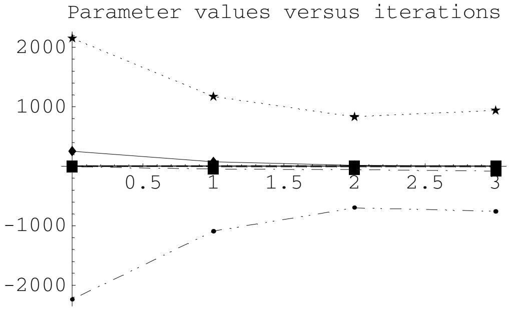
```
In[21]:= NetPlot[rbf2, x, y, DataFormat → ErrorDistribution]
```



The training record can be used to evaluate the training. Section 7.8, The Training Record, shows you how to extract various kinds of information from the training record. You can also use `NetPlot` to plot some information.

Look at how the parameter values change during the training.

```
In[22]:= NetPlot[fitrecord, x, y]
```



Often, depending on the realization, you can see two parameters becoming very large, but with opposite signs. If this happens, then these two parameters are probably in $w_2$ belonging to two identical neuron basis functions. Since the basis functions are nearly identical, the effects of the huge parameters cancel each other.

Check how the parameters are stored in the RBF network.

```
In[23]:= rbf2

Out[23]= RBFNet[{{w1, λ, w2}, χ}, {AccumulatedIterations → 3,
           CreationDate → {2002, 4, 3, 13, 36, 27}, Neuron → Exp,
           FixedParameters → None, OutputNonlinearity → None, NumberOfInputs → 1}]
```

Extract matrix *w2* to check which basis function has large parameter values.

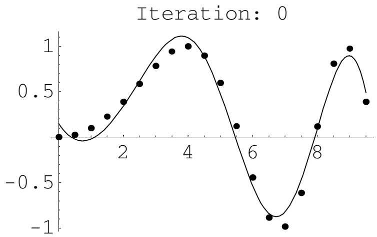
```
In[24]:= MatrixForm[rbf2[[1, 1, 3]]]

  Out[24]//MatrixForm=
```

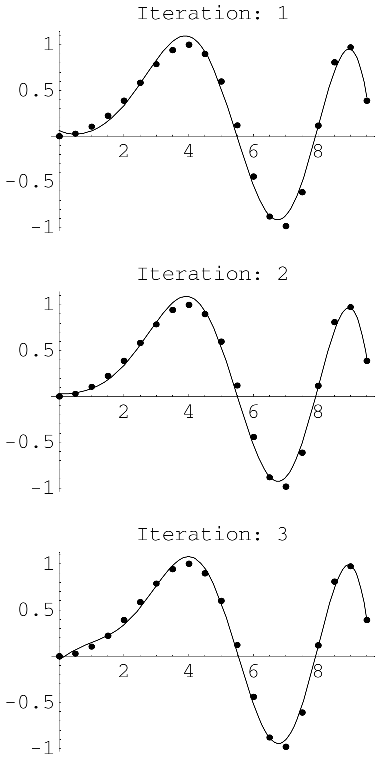
$$\begin{pmatrix} 9.54363 \\ -755.36 \\ -1.55335 \\ 940.804 \\ -81.4446 \end{pmatrix}$$

By choosing `DataFormat→FunctionPlot`, you can see how the approximation improves during the training.

Look at the function approximation after every training iteration.

```
In[25]:= NetPlot[fitrecord, x, y, DataFormat → FunctionPlot,
           PlotStyle → PointSize[0.02], Intervals → 1]

           Function estimate after
```





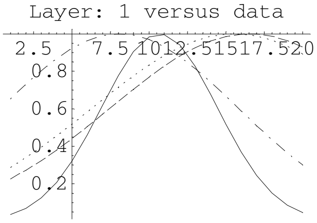
If you prefer, the progress can be animated as described in Section 5.2.1, Function Approximation in One Dimension, instead of being given in a graphics array.

You can also look at how the basis functions move during the training by giving the option `DataFormat→ HiddenNeurons`.

Look at the basis functions during training.

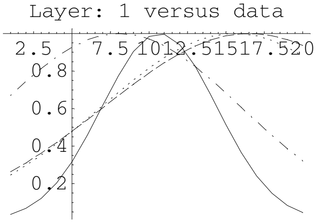
```
In[26]:= NetPlot[fitrecord, x, y, DataFormat → HiddenNeurons, Intervals → 2]
```

```
Values of hidden neurons after

     Iteration: 1
```



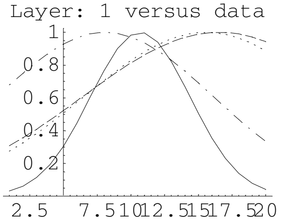
Layer: 1 versus data

```
     Iteration: 3
```



Layer: 1 versus data

```
     Iteration: 4
```



Layer: 1 versus data

From the series of neuron plots, you may see if two basis functions become equal.

### 6.2.2 Function Approximation from One to Two Dimensions

You can approximate functions with several outputs in the same way as with one output. Thus, you can proceed as in the previous example with the only difference being that the output data *y* should contain one output in each column. The basic approach is illustrated in Section 5.2.2, Function Approximation from One to Two Dimensions. To obtain an RBF example, you only have to change the initialization of the network to an RBF network and to re-evaluate the whole example.

### 6.2.3 Function Approximation in Two Dimensions

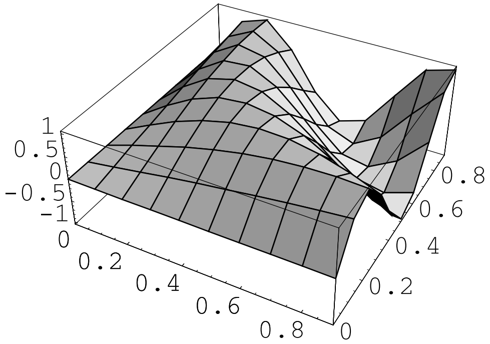
It is easy to approximate functions with several inputs in the same way as with single input functions. A simple example with two inputs is shown here.

Read in the *Neural Networks* package.

```
In[1]:=  << NeuralNetworks`
```

Generate data and look at the function.

```
In[2]:=  Ndata = 10;
         x = Table[N[{i / Ndata, j / Ndata}], {i, 0, Ndata - 1}, {j, 0, Ndata - 1}];
         y = Map[Sin[10. #[[1]] #[[2]]] &, x, {2}];
         ListPlot3D[y, MeshRange → {{0, 0.9}, {0, 0.9}}];
         x = Flatten[x, 1];
         y = Transpose[{Flatten[y]}];
```



You can modify the example by changing the function generating the data and by changing the number of neuron basis functions in the following initialization. Notice that you will obtain slightly different results if you repeat the example without any changes at all. This is due to the randomness in the RBF initialization, described in the algorithm for RBF network initialization.

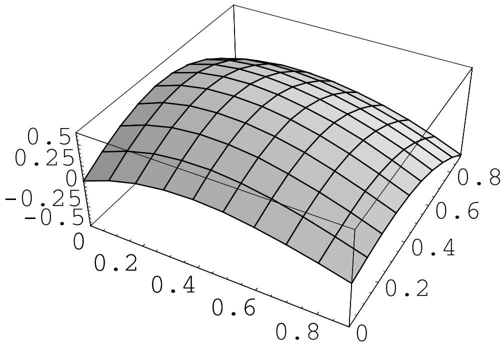
Initialize an RBF network with two neurons.

```
In[8]:=  rbf = InitializeRBFNet[x, y, 2]

Out[8]=  RBFNet[{{w1, λ, w2}, χ}, {Neuron → Exp, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 38, 31},
           OutputNonlinearity → None, NumberOfInputs → 2}]
```

You can apply the initialized network to the input data and plot the network output. Compare the result with the true function in the previous plot.

Look at the initialized RBF network.

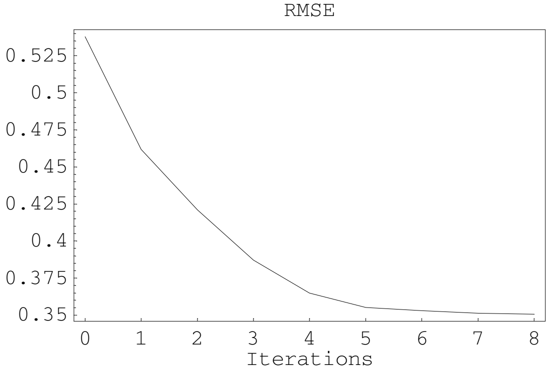
```
In[9]:=  y2 = rbf[x];
         ListPlot3D[Partition[Flatten[y2], Ndata], MeshRange → {{0, 0.9}, {0, 0.9}}]
```



So far the network has only been initialized. Now it is time to train it.

Fit the RBF network to the data.

```
In[11]:= {rbf2, fitrecord} = NeuralFit[rbf, x, y, 8];
```

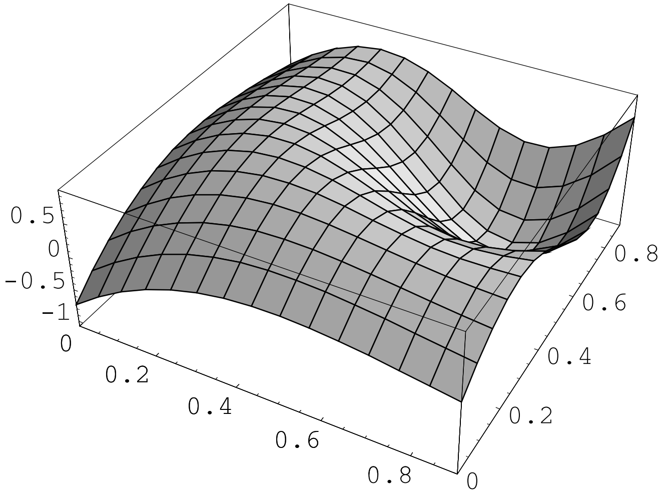


Normally, there are several local minima; therefore, if you repeat the initialization and the training, you can expect to obtain different results.

You can use `NetPlot` to illustrate the trained RBF network over the data range.

Look at the result with the fitted RBF network.

```
In[12]:= NetPlot[rbf2, x, y, DataFormat → FunctionPlot]
```

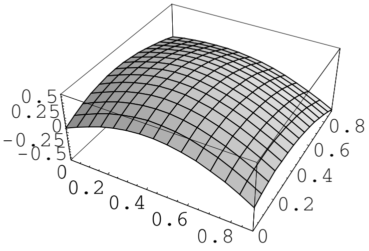


This plot shows the function approximation after the eight iterations of training. `NetPlot` may also be applied to the training record to obtain a graphics array of function estimates at specified iteration intervals, as shown here.

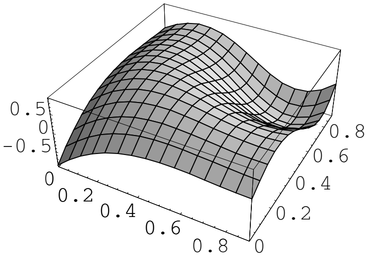
```
In[13]:= NetPlot[fitrecord, x, y, DataFormat → FunctionPlot, Intervals → 3]
```

```
Function estimate after
```

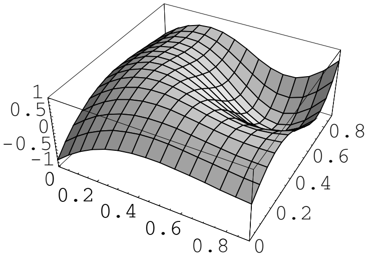
Iteration: 0

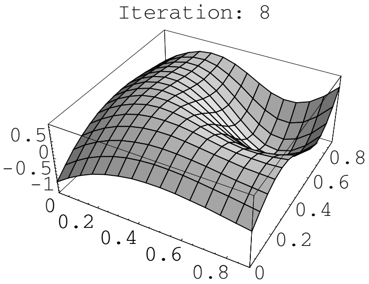


Iteration: 3



Iteration: 6

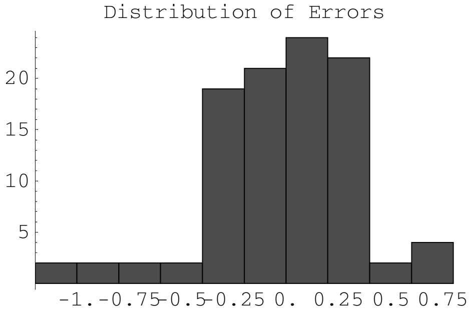




If you prefer, the progress can be animated as described in Section 5.2.1, Function Approximation in One Dimension, instead of being given in a graphics array.

With the option `DataFormat→ErrorDistribution`, you can obtain a histogram showing the distribution of the approximation errors.

```
In[14]:= NetPlot[rbf2, x, y, DataFormat → ErrorDistribution]
```



## 6.3 Classification with RBF Networks

If you have not done so already, load the *Neural Networks* package.

Load the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

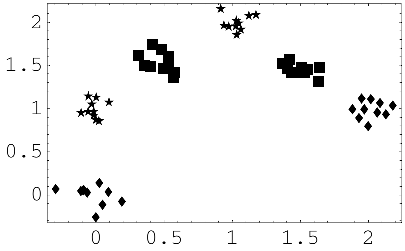
Load data consisting of three classes divided into two clusters each. The data is represented by inputs $x$ and their individual classifications by output $y$. The data format is described in Section 3.2, Package Conventions.

Load the data vectors and output indicating class.

```
In[2]:= << vqthreeclasses.dat;
```

Look at the data.

```
In[3]:= NetClassificationPlot[x, y]
```



In classification problems it is important to have a nonlinearity at the output of the RBF network model. The purpose of the nonlinearity is to ensure that the output value stays within the range indicating the different classes. This is done by using the option `OutputNonlinearity`. Its default is `None`. Set it to `Sigmoid` so that its saturating values are 0 and 1, exactly as the output data indicating the classes.

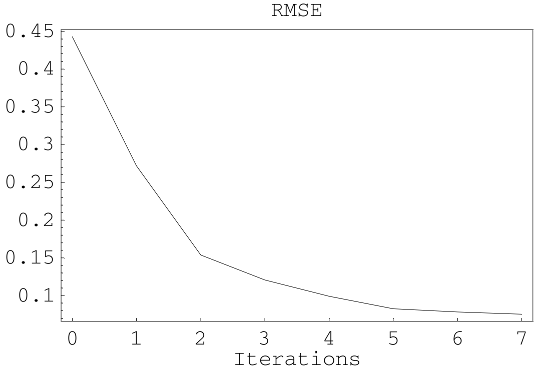
Initialize an RBF network with eight basis functions.

```
In[4]:= rbf = InitializeRBFNet[x, y, 8, OutputNonlinearity → Sigmoid]

Out[4]= RBFNet[{{w1, λ, w2}, χ}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 39, 18},
          OutputNonlinearity → Sigmoid, NumberOfInputs → 2}]
```

The initialized network can now be trained by using `NeuralFit`.

Train the initialized RBF network.

```
In[5]:= {rbf2, fitrecord} = NeuralFit[rbf, x, y, 7];
```



As usual, the reduction in RMSE over the seven iterations is displayed in a plot. Often, you will also get a warning from the program that the minimum was not reached. It is quite normal not to reach the minimum in neural network training. Neural network models often contain so many parameters that it is extremely difficult to determine them precisely. Instead you should inspect the RMSE plot to determine whether more training is needed to converge to a reliable minimum. If more training is needed, you can use `NeuralFit` to continue the training from where you left off. Since the result also depends on the randomness of the initialization, it might be worthwhile to repeat the training with different initial models.

Obtain some information about the RBF network model.

```
In[6]:= NetInformation[rbf2]

Out[6]= Radial Basis Function network. Created 2002-4-3 at
          13:39. The network has 2 inputs and 3 outputs.  It consists
          of 8 basis functions of Exp type. The network has a linear
          submodel. There is a nonlinearity at the output of type Sigmoid.
```

The trained RBF network classifier may now be used to classify new input vectors.

Classify two data vectors.

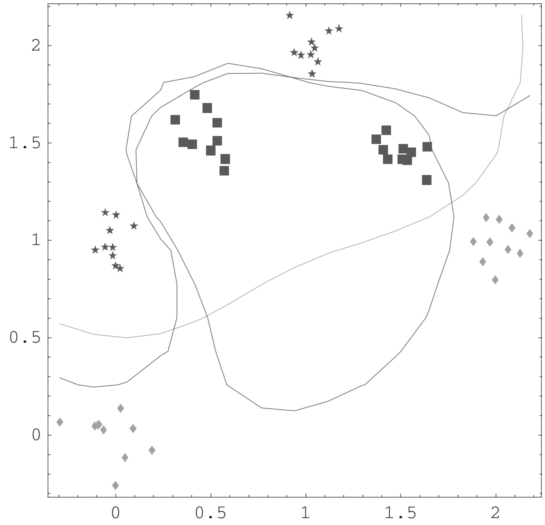
```
In[7]:= rbf2[{{0, 0.5}, {2, 1}}]

Out[7]= {{0.503741, 0.988144, 0.000395902}, {0.91636, 0.0118067, 0.083243}}
```

The data vectors are classified to the class with the largest output value. If several outputs give large values, or if none of them do, then the classifier is highly unreliable for this data.

The result can be illustrated in several ways using `NetPlot`.

Plot the classification borders together with the data.

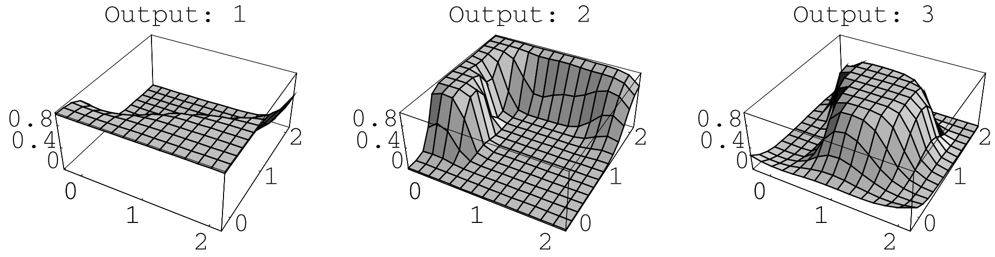
```
In[8]:= NetPlot[rbf2, x, y, DataFormat → Classifier,
         ContourStyle → {Hue[.4], Hue[.6], Hue[.8]},
         SymbolStyle → {Hue[.4], Hue[.6], Hue[.8]}]
```



This option can, of course, only be used in two-dimensional classification problems.

Look at the functions.

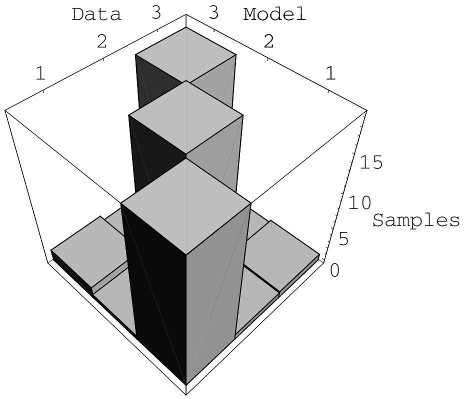
```
In[9]:=  NetPlot[rbf2, x, y, DataFormat → FunctionPlot,
           Ticks → {{0, 1, 2}, {0, 1, 2}, {0, 0.4, 0.8}}]
```



This option can be used for problems with one or two input signals. The result is given as a graphics array. The classification boundaries are defined where the functions take the value 0.5.

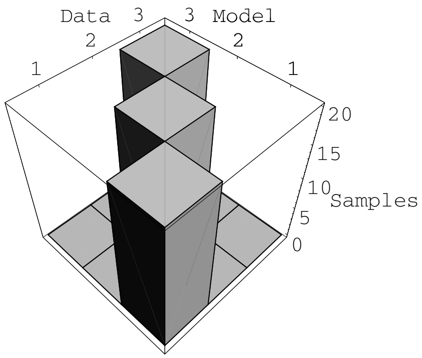
By giving the option `BarChart`, you can obtain bar charts showing the classification performance. Correctly classified data is found on the diagonal, and the misclassified data corresponds to the off-diagonal bars. Notice that a data vector may be assigned to several classes or to no class at all.

```
In[10]:=  NetPlot[rbf2, x, y, DataFormat → BarChart]
```



The RBF classifier with a sigmoid on the output gives outputs in the open interval {0, 1}. This can be interpreted as the “degree” of membership or probability of membership. However, it is often interesting to have a discrete answer of 0 or 1. This can be obtained in `NetPlot`, by specifying the option `OutputNonlinear` ity→UnitStep, which will replace the output sigmoid with a discrete unit step. Compare the resulting plot of the classifier function with the preceding one.

```
In[11]:= NetPlot[rbf2, x, y, DataFormat → BarChart, OutputNonlinearity → UnitStep]
```



In contrast to FunctionPlot and Classifier, the BarChart option can be used for classifiers of any dimensions.

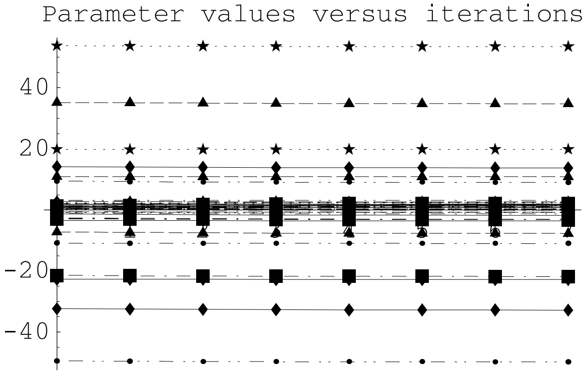
So far you have evaluated the end result of the training: the obtained RBF network. Now consider the training record.

```
In[12]:= fitrecord

Out[12]= NeuralFitRecord[RBFNet, ReportFrequency → 1, CriterionValues → {-values-},
          CriterionValidationValues → {-values-}, ParameterRecord → {-parameters-}]
```

The first component is just a copy of the RBF network. The second component contains several items with information about the training. Section 7.8, The Training Record, shows you how to extract the information from the training record. Here you will see how this information can be illustrated in different ways using NetPlot and depending on which DataFormat option is chosen.

Look at how the parameters change during the training.

```
In[13]:= NetPlot[fitrecord, x, y, DataFormat → ParameterValues]
```

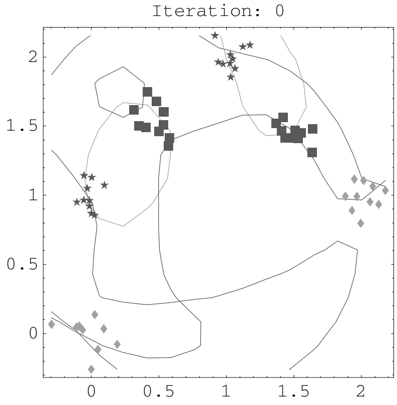


The evolution of the classifier as a function of training iterations can be displayed using the option `Classifier`. As before, you can display snapshots at prescribed iteration intervals using the option `Intervals`.

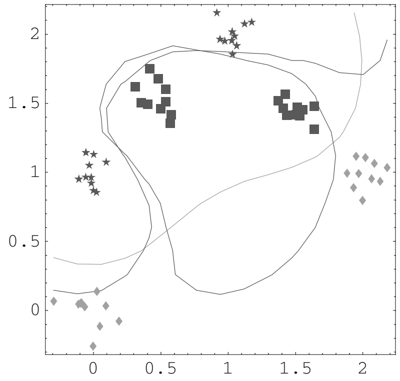
Plot a classifier with a frequency three times the value of `ReportFrequency`.

```
In[14]:= NetPlot[fitrecord, x, y, DataFormat → Classifier,
          Intervals → 3, ContourStyle → {Hue[.4], Hue[.6], Hue[.8]},
          SymbolStyle → {Hue[.4], Hue[.6], Hue[.8]}]
```

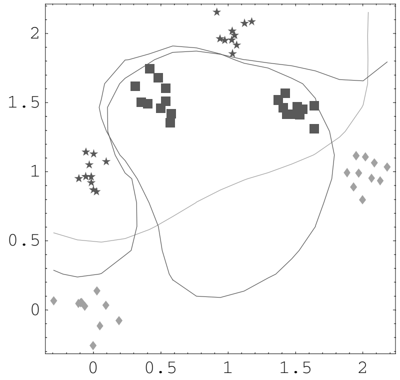
```
Classification  Boundaries
```



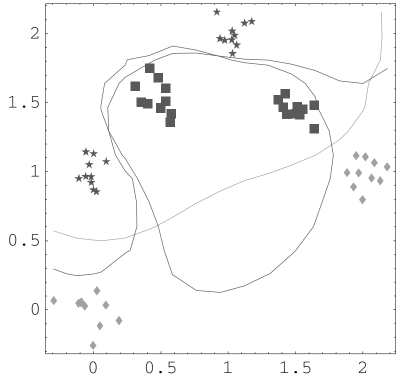
Iteration: 3



Iteration: 6



Iteration: 7



## 6.4 Further Reading

RBF networks are covered in most textbooks on neural networks. Some examples are as follows:

M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, Cambridge, MA, The MIT Press, 1995.

S. Haykin, *Neural Networks: A Comprehensive Foundation,* 2nd ed., New York, Macmillan, 1999.

# 7 Training Feedforward and Radial Basis Function Networks

This section describes the training algorithms available for FF networks and RBF networks using the command `NeuralFit`. This command is also used for dynamic networks. First, a detailed description of the command is given in Section 7.1, NeuralFit, followed by Section 7.2, Examples of Different Training Algorithms. In Section 7.3, Train with FindMinimum, `NeuralFit` is used to call the built-in command `FindMinimum`. In Section 7.4, Troubleshooting, some possible remedies to frequent problems with the training are presented. Sections 7.5 to 7.8 contain examples on how the options of `NeuralFit` can be used to change the basic minimization algorithm. In Section 7.9, Writing Your Own Training Algorithms, the commands `SetNeuralD` and `NeuralD` are described; they may be useful if you want to develop your own training algorithm for FF and RBF networks.

A short tutorial on the training (or, equivalently, minimization) can be found in Section 2.5.3, Training Feedforward and Radial Basis Function Networks. For a more thorough background on minimization, you can consult the references at the end of the chapter.

## 7.1 NeuralFit

`NeuralFit` is used to train FF and RBF networks. Prior to the training you need to initialize the network, as described in Section 5.1.1, InitializeFeedForwardNet, and Section 6.1.1, InitializeRBFNet. In the following, *net* indicates either of the two possible network types.

Indirectly `NeuralFit` is also used to train dynamic networks, since `NeuralARXFit` and `NeuralARFit` actually call `NeuralFit`. Therefore, the description given here also applies to these two commands.

To train the network you need a set of training data $\{x_i, y_i\}_{i=1}^{N}$ containing $N$ input-output pairs.

| | |
|---|---|
| NeuralFit[*net*, *x*, *y*] | trains the model *net* using input data *x* and output data *y* for a default number of training iterations (30) |
| NeuralFit[*net*, *x*, *y*, *xv*, *yv*] | |
| | trains the model *net* using input data *x* and output data *y* for a default number of training iterations with the validation data *xv*, *yv* submitted |

| | |
|---|---|
| `NeuralFit[`*net*, *x*, *y*, *iterations*`]` | trains the model *net* using input data *x* and output data *y* for a specified number of iterations |
| `NeuralFit[`*net*, *x*, *y*, *xv*, *yv*, *iterations*`]` | trains the model *net* using training data *x* and *y*, with submitted validation data *xv* and *yv*, for a specified number of iterations |

Training an FF, RBF, or dynamic network.

`NeuralFit` returns a list of two variables. The first one is the trained *net* and the second is an object with head `NeuralFitRecord` containing information about the training.

An existing network can be submitted for more training by setting *net* equal to the network or its training record. The advantage of submitting the training record is that the information about the earlier training is combined with the additional training.

During the training, intermediate results are displayed in an automatically created notebook. After each training iteration, the following information is displayed:

- Training iteration number
- The value of the RMSE
- If validation data is submitted in the call, then also displayed is the RMSE value computed on this second data set
- The step-size control parameter of the minimization algorithm ($\lambda$ or $\mu$), which is described in Section 2.5.3, Training Feedforward and Radial Basis Function Networks

At the end of the training, the RMSE decrease is displayed in a plot as a function of iteration number.

Using the options of `NeuralFit`, as described in Section 7.7, Options Controlling Training Results Presentation, you can change the way the training results are presented.

At the end of the training process, you may receive different warning messages. Often, however, the RMSE curve flattens out although no exact minimum is reached. Instead, by looking at the RMSE plot, you can usually tell if more training iterations are necessary. If the RMSE curve has not flattened out toward the end of the training, then you should consider continuing the training. This can be done by submitting the trained network, or its training record, a second time to `NeuralFit` so that you do not have to restart the training from the beginning.

If you do not want the warnings, you can switch them off using the command `Off`.

All training algorithms may have problems with local minima. By repeating the training with different initializations of the network, you decrease the risk of being caught in a minimum giving a bad performing network model.

`NeuralFit` takes the following options.

| *option* | *default values* | |
|---|---|---|
| `Regularization` | `None` | quadratic regularization of the criterion of fit |
| `Method` | `Automatic` | training algorithm |
| `Momentum` | `0` | momentum in backpropagation training |
| `StepLength` | `Automatic` | step size in steepest descent<br>and backpropagation training |
| `FixedParameters` | `None` | parameters excluded from training |
| `Separable` | `Automatic` | use the separable algorithm |
| `PrecisionGoal` | `6` | number of digits in the stop criterion |
| `CriterionPlot` | `Automatic` | criterion plot given at end of training |
| `CriterionLog` | `True` | intermediate results are logged during training |
| `CriterionLogExtN` | `True` | training progress is given in an external notebook |
| `ReportFrequency` | `1` | period of training log and training report |
| `ToFindMinimum` | `{}` | list of options to be used if `Method→FindMinimum` |
| `Compiled` | `True` | use compiled version |
| `MinIterations` | `3` | minimum number of training iterations |
| `MoreTrainingPrompt` | `False` | prompts for more training iterations if set to `True` |

Options of `NeuralFit`.

The options `CriterionPlot`, `CriterionLog`, `CriterionLogExtN`, `ReportFrequency`, and `MoreTrainingPrompt` are common with the other training commands in the *Neural Networks* package, and they are described in Section 7.7, Options Controlling Training Results Presentation. The rest of the options are explained and illustrated with examples in the section that follows.

## 7.2 Examples of Different Training Algorithms

This section includes a small example illustrating the different training algorithms used by `NeuralFit`. If you want examples of different training algorithms of more realistic sizes, see the ones in Chapter 8, Dynamic Neural Networks, or Chapter 12, Application Examples, and change the option `Method` in the calls to `NeuralFit`.

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

Consider the following small example where the network only has two parameters. This makes it possible to illustrate the RMSE being minimized as a surface. To do this, you need the following package.

Read in a standard package for graphics.

```
In[2]:= << Graphics`Graphics3D`
```

The "true" function is chosen to be an FF network with one input and one output, no hidden layer and with a sigmoidal nonlinearity at the output. The true parameter values are 2 and −1.

Initialize a network of correct size and insert the true parameter values.

```
In[3]:= fdfrwrd = InitializeFeedForwardNet[{{1}}, {{1}},
           {}, RandomInitialization → True, OutputNonlinearity → Sigmoid];
        fdfrwrd[[1]] = {{{{2.}, {-1.}}}};
```

Generate data with the true function.

```
In[5]:= Ndata = 50;
        x = Table[{N[i]}, {i, 0, 5, 10 / (Ndata - 1)}];
        y = fdfrwrd[x];
```

A two-parameter function is defined to carry out the RMSE computation. Note that this function makes use of the generated data $\{x, y\}$ and is needed to generate the plots.

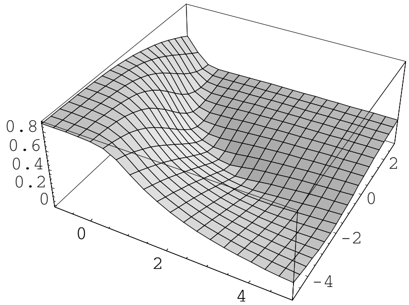
Define the criterion function.

```
In[8]:= criterion[a_, b_] := (fdfrwrd[[1]] = {{{{a}, {b}}}};
           Sqrt[(Transpose[#].#) &[y - fdfrwrd[x]] / Length[x]])[[1, 1]]
```

The criterion function can be plotted in a neighborhood of the minimum (2, −1) using `Plot3D`.

Look at the criterion function.

```
In[9]:= surf = Plot3D[criterion[a, b], {a, -1, 5}, {b, -5, 3}, PlotPoints → 20]
```

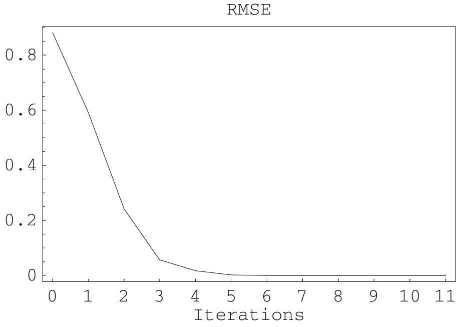


Now it is time to test the different training methods. The initial parameters are chosen to be (−0.5, −5). You can repeat the example with different initializations.

### Levenberg-Marquardt

Initialize the network and train with the Levenberg-Marquardt method.

```
In[10]:= fdfrwrd2 = fdfrwrd;
         fdfrwrd2[[1]] = {{{{-0.5}, {-5}}}};
         {fdfrwrd3, fitrecord} = NeuralFit[fdfrwrd2, x, y];
```



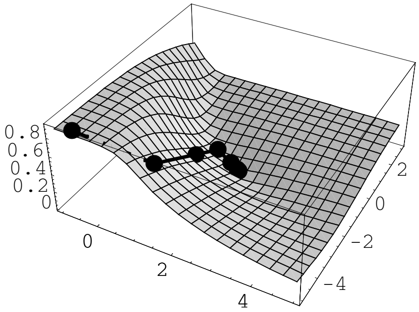
The parameter record and the criterion log are included as rules in the training record, constituting the second output argument of `NeuralFit`. This information may be inserted into a list of three-element sublists containing the two parameter values and the corresponding RMSE value for each iteration of the training process. Viewing this list as three-dimensional {$x$, $y$, $z$} points, it can be used to illustrate the RMSE surface as a function of parameters using `Plot3D`.

Form a list of the trajectory in the parameter space.

```
In[13]:= trajectory =
           Transpose[Append[Transpose[Map[Flatten, (ParameterRecord /. fitrecord[[2]])]],
             (CriterionValues /. fitrecord[[2]]) + 0.05]];
```

Form plots of the trajectory and show it together with the criterion surface.

```
In[14]:= trajectoryplot = ScatterPlot3D[trajectory,
            PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
         trajectoryplot2 = ScatterPlot3D[trajectory, PlotJoined → True,
            PlotStyle → Thickness[0.01], DisplayFunction → Identity];
         Show[surf, trajectoryplot, trajectoryplot2]
```



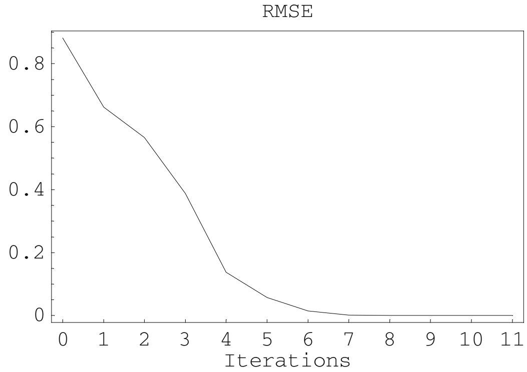
The {$x$, $y$, $z$} iterates of the training process are marked with dots that are connected with straight lines to show the trajectory. The training has converged after about five iterations.

### Gauss-Newton Algorithm

The training of the initial neural network is now repeated with the `GaussNewton` algorithm.

Train the same neural network with the Gauss-Newton algorithm.

```
In[17]:= {fdfrwrd3, fitrecord} = NeuralFit[fdfrwrd2, x, y, Method → GaussNewton];
```

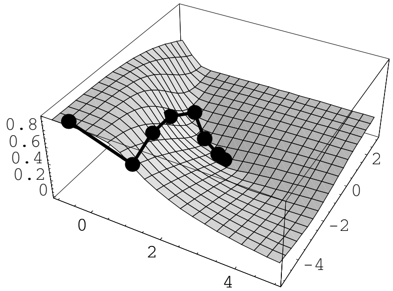


Form a list of the trajectory in the parameter space.

```
In[18]:= trajectory =
           Transpose[Append[Transpose[Map[Flatten, (ParameterRecord /. fitrecord[[2]])]],
             (CriterionValues /. fitrecord[[2]]) + 0.05]];
```

Form plots of the trajectory and show it together with the criterion surface.

```
In[19]:= trajectoryplot = ScatterPlot3D[trajectory,
            PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
         trajectoryplot2 = ScatterPlot3D[trajectory, PlotJoined → True,
            PlotStyle → Thickness[0.01], DisplayFunction → Identity];
         Show[surf, trajectoryplot, trajectoryplot2]
```

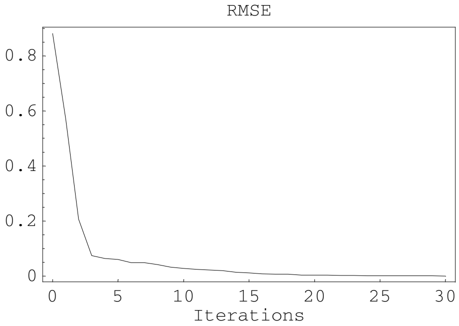


The Gauss-Newton algorithm converges in seven iterations.

### Steepest Descent Method

Train the same neural network with `SteepestDescent`.

```
In[22]:=  {fdfrwrd3, fitrecord} = NeuralFit[fdfrwrd2, x, y, Method → SteepestDescent];
```



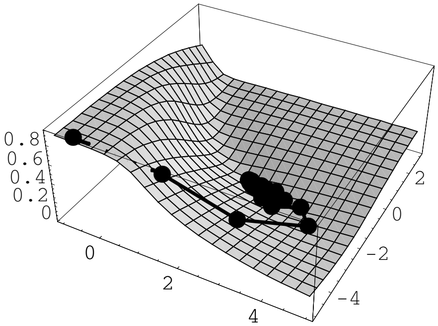
The training did not converge within the 30 iterations. This is not necessarily a problem, since the parameter values may still be close enough to the minimum.

Form a list of the trajectory in the parameter space.

```
In[23]:=  trajectory =
            Transpose[Append[Transpose[Map[Flatten, (ParameterRecord /. fitrecord[[2]])]],
              (CriterionValues /. fitrecord[[2]]) + 0.05]];
```

Form plots of the trajectory and show it together with the criterion surface.

```
In[24]:= trajectoryplot = ScatterPlot3D[trajectory,
            PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
         trajectoryplot2 = ScatterPlot3D[trajectory, PlotJoined → True,
            PlotStyle → Thickness[0.01], DisplayFunction → Identity];
         Show[surf, trajectoryplot, trajectoryplot2]
```



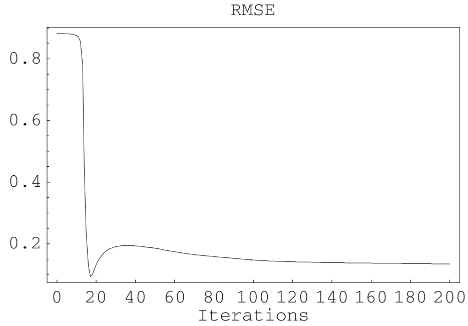
Toward the end of the training the convergence is particularly slow. There, the steepest descent method exhibits much slower convergence than either the Levenberg-Marquardt or Gauss-Newton methods.

### Backpropagation Algorithm

When you use the backpropagation algorithm, you have to choose the step size and the momentum. It may not be an easy matter to choose judicious values for these parameters, something that is not an issue when using the other methods since they automatically tune the step size. You can repeat the example with different values of these parameters to see their influence.

Train the same neural network with backpropagation.

```
In[27]:= {fdfrwrd3, fitrecord} = NeuralFit[fdfrwrd2, x, y,
            200, Method → BackPropagation, StepLength → 0.1, Momentum → 0.9];
```

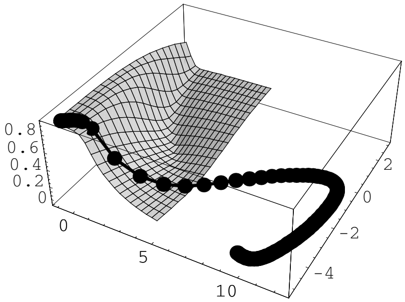


Form a list of the trajectory in the parameter space.

```
In[28]:= trajectory =
           Transpose[Append[Transpose[Map[Flatten, (ParameterRecord /. fitrecord[[2]])]],
             (CriterionValues /. fitrecord[[2]]) + 0.05]];
```

Form plots of the trajectory and show it together with the criterion surface.

```
In[29]:= trajectoryplot = ScatterPlot3D[trajectory,
            PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
         trajectoryplot2 = ScatterPlot3D[trajectory, PlotJoined → True,
            PlotStyle → Thickness[0.01], DisplayFunction → Identity];
         Show[surf, trajectoryplot, trajectoryplot2]
```



Due to the momentum term used in the training, the parameter estimate goes up on the slope adjacent to the initial parameter values. You can repeat the training with different values of the `StepLength` and `Momentum` options to see how they influence the minimization.

## 7.3 Train with FindMinimum

If you prefer, you can use the built-in *Mathematica* command `FindMinimum` to train FF, RBF, and dynamic networks. This is done by giving the option `Method→FindMinimum` to `NeuralFit`. The other choices for `Method` call algorithms especially written for neural network minimization and thus happen to be superior to `FindMinimum` in most neural network problems.

You can submit any `FindMinimum` options by putting them in a list and using the option `ToFindMinimum`.

See the documentation on `FindMinimum` for further details.

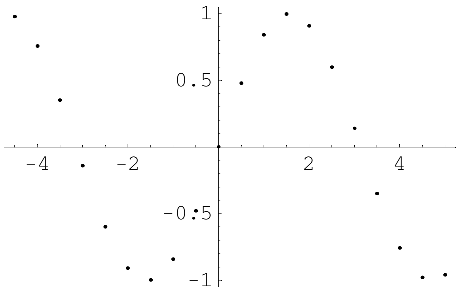
Consider the following small example.

Read in the *Neural Networks* package and a standard add-on package.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

Generate data and look at the function.

```
In[3]:= Ndata = 20;
        x = Table[5. - 10 N[{i / Ndata}], {i, 0, Ndata - 1}];
        y = Sin[x];
        ListPlot[AppendRows[x, y]]
```



Initialize an RBF network randomly.

```
In[7]:= rbf = InitializeRBFNet[x, y, 4, RandomInitialization → True]

Out[7]= RBFNet[{{w1, λ, w2}, χ}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 42, 32},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

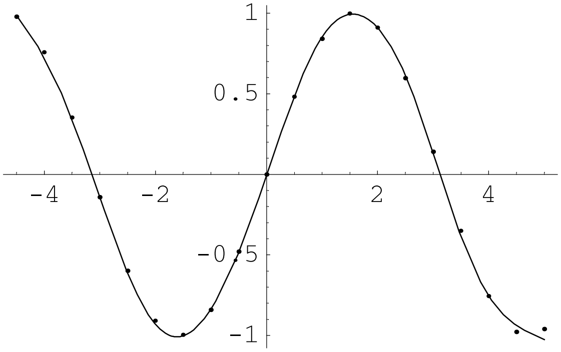
Train with `FindMinimum` and specify that the Levenberg-Marquardt algorithm of `FindMinimum` should be used.

```
In[8]:= {rbf2, fitrecord} = NeuralFit[rbf, x, y, 50,
           Method → FindMinimum, ToFindMinimum → {Method → LevenbergMarquardt}];

        FindMinimum::fmlim : The minimum could not be bracketed in 50 iterations.
```

A main disadvantage with `FindMinimum` is that it is hard to say whether or not the training was in fact successful. You have no intermediate results during training and no criterion plot given at the end of the training. You almost always get the warning at the end of the training that the minimum was not reached. However, the trained network might be a good description of the data anyway. You can visually inspect the model with a plot.

Plot the approximation obtained with the network.

```
In[9]:= NetPlot[rbf2, x, y]
```



You can repeat the example changing the number of iterations and the algorithm used in `FindMinimum`.

## 7.4 Troubleshooting

Sometimes, due to numerical problems, the training stops before reaching the minimum and without completing the iterations specified. Listed here are some measures you can take to possibly circumvent this situation.

- Re-initialize the network model and repeat the training.
- Try a different training algorithm by changing the option `Method`. (See Section 7.1, NeuralFit.)
- Exclude (or include if it is already excluded) a linear part in the network by setting the option `LinearPart→False`. (See Section 5.1.1, InitializeFeedForwardNet, or Section 6.1.1, InitializeRBFNet.)
- Decrease/increase the number of neurons in the network model.
- Check that the data is reasonably scaled so that unnecessary numeric problems are avoided.

## 7.5 Regularization and Stopped Search

A central issue in choosing a neural network model for a given problem is selecting the level of its structural complexity that best suits the data that it must accommodate. If the model contains too many parameters, it will approximate not only the data but also the noise in the data. Then the model is *overfitting* the data. The misfit induced by noise is called the *variance* contribution to the model misfit, which increases with the number of parameters of the model. On the other hand, a model that contains too few parameters will not be flexible enough to approximate important features in the data. This gives a *bias* contribution to the misfit due to lack of flexibility. Since the flexibility increases with the number of parameters, the bias error decreases when the model size increases. Deciding on the correct amount of flexibility in a neural network model is therefore a tradeoff between these two sources of the misfit. This is called the *bias-variance tradeoff*.

Overfitting may be avoided by restricting the flexibility of the neural model in some way. For neural networks, flexibility is specified by the number of hidden neurons.

The *Neural Networks* package offers three ways to handle the bias-variance tradeoff and all three rely on the use of a second, independent data set, the so-called *validation data*, which has not been used to train the model.

- The traditional way to carry out the bias-variance tradeoff is to try different candidate neural networks, with different numbers of hidden neurons. The performance of the trained networks can then be computed on the validation data, and the best network is selected.

- By specifying a *regularization* parameter larger than zero, a regularized performance index is minimized instead of the original MSE. This type of regularization is often called *weight decay* in connection with neural networks.
- By submitting the validation data in the call to `NeuralFit`, you apply *stopped search*. The MSE is minimized with respect to the training data, but the obtained parameter estimate is the one that gave the best performance on the validation data at some intermediate iteration during the training.

The last two of these techniques make effective use of only a subset of the parameters in the network. Therefore, the number of *efficient* parameters becomes lower then the nominal number of parameters. This is described in the following two sections.

### 7.5.1 Regularization

You can apply regularization to the training by setting the option `Regularization` to a positive number. Then the criterion function minimized in the training becomes

$$W_N^{\delta}(\theta) = V_N(\theta) + \delta\,\theta^T\theta \qquad (1)$$

instead of $V_N(\theta)$, given in Section 2.5.3, Training Feedforward and Radial Basis Function Networks, where $\delta$ is the number you specify with the option. The second term in Equation 7.1 is called the regularization term, which acts like a spring pulling the parameters toward the origin. The spring only marginally influences those parameters that are important for the first term $V_N(\theta)$, while parameters that do not have any large impact on $V_N(\theta)$ will be pulled to the origin by the regularization. This second class of parameters is effectively excluded from the fit, thus reducing the network's flexibility or, equivalently, reducing the number of efficient parameters. You use $\delta$ to control the importance of a parameter to the training process.

The critical issue in using regularization is to choose a good value of $\delta$. This may be done by trial and error using validation data. Typically, you try several different $\delta$ values and compare the results.

### 7.5.2 Stopped Search

Stopped search refers to obtaining the network's parameters at some intermediate iteration during the training process and not at the final iteration as is normally done. Like the regularization, this is a way to limit the number of used parameters in the network. During the training the efficient number of parameters grows gradually and eventually becomes equal to the nominal number of parameters at the minimum of the MSE. Using validation data, it is possible to identify an intermediate iteration where the parameter values yield a minimum MSE. At the end of the training process the parameter values at this minimum are the ones used in the delivered network model.

In the following example, the performance of this stopped search technique is compared to that of a fully trained model.

### 7.5.3 Example

In this small example, you will see how stopped search and regularization can be used to handle the bias-variance tradeoff. The example is in a one-dimensional space so that you can look at the function. In Section 8.2.4, Bias-Variance Tradeoff—Avoiding Overfitting, a larger example is given.

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

Some additional standard add-on packages are needed in the example. Load them.

```
In[2]:= << Statistics`ContinuousDistributions`
        << Statistics`DataManipulation`
```

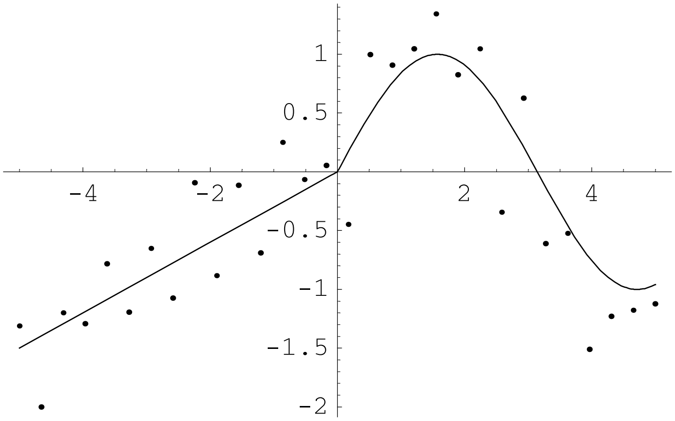
To generate data, the true function has to be defined. It is called `trueFunction` here and you can change it and repeat the calculations to obtain several different examples. You can also modify the number of data samples to be generated, the noise level on the data, and the number of hidden neurons in the model.

Generate noisy data and look at it.

```
In[4]:= Ndata = 30;
        trueFunction[xx_] := If[xx < 0, 0.3 xx, Sin[xx]]
        x = Table[{N[i]}, {i, -5, 5, 10 / (Ndata - 1)}];
        y = Map[trueFunction, x, {2}] + RandomArray[NormalDistribution[0, 0.4], {Ndata, 1}];
```

Look at the data and the true function.

```
In[8]:= Show[Plot[trueFunction[x], {x, -5, 5}, DisplayFunction → Identity],
         ListPlot[RowJoin[x, y], DisplayFunction → Identity],
         DisplayFunction → $DisplayFunction]
```



To apply stopped search, you need validation data. Thus the available data is divided into two sets: training data and validation data.

Divide the data into training data and validation data.

```
In[9]:= xt = x[[Range[1, Ndata, 2]]];
        yt = y[[Range[1, Ndata, 2]]];
        xv = x[[Range[2, Ndata, 2]]];
        yv = y[[Range[2, Ndata, 2]]];
```

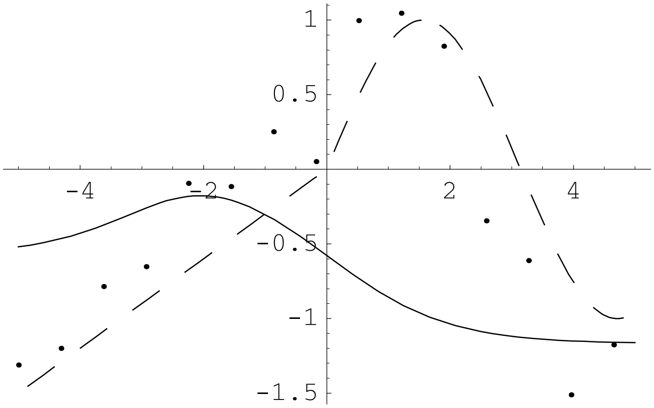
The default initialization of FF and RBF networks fits the linear parameters of the network using the least-squares algorithms, as described in Section 5.1.1, InitializeFeedForwardNet, and Section 6.1.1, InitializeRBFNet. If the network is overparameterized, this may lead to very large values of the linear parameters. The large values of the linear parameters can then cause problems in the training, especially if you want to use regularization and stopped search. There are two straightforward ways to handle this. The first one is to use regularization in the initialization, which keeps the parameter values smaller. The second way, which is used here, is to choose `RandomInitialization→LinearParameters` so that the least-squares step is skipped and the linear parameters are chosen randomly.

Initialize an FF network.

```
In[13]:= fdfrwrd =
          InitializeFeedForwardNet[xt, yt, {4}, RandomInitialization → LinearParameters]

Out[13]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2003, 7, 28, 21, 12, 35.4777536},
           OutputNonlinearity → None, NumberOfInputs → 1}]
```

Look at the initialized network, the true function, and the training data.

```
In[14]:= Show[Plot[{fdfrwrd[{a}], trueFunction[a]}, {a, -5, 5},
           PlotStyle → {{}, {Dashing[{0.05, 0.05}]}}, DisplayFunction → Identity],
          ListPlot[RowJoin[xt, yt], DisplayFunction → Identity],
          DisplayFunction → $DisplayFunction]
```

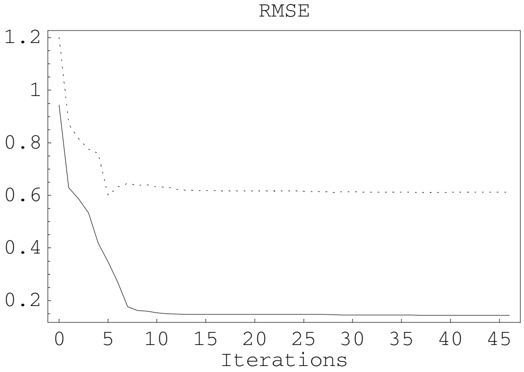


It is now time to train the network. Validation data is submitted so that stopped search can be applied. If you have not set `CriterionLog` to `False`, the value of the criterion $\sqrt{W_N^\delta(\theta)}$ for training data and RMSE $\sqrt{V_N(\theta)}$ for validation data are written out during the training process. At the end of the training process, a message is given indicating at which iteration the RMSE reaches the minimum for the validation data used. It is the parameters at that iteration that are returned and define the network model. If `CriterionPlot` is not set to `False`, you also get a plot at the end of the training showing the decrease of $\sqrt{W_N^\delta(\theta)}$ for training data and RMSE $\sqrt{V_N(\theta)}$ for validation data.

The separable algorithm, which is described in Section 7.6, Separable Training, fits the linear parameters in each step of the iterative training with the least-squares algorithm. Hence, for a reason similar to that of initializing the network without using least-squares for the linear parameters, it might be better to carry out the training without the separable algorithm. In this way, extremely large parameter values are avoided. This is done by setting `Separable→False`.

Train the network.

```
In[15]:= {fdfrwrd1, fitrecord} = NeuralFit[fdfrwrd, xt, yt, xv, yv, 50, Separable → False];
```

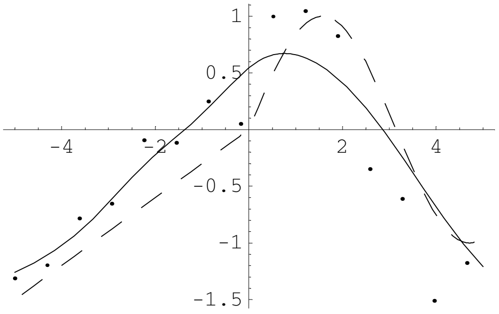


```
NeuralFit::StoppedSearch :
 The net parameters are obtained by stopped search using the supplied
   validation data. The neural net is given at the 5th training iteration.
```

The obtained function estimate using stopped search can now be plotted together with the true function and the training data.

Plot the obtained estimate.

```
In[16]:= Show[Plot[{fdfrwrd1[{a}], trueFunction[a]}, {a, -5, 5},
           PlotStyle → {{}, {Dashing[{0.05, 0.05}]}}, DisplayFunction → Identity],
         ListPlot[RowJoin[xt, yt], DisplayFunction → Identity],
         DisplayFunction → $DisplayFunction]
```



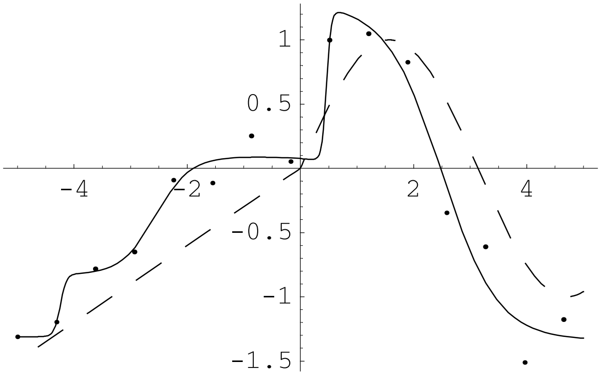
If no validation data had been submitted, you would have received the parameters at the final iterations in the model. These parameters can be extracted from the training record and put into the network model. In that way, you can compare the result obtained with stopped search shown in the plot with the result you would have received without stopped search.

Put in the parameters at the final iteration.

```
In[17]:= parameters = ParameterRecord /. fitrecord[[2]];
         fdfrwrd1[[1]] = Last[parameters];
```

Look at the estimate without using stopped search.

```
In[19]:= Show[Plot[{fdfrwrd1[{a}], trueFunction[a]}, {a, -5, 5},
           PlotStyle → {{}, {Dashing[{0.05, 0.05}]}}, DisplayFunction → Identity],
          ListPlot[RowJoin[xt, yt], DisplayFunction → Identity],
          DisplayFunction → $DisplayFunction]
```



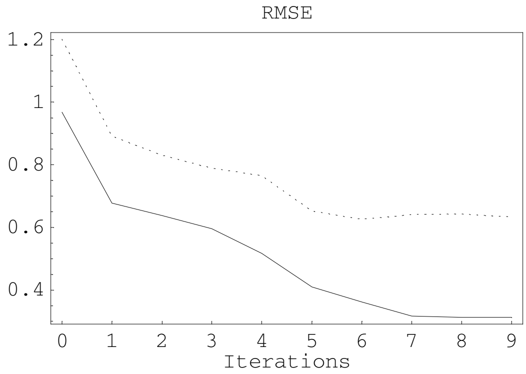
Compare this with the earlier plot where stopped search was used.

Now consider regularization instead of stopped search. As explained already, the linear parameters of an initialized network might become very large when using the default initialization due to the least-squares step. This may cause problems when regularization is applied, because the regularization term of the criterion dominates if the parameters are extremely large. As mentioned, there are two ways to handle this: using regularization also in the initialization, or skipping the least-square step.

For the same reason, problems may also arise when using the separable algorithm together with regularization. To avoid that, you can set `Separable→False` in the training. You can supply validation data as before. By inspecting the RMSE criterion on the validation data, you can see if the regularization parameter is of appropriate size. Too small of a value gives a validation error that increases toward the end of the training.

Train the network using regularization.

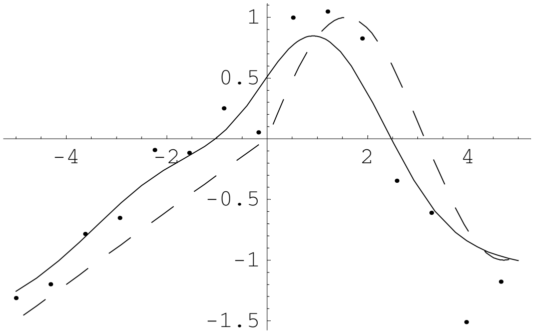
```
In[20]:= {fdfrwrd1, fitrecord} =
           NeuralFit[fdfrwrd, xt, yt, xv, yv, 30, Regularization → 0.001, Separable → False];
```



```
NeuralFit::StoppedSearch :
 The net parameters are obtained by stopped search using the supplied
   validation data. The neural net is given at the 6th training iteration.
```

Look at the estimate obtained using regularization.

```
In[21]:= Show[Plot[{fdfrwrd1[{a}], trueFunction[a]}, {a, -5, 5},
           PlotStyle → {{}, {Dashing[{0.05, 0.05}]}}, DisplayFunction → Identity],
          ListPlot[RowJoin[xt, yt], DisplayFunction → Identity],
          DisplayFunction → $DisplayFunction]
```



Compare the result with the ones obtained using stopped search and just normal learning.

You can modify the design parameters and repeat the example. Try networks of different sizes and with several layers. Try an RBF network.

## 7.6 Separable Training

Separable training can be used when the neural network model is linear in some of the parameters. The separable algorithm gives a numerically better-conditioned minimization problem that is easier to solve. Therefore, by using a separable algorithm, the training is likely to converge to the solution in fewer training iterations. If the neural network has several outputs, the computational burden per iteration will also be less, which will speed up the training further.

You can indicate if the separable algorithm should be used with the option `Separable`. The default value is `Automatic`, which means that the separable algorithm will be used whenever possible for all training algorithms except the backpropagation algorithm. In the following cases, the separable algorithm cannot be used:

- If there is a nonlinearity at the output; that is, if `OutputNonlinearity` is set to anything else but `None`
- If some of the parameters are fixed in the training

The separable algorithm is illustrated with two examples. The first example is very simple: it has only two parameters, so that the result can be illustrated in a surface plot. The second example is of a more realistic size.

### 7.6.1 Small Example

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

The following standard package is needed for the surface plot of the criterion.

```
In[2]:= << Graphics`Graphics3D`
```

First, the “true” function has to be defined, which is then used to generate the data. To make the example small, with one linear and one nonlinear parameter, a small FF network is chosen. It consists of one input and one output, without any hidden layers and it has no bias parameter. This small network has only two parameters and these are chosen to be (2, 1).

Define the “true” function.

```
In[3]:= fdfrwrd = InitializeFeedForwardNet[{{1}}, {{1}},
           {1}, RandomInitialization → True, BiasParameters → False];
        fdfrwrd[[1]] = {{{{2.}}, {{1.}}}};
```

Generate data with the true function.

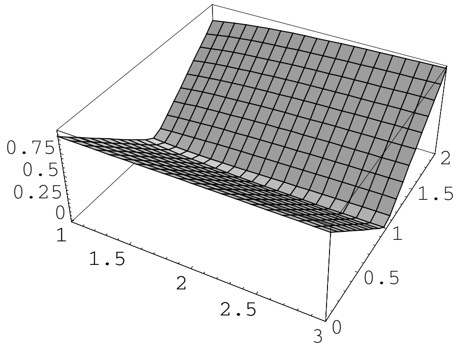
```
In[5]:= Ndata = 50;
        x = Table[{N[i]}, {i, 0, 5, 10 / (Ndata - 1)}];
        y = fdfrwrd[x];
```

To illustrate the result with plots you need the following function, which computes the criterion of fit, described in Section 2.5.3, Training Feedforward and Radial Basis Function Networks.

```
In[8]:= criterion[a_, b_] := (fdfrwrd[[1]] = {{{{a}}, {{b}}}};
          Sqrt[(Transpose[#].#) &[y - fdfrwrd[x]][[1, 1]] / Length[x]])
```

Look at the criterion as a function of the two parameters.

```
In[9]:= surf = Plot3D[criterion[a, b], {a, 1, 3}, {b, 0, 2}, PlotPoints → 20]
```



The parameter that has the largest influence on the criterion is the linear parameter. The separable algorithm minimizes the criterion in the direction of the linear parameter in each iteration of the algorithm so that the iterative training follows the valley. This will be obvious from the following computations.

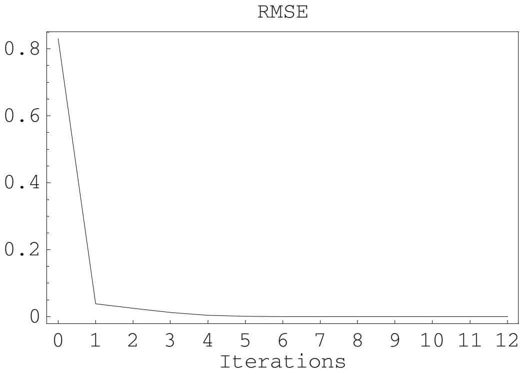
The network is now initialized at the point (1.1, 2) in the parameter space, and separable training is compared with the nonseparable training. This is done using the default Levenberg-Marquardt training algorithm. You can repeat the example using the Gauss-Newton or the steepest descent training by changing the option `Method`.

Initialize the network and insert the true parameter values.

```
In[10]:= fdfrwrd2 = fdfrwrd;
         fdfrwrd2[[1]] = {{{{1.1}}, {{2.}}}};
```

Train with the separable algorithm.

```
In[12]:= {fdfrwrd3, fitrecord} = NeuralFit[fdfrwrd2, x, y];
```

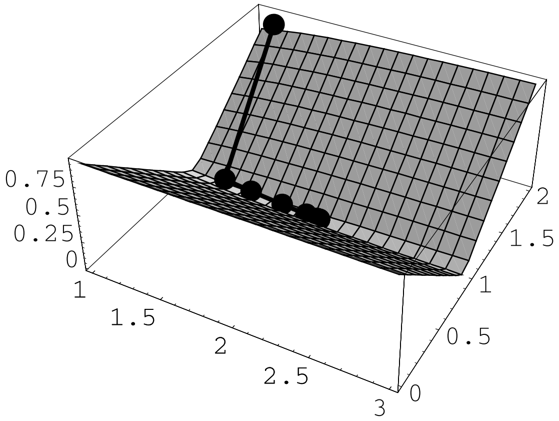


Form a list of the trajectory in the parameter space.

```
In[13]:= trajectory =
           Transpose[Append[Transpose[Map[Flatten, (ParameterRecord /. fitrecord[[2]])]],
             (CriterionValues /. fitrecord[[2]]) + 0.05]];
```

Form plots of the trajectory and show it together with the criterion surface.

```
In[14]:= trajectoryplot = ScatterPlot3D[trajectory,
            PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
         trajectoryplot2 = ScatterPlot3D[trajectory, PlotJoined → True,
            PlotStyle → Thickness[0.01], DisplayFunction → Identity];
         Show[surf, trajectoryplot, trajectoryplot2]
```

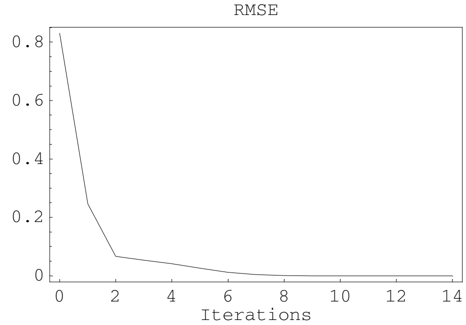


As you can see from the plot, the parameter estimate is at the bottom of the valley already at the first iteration. The minimization problem has been reduced to a search in one dimension, along the valley, instead of the original two-dimensional space. The training converged after approximately five iterations.

The calculations can now be repeated but without using the separable algorithm.

Train without the separable algorithm.

```
In[17]:= {fdfrwrd3, fitrecord} = NeuralFit[fdfrwrd2, x, y, Separable → False];
```

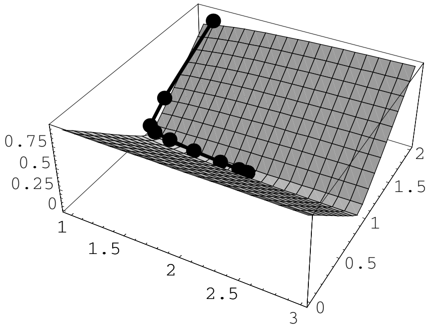


Form a list of the trajectory in the parameter space.

```
In[18]:= trajectory =
           Transpose[Append[Transpose[Map[Flatten, (ParameterRecord /. fitrecord[[2]])]],
             (CriterionValues /. fitrecord[[2]]) + 0.05]];
```

Form plots of the trajectory and show it together with the criterion surface.

```
In[19]:= trajectoryplot = ScatterPlot3D[trajectory,
            PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
         trajectoryplot2 = ScatterPlot3D[trajectory, PlotJoined → True,
            PlotStyle → Thickness[0.01], DisplayFunction → Identity];
         Show[surf, trajectoryplot, trajectoryplot2]
```



Without the separable algorithm, the training is slowed down a little. Several iterations are necessary before the bottom of the valley is reached. Also the convergence along the valley is somewhat slower: the algorithm needs about eight iterations to converge.

### 7.6.2 Larger Example

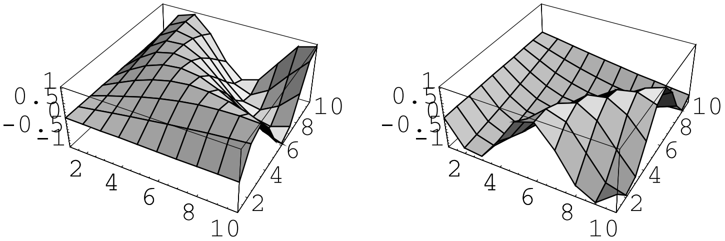
In this example, a function with two outputs is approximated.

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

Generate data and look at the two outputs.

```
In[2]:= Ndata = 10;
        x = Table[N[{i / Ndata, j / Ndata}], {i, 0, Ndata - 1}, {j, 0, Ndata - 1}];
        y1 = Map[Sin[10. #[[1]] #[[2]]] &, x, {2}];

        y2 = Map[Sin[10. (#[[1]] - 1) #[[2]]] &, x, {2}];
        Show[GraphicsArray[{ListPlot3D[y1, DisplayFunction → Identity],
           ListPlot3D[y2, DisplayFunction → Identity]}]]
        x = Flatten[x, 1];
        y = Transpose[{Flatten[y1], Flatten[y2]}];
```



An RBF network containing four neurons is chosen. You can modify the structure and repeat the example. You can also change it to an FF network.

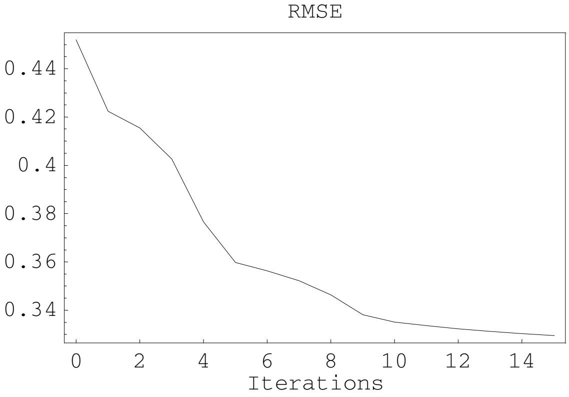
Initialize an RBF network.

```
In[9]:= rbf = InitializeRBFNet[x, y, 4, LinearPart → False]

Out[9]= RBFNet[{{w1, λ, w2}}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 46, 0},
          OutputNonlinearity → None, NumberOfInputs → 2}]
```

Different algorithms will be compared with respect to execution time and efficiency, that is, their RMSE rate of decrease. The Levenberg-Marquardt algorithm is tested first using the separable algorithm.

Train 15 iterations and find the time used.

```
In[10]:= {t, {rbf2, fitrecord}} = Timing[NeuralFit[rbf, x, y, 15]];
         t
```

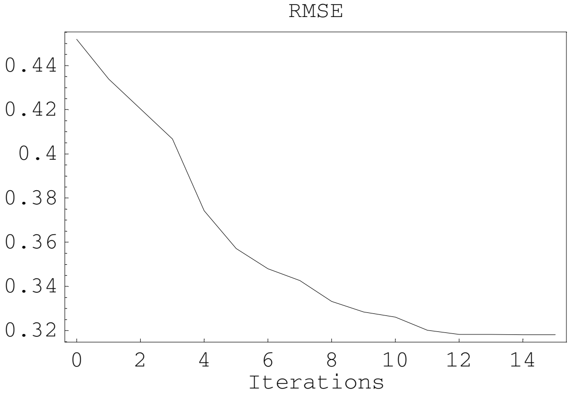


```
Out[11]= 0.971 Second
```

Now consider the case where the separable algorithm is not used.

Train 15 iterations and find the time used.

```
In[12]:= {t, {rbf2, fitrecord}} = Timing[NeuralFit[rbf, x, y, 15, Separable → False]];
         t
```



```
Out[13]= 0.831 Second
```

Compare the obtained fit illustrated in the previous two plots. Normally, the separable algorithm is more efficient, showing a larger rate of error decrease per iteration using approximately the same time.

You can repeat the example with the other training algorithms, such as Gauss-Newton and steepest descent, by changing the option `Method`.

## 7.7 Options Controlling Training Results Presentation

The following options are the same for all training commands in the *Neural Networks* package, with the exception of `HopfieldFit`.

- `CriterionLog` indicates if intermediate results during training should be displayed and logged in the training record.
- `CriterionLogExtN` indicates whether the intermediate results during training should be displayed in a separate notebook, which is the default, or in the current notebook.
- `ReportFrequency` indicates the interval of the intermediate results in the training record.

- `CriterionPlot` indicates whether the performance index should be shown at the end of the training.
- `MoreTrainingPrompt` indicates whether you want be asked if training should be continued at the last iteration.

Here are some examples of how the information given during training can be influenced by using these options.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

First some test data is loaded and an FF network is initialized.

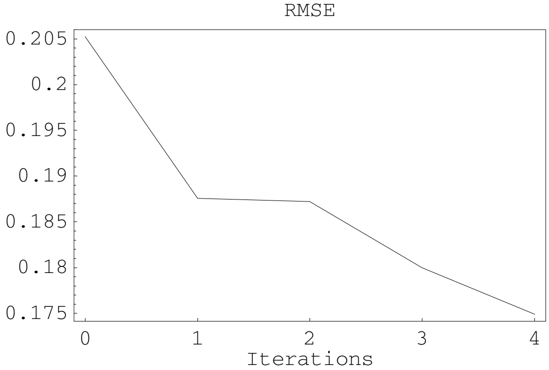
Load some test data and initialize an FF network.

```
In[2]:= <<onedimfunc.dat;
        fdfrwrd=InitializeFeedForwardNet[x,y,{4}]

Out[3]=  FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 46, 53},
           OutputNonlinearity → None, NumberOfInputs → 1}]
```

Using the default options gives a separate notebook where the criterion value and some control parameters are written out after each iteration of the training algorithm. At the end of the training, the decrease of the performance index is displayed in a plot in the current notebook. You only get the separate notebook if you re-evaluate the commands.

Train with the default options.

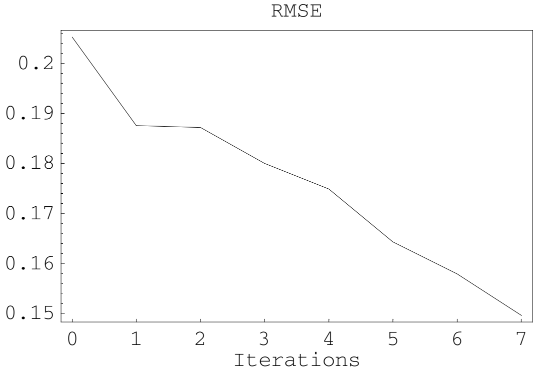
```
In[4]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 4];
```



If you work with large neural networks, and if you are uncertain of how many training iterations you need, it might be advantageous to set the `MoreTrainingPrompt→True` to avoid the initialization computation when you call the training command several times. With `MoreTrainingPrompt→True` you receive a question at the last iteration; you are asked to enter any number of additional training iterations before the training terminates. You can give any positive integer; if you answer anything else the training terminates.

Prompt for more training iterations before exiting the training command.

```
In[5]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 4, MoreTrainingPrompt → True];
```

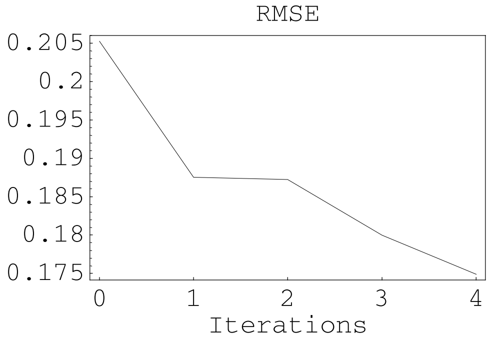


If you do not want any plot at the end of the training, you set `CriterionPlot→False`. You can also have the intermediate results in the current notebook instead. This is done by setting `CriterionLogExtN→False`. The interval of the intermediate results can be set with `ReportFrequency`.

Train without the criterion plot and with the training result in the current notebook with interval 2.

```
In[6]:= {fdfrwrd2, fitrecord} =
          NeuralFit[fdfrwrd, x, y, 4, CriterionLogExtN → False, ReportFrequency → 2];
```

```
Iteration RMSE      λ
==============================

    0.  0.2052

    2.  0.1872    3.72

    4.  0.1749    0.782
```



`ReportFrequency` also indicates the iteration interval with which parameter values are logged in the training record during the training. This is described in Section 7.8, The Training Record.

If you do not want any intermediate results at all during the training, you can switch them off by setting `CriterionLog→False`.

```
In[7]:= {fdfrwrd2, fitrecord} =
          NeuralFit[fdfrwrd, x, y, 4, CriterionPlot → False, CriterionLog → False];
```

The training process is speeded up a little by excluding the intermediate results or by setting the report frequency to a higher value.

You can reset the default using `SetOptions[`*function*`,` *option→value*`]`. This will change the default value for the entire *Mathematica* session.

Change the `CriterionLog` option so that no intermediate training results are displayed.

```
In[8]:= SetOptions[NeuralFit, CriterionLog → False]

Out[8]= {Compiled → True, CriterionLog → False, CriterionLogExtN → True,
         CriterionPlot → Automatic, FixedParameters → None, Method → Automatic,
         MinIterations → 3, Momentum → 0, MoreTrainingPrompt → False,
         PrecisionGoal → 6, Regularization → None, ReportFrequency → 1,
         Separable → Automatic, StepLength → Automatic, ToFindMinimum → {}}
```

By using `SetOptions`, you do not have to supply `CriterionLog→False` each time `NeuralFit` is called.

## 7.8 The Training Record

All neural network training functions, with the exception of `HopfieldFit`, return lists with two components. The first element is the trained network, and the second element is a training record, containing information logged during the training. The training record can be used to graphically illustrate the training, using the command `NetPlot`. The command works somewhat differently depending on which kind of network it is applied to. See the description in connection to each type of neural network. This will show how you can extract information directly from the training record.

First, some test data and a demonstration network are needed. Although an FF network is used in the example, training records from all other neural network models can be handled in the same way.

Load the *Neural Networks* package and test the data, then initialize and train an FF network.

```
In[1]:= << NeuralNetworks`

In[2]:= <<onedimfunc.dat;
        fdfrwrd=InitializeFeedForwardNet[x,y,{4}];
        {fdfrwrd2,fitrecord}=NeuralFit[fdfrwrd,x,y,10, CriterionPlot→False];
```

Look at the training record.

```
In[5]:=  fitrecord

Out[5]=  NeuralFitRecord[FeedForwardNet,
          ReportFrequency → 1, CriterionValues → {-values-},
          CriterionValidationValues → {-values-}, ParameterRecord → {-parameters-}]
```

The head of the training record depends on the type of neural network training it describes. For FF and RBF networks, trained with `NeuralFit`, the head is `NeuralFitRecord`.

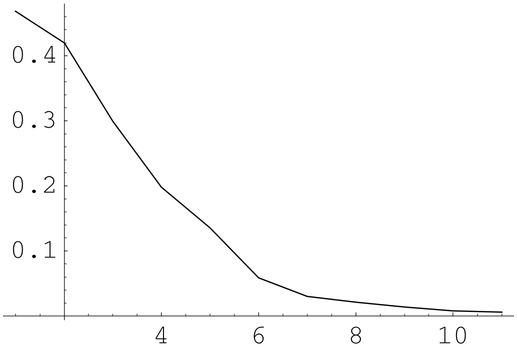
The first component of the training record contains a copy of the trained network. The second component is a list of rules. The left sides of the rules are used as pointers indicating different information.

- `ReportFrequency` indicates the value of this option when the network was trained. That is, it indicates the interval of training iterations at which the information is logged in the rules that follow.
- `CriterionValues` points at a list containing the performance index after each iteration. It can easily be extracted and plotted.
- `CriterionValidationValues` contains a list of the performance index on validation data. Note that this only holds if validation data was submitted in the call, and that can only be done with the `NeuralFit` command. See Section 7.5, Regularization and Stopped Search for more information.
- `ParameterRecord` contains a list of the parameters used during the training. The elements in the list have the same structure as the first element of the neural network model.

With these specifications you can extract and use the intermediate results of the training in any way you like.

Extract the criterion decrease and plot it.

```
In[6]:= ListPlot[CriterionValues /. fitrecord[[2]], PlotJoined → True]
```



Extract the list of parameters versus training iterations and check the length of the list.

```
In[7]:= par = ParameterRecord /. fitrecord[[2]];
        Length[par]

Out[8]= 11
```

The elements in the parameter list have the same structure as the parameter structure in the network. This means that the parameters at some stage of the training can be easily obtained by inserting the parameter values in the network. Suppose you want to obtain the network model you had after five training iterations. Then you have to extract the sixth element (recall that the initial parameters are at the first position of the list) and put it at the first position of the network.

Find the model after the fifth iteration by putting in the parameters obtained using `ParameterRecord`.

```
In[9]:= fdfrwrd2[[1]] = par[[6]]

Out[9]= {{{{-1.07433, 2.03512, 1.26074, 0.0173203}, {10.1864, -14.4612, -3.92809, 11.6643}},
          {{8.16393}, {0.895992}, {-7.00234}, {7.66623×10^6}, {-7.66617×10^6}}}}
```

The structure of the network is the same as before; only the values of the parameters have been changed.

Check the structure of the network.

```
In[10]:= NetInformation[fdfrwrd2]
```

```
Out[10]= FeedForward network created 2002-4-3 at 13:48. The
           network has 1 input and 1 output. It consists of 1 hidden
           layer with 4 neurons with activation function of Sigmoid type.
```

## 7.9 Writing Your Own Training Algorithms

The different options of NeuralFit offer you several training algorithms for FF and RBF networks. Its options give you further possibilities to modify the algorithms. Nevertheless, on occasion you may want to develop your own training algorithm. As long as the network parameter weights are stored in the correct way, as described in Section 13.1, Change the Parameter Values of an Existing Network, you may modify their values in whatever way you want. This is, in fact, enough to let you use all the capabilities of *Mathematica* to develop new algorithms. The advantage of representing the network in the standard way is that you can apply all other functions of the *Neural Networks* package to the trained network.

Many algorithms are based on the derivative of the network with respect to its parameters, and SetNeuralD and NeuralD help you compute it in a numerically efficient way. This command is described in the following.

SetNeuralD produces the code for NeuralD. Therefore, SetNeuralD has to be called first, then NeuralD is used to compute the derivative. The advantage of this procedure is that SetNeuralD optimizes the symbolic expression for the derivative so that the numerical computation can be performed as fast as possible.

Notice that SetNeuralD only has to be called once for a given network structure. It does not have to be recalled if the parameters have changed; SetNeuralD only needs to be called if you change the network structure. Typically, SetNeuralD is called at the beginning of a training algorithm and only NeuralD is used inside the training loop.

Note also that NeuralD does not perform any tests of its input arguments. The reason for this is that it is intended for internal use. Instead, you have to add the tests yourself in the beginning of the training algorithm.

This use is illustrated in the following example.

Read in the *Neural Networks* package.

```
In[1]:=  << NeuralNetworks`
```

Load some data.

```
In[2]:=  << one2twodimfunc.dat;
```

Check the dimensions of the data.

```
In[3]:=  Dimensions[x]
         Dimensions[y]

Out[3]=  {20, 1}

Out[4]=  {20, 2}
```

There are 20 data samples available and there are one input and two outputs.

Initialize an RBF network.

```
In[5]:=  net = InitializeRBFNet[x, y, 2, LinearPart → False]

Out[5]=  RBFNet[{{w1, λ, w2}}, {Neuron → Exp, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 13, 48, 43},
           OutputNonlinearity → None, NumberOfInputs → 1}]
```

Generate the code to calculate the derivative.

```
In[6]:=  SetNeuralD[net];
```

Compute the derivative.

```
In[7]:=  der = NeuralD[net, x[[Range[3]]]]

Out[7]=  {{{0.0187566, 0.108421, 0.318221, 1.7893, 0.0124692, 0., 0.841608, 0., 1., 0.},
           {-0.0124813, -0.0522312, -0.211755,
            -0.861988, 0., 0.0124692, 0., 0.841608, 0., 1.}},
          {{0.0347679, 0.0930977, 0.540381, 1.24297, 0.0252298, 0., 0.893275, 0., 1., 0.},
           {-0.0231357, -0.0448493, -0.359587,
            -0.598794, 0., 0.0252298, 0., 0.893275, 0., 1.}},
          {{0.060082, 0.0745408, 0.848315, 0.760255, 0.0479941, 0., 0.936261, 0., 1., 0.},
           {-0.0399805, -0.0359096, -0.564497,
            -0.366249, 0., 0.0479941, 0., 0.936261, 0., 1.}}}
```

The first input argument is the network. It must have the same structure as it had when `SetNeuralD` was called, but the parameter values may have changed. The second input argument should be a matrix with one numerical input vector on each row. The output is better explained by looking at the dimension of its structure.

```
In[8]:=  Dimensions[der]

Out[8]=  {3, 2, 10}
```

The first index indicates the data sample (the derivative was computed on three input data), the second index indicates the output (there are two outputs of the network), and the third index indicates the parameters (there are obviously 10 parameters in the network). The derivatives with respect to the individual parameters are placed in the same order as the parameters in the flattened parameter structure of the network, that is, the position in the list `Flatten[`*net*`[[1]]]`.

If some parameters should be excluded from the fit, you may indicate that in the call to `SetNeuralD`. Then `SetNeuralD` tests for any possible additional simplifications so that `NeuralD` becomes as fast as possible. Parameters are excluded using `FixedParameters`.

Exclude four parameters from the fit.

```
In[9]:=  SetNeuralD[net, FixedParameters → {2, 5, 6, 8}];
```

Calculate the derivative of the remaining six parameters.

```
In[10]:= der = NeuralD[net, x[[Range[3]]]];
         Dimensions[der]

Out[11]= {3, 2, 6}
```

Compared to earlier, there are now only six components in the third level. They correspond to the six parameters, and the fixed four are considered to be constants.

## 7.10 Further Reading

The following are standard books on minimization:

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

R. Fletcher, *Practical Methods of Optimization*, Chippenham, Great Britain, John Wiley & Sons, 1987.

Stopped search and the separable algorithms are explained in the following articles:

J. Sjöberg and L. Ljung, “Overtraining, Regularization, and Searching for Minimum with Application to Neural Nets”, *Int. J. Control*, **62** (6), 1995, pp. 1391–1407.

J. Sjöberg and M. Viberg, “Separable Non-linear Least-Squares Minimization—Possible Improvements for Neural Net Fitting”, in *IEEE Workshop in Neural Networks for Signal Processing*, Amelia Island Plantation, Florida, Sep. 24–26, 1997, pp. 345–354.

This standard book on neural networks may also be of interest:

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, NY, Macmillan, 1999.

# 8 Dynamic Neural Networks

This section demonstrates how the *Neural Networks* package can be used to estimate models of dynamic systems and time series using input and output data from the system.

A tutorial on estimation of dynamic systems and time series was given in Section 2.6, Dynamic Neural Networks. In Section 8.1, Dynamic Network Functions and Options, the functions and their options to estimate dynamic neural network models are given, and in Section 8.2, Examples, you find examples illustrating the use of the commands.

## 8.1 Dynamic Network Functions and Options

This section introduces the commands you need to work with dynamic neural networks. Examples can be found in Section 8.2, Examples.

### 8.1.1 Initializing and Training Dynamic Neural Networks

There are two commands to obtain dynamic model structures. `NeuralARXFit` is used to model dynamic systems with input signals, and `NeuralARFit` is used to model time series where there is no external input signal. As the names indicate, they produce neural ARX and neural AR models as described in Section 2.6, Dynamic Neural Networks. The term *neural AR(X)* will be used when either a neural ARX or a neural AR model can be considered. There is no restriction on the dimensionality of the input and output signals; that is, the package supports multi-input, multi-output (MIMO) models. In contrast to other neural network types in the *Neural Networks* package, there are no specific commands for initialization. Instead, if you only want to initialize the dynamic neural network, you use the training commands with 0 training iterations.

Both commands call `NeuralFit` internally (see Section 7.1, NeuralFit). Algorithmic details are described in Chapter 7, Training Feedforward and Radial Basis Function Networks. There are quite a few input arguments, and depending on how they and the options are chosen, a large variety of different models and algorithms may be obtained.

`NeuralARXFit[`*u*, *y*, {*na*, *nb*, *nk*}, *type*, *nh*`]`
initializes and estimates a neural ARX model using
input data *u*, output data *y*, regressor {*na, nb, nk*},
and neural network of *type* with hidden neurons *nh*

`NeuralARXFit[`*u*, *y*, {*na*, *nb*, *nk*}, *type*, *nh*, *uv*, *yv*, *iterations*`]`
initializes and estimates a neural ARX model using
input data *u*, output data *y*, regressor {*na, nb, nk*},
and neural network of *type* with hidden neurons *nh*,
also submits validation data *uv* and *yv,*
and specifies the number of training iterations

`NeuralARXFit[`*u*, *y*, *NeuralARX*, *uv*, *yv*, *iterations*`]`
continues the training of an already-existing neural ARX
model; the arguments *uv*, *yv*, and iterations are optional

Initializing and training neural ARX models.

`NeuralARXFit` returns a list of two variables. The first variable is the trained neural ARX model, with head `NeuralARX`, and the second is a training record.

An existing neural ARX model can be submitted for more training by setting the argument *NeuralARX* equal to the neural ARX model or its training record.

The format of the training data *u* and *y* is described in Section 3.2, Package Conventions. The regressor is specified as indicated in Section 2.6, Dynamic Neural Networks. The number of lagged outputs to be included in the regressor is given by $n_a$, and the number of lagged inputs is given by $n_b$. The delay is given by $n_k$. For SISO systems, these indices are given as three (positive) integers. For MIMO systems, each one of them should be a list with one component for each input or output.

The neural network type indicated by the argument *type* should be either `FeedForwardNet` or `RBFNet`. The number of neurons (and layers for FF networks) is specified by *nh* as described in Section 5.1.1, InitializeFeedForwardNet, and Section 6.1.1, InitializeRBFNet. A linear model is obtained by choosing *type* = `FeedForwardNet` and *nh* = { }.

Time-series models are obtained with `NeuralARFit` in similar fashion to `NeuralARXFit`. The only difference is that only the number of lagged outputs have to be indicated in the call.

| | |
|---|---|
| `NeuralARFit`[*y*, *na*, *type*, *nh*] | initializes and estimates a neural AR model using output data *y*, regressor indicated by *na*, and the neural network *type* with hidden neurons indicated by *nh* |
| `NeuralARFit`[*y*, *na*, *type*, *nh*, *yv*, *iterations*] | initializes and estimates a neural AR model using output data *y*, regressor indicated by *na*, and the neural network *type* with hidden neurons indicated by *nh*, also submits validation data *yv* and specifies the number of training iterations |
| `NeuralARFit`[*y*, *NeuralAR*, *yv*, *iterations*] | trains an existing neural AR model further, where the arguments *yv* and *iterations* are optional |

Initializing and estimating neural AR models.

`NeuralARFit` returns two arguments just as `NeuralARXFit` does; the first one is a model of type `NeuralAR`, and the second argument is the training record.

An existing neural ARX model can be submitted for more training by setting the argument *Neural ARX* equal to the neural ARX model or its training record.

In addition to the compulsory arguments of `NeuralARXFit` and `NeuralAR`, you can also specify a number of additional arguments:

- The number of training iterations to be applied in the call to `NeuralFit`.
- Validation data *uv* and *yv*, so that the RMSE can be computed on validation data after each training iteration. The returned trained model is the model that gives the smallest RMSE on the validation data. This can be the model at an iteration before the last one. In that case the model is obtained by *stopped search,* which can be useful to avoid overfitting. See Section 7.5, Regularization and Stopped Search, for more details.

`NeuralARXFit` and `NeuralARFit` have no options of their own, but because they rely on initialization and training of FF or RBF networks, you can submit any option applicable to `InitializeRBFNet` or `InitializeFeedForwardNet` and `NeuralFit`.

During the training, intermediate results are displayed in a separate notebook, which is created automatically. After each training iteration you can see the value of the RMSE. The step length control parameter of the minimization algorithm is also shown; see Chapter 7, Training Feedforward and Radial Basis Function Networks. If you submit validation data to `NeuralFit`, then you also get the RMSE on this data set.

At the end of the training, a plot is displayed showing the RMSE reduction as a function of the iteration number.

Using the various options of `NeuralFit`, as described in Section 7.7, Options Controlling Training Results Presentation, you can change the way the training results are presented.

There are often different warning messages given at the end of the training, providing some feedback as to the success of the training. By inspecting the error reduction plot, you can usually tell whether more iterations are necessary. Specifically, if the curve exhibits a reasonable downward slope by the end of the training, you should consider continuing the training.

The first output argument, the neural AR(X) model, consists of two components. The first component is the FF or the RBF network, which the model is based on. The second component is a replacement rule, `Regressor->{` $n_a$, $n_b$, $n_k$`}`, specifying the regressor by the three indices $n_a$, $n_b$, and $n_k$.

The second output argument of `NeuralAR(X)Fit` is the training record. Its head is `NeuralFitRecord`, which contains two components. The first component is a copy of the trained neural AR(X) model and the second component is a list of rules specifying intermediate values of parameters and the RMSE during the training. In Section 7.8, The Training Record, you learn how to extract the information from the training record. You can also use `NetPlot` to plot some information.

### 8.1.2 NetInformation

The command `NetInformation` can be used to obtain some written information about the neural dynamic model.

| | |
|---|---|
| `NetInformation[`*NeuralAR*(*X*)`]` | presents some information about the network model |

`NetInformation`.

This information specifies the regressor and the FF or RBF network used in the model.

### 8.1.3 Predicting and Simulating

Unlike the other neural network types, a dynamic model cannot be applied directly on a data input vector. The reason for this is that the models are dynamic and you have to consider sequences of data to yield simulations and predictions. The commands for this are described in the following.

The one-step ahead prediction $\hat{y}(t \mid t-1)$ is obtained as described in Section 2.6, Dynamic Neural Networks. It is based on measured outputs up to time $t-1$.

It is often interesting to check if the model is capable of performing predictions several time-steps ahead; that is, to increase the *prediction horizon*. Consider, for example, the two-step ahead prediction, $\hat{y}(t \mid t-2)$, where past values only up to $y(t-2)$ are given. This is accomplished by using the estimate $\hat{y}(t-1 \mid t-2)$ in place of the missing $y(t-1)$ value to do the normal one-step prediction. This procedure may be similarly extended to obtain larger prediction horizons.

If the prediction horizon is infinite, it is said that the model is being used in *simulation*. Then, measured outputs are not used at all, and all outputs in the regressor are replaced by model outputs.

Predictions can be obtained with the command `NetPredict`.

| | |
|---|---|
| `NetPredict[`*u*`,`*y*`,`*NeuralARX*`]` | predicts the output signal *y* using the model *NeuralARX* and input data *u*; default prediction horizon: 1 |
| `NetPredict[`*y*`,`*NeuralAR*`]` | predicts the time series *y* using the model *NeuralAR*; default prediction horizon: 1 |

Predicting future outputs.

The prediction error, which is the difference between predicted and true output, can be obtained using `NetPredictionError` in the following way.

| | |
|---|---|
| `NetPredictionError[`<br>`u, y, NeuralARX]` | computes the prediction error of the model *NeuralARX* applied to data *u* (input) and *y* (output); default prediction horizon: 1 |
| `NetPredictionError[y, NeuralAR]` | computes the prediction error of the model *NeuralAR* applied to time series data *y*; default prediction horizon: 1 |

Obtain the prediction error.

`NetPredict` and `NetPredictionError` have the following options.

| *option* | *default value* | |
|---|---|---|
| `PredictHorizon` | `1` | indicates the prediction horizon; if set to `Infinity`, a simulation is obtained |
| `MultiplePrediction` | `False` | obtains a series of predictions with horizons 1, 2, …, `PredictHorizon` if set to `True` |
| `InitialOutput` | `Automatic` | initial values of the lagged output values |
| `InitialInput` | `Automatic` | initial values of the lagged input values |

Options of `NetPredict` and `NetPredictionError`.

The two options `InitialOutput` and `InitialInput` can be used to set the values of the regressor at the beginning of the prediction. The default is to use the first available value in the submitted data; that is, $y(-1)$, $y(-2)$, … are set identically to $y(1)$ and similarly for $u(-1)$, $u(-2)$, ….

You can set `InitialOutput` and `InitialInput` to other values according to the following rules.

For SISO models you can set the options to:

- Real numbers; in which case all lagged inputs and outputs are set to these real numbers.
- Sequences of lagged values that are lists of real-valued numbers, such as $\{y(-1), y(-2), ..., y(-n_a)\}$ or $\{u(-1), u(-2), ..., u(-n_k - n_b + 1)\}$.

For MIMO models, $u(t)$ and $y(t)$ are vectors with the number of components equal to the number of inputs and outputs, respectively. Also the structural indices $n_a$, $n_b$, and $n_k$ are vectors. In this case the options `InitialOutput` and `InitialInput` can be set to:

- Real numbers; in which case the lagged values of all inputs and outputs are set to these real numbers.
- Lists of length equal to the number of inputs/outputs; in which case all lagged values of input/output number *m* are set to the value at position *m* in the list.
- A matrix; $\{y(-1),\ y(-2),\ \ldots,\ y(-\text{Max}[n_a])\}$ and $\{u(-1),\ u(-2),\ \ldots,\ u(\text{Max}[-n_k - n_b + 1])\}$, where $y(-1)$ is a list with one component for each output. The rest of the rows of the matrices are set correspondingly.

A model can be simulated using the command `NetSimulate`. A neural AR model can only be simulated if a noise signal *e* is submitted in the call.

| | |
|---|---|
| `NetSimulate[`*u*`,` *NeuralARX*`]` | simulates *NeuralARX* using the input data *u* |
| `NetSimulate[`*u*`,` *e*`,` *NeuralARX*`]` | simulates *NeuralARX* using<br>the input data *u* and noise data *e* |
| `NetSimulate[` *e*`,` *NeuralAR*`]` | simulates *NeuralAR* using noise data *e* |

Simulating the model.

`NetSimulate` has the following two options.

| *option* | *default value* | |
|---|---|---|
| `InitialOutput` | `0` | initial values of the lagged output values |
| `InitialInput` | `Automatic` | initial values of the lagged input values |

Options of `NetSimulate`.

These options were described earlier in this section in connection to the command `NetPredict`.

A very convenient command that may be used to evaluate a model is `NetComparePlot`. It simulates or predicts the output, depending on the option `PredictionHorizon`, and plots the result together with the supplied output signal. Therefore, you can visually inspect the performance of the neural model. The RMSE is also displayed.

`NetComparePlot[`$u$`,`$y$`,`*NeuralARX*`]`

using the model *NeuralARX* and the data $u$ (input) and $y$ (output), simulates/predicts and plots the output together with the true output; default: simulation

`NetComparePlot[`$y$`,`*NeuralAR*`]`

using the model *NeuralAR* and the data $y$ (output), predicts and plots the output together with the true output

Simulate or predict and compare with true output.

You can change the prediction horizon using the same option `PredictionHorizon` as in `NetPredict` and `NetPredictionError`. Often there are transients in the beginning of the data sequence due to the fact that initial input and output values in the regressor are incorrect; that is, the values used for $y(-1)$, $y(-2)$, … and $u(-1)$, $u(-2)$, … . You can remove the transients by excluding the first part of the data sequence from the comparison. This is done by giving the start and end numbers of the samples to be included in the option `ShowRange`. Another way to handle the transients is to set the initial values with the options `InitialOutput` and `InitialInput`. These options were described earlier in this passage in connection to the command `NetPredict`.

| *option* | *default value* | |
|---|---|---|
| `PredictHorizon` | `Infinity` | indicates prediction horizon |
| `ShowRange` | `All` | if a list containing two integers is given, then only the data samples between these values are included in the comparison |
| `InitialOutput` | `Automatic` | initial values of the lagged output values |
| `InitialInput` | `Automatic` | initial values of the lagged input values |

Options of `NetComparePlot`.

In addition to these options, you can supply any option of `MultipleListPlot` to modify the given plot.

### 8.1.4 Linearizing a Nonlinear Model

A nonlinear neural ARX or AR model can be linearized at any value of the regressor $x$. This is done with the command `LinearizeNet`.

| | |
|---|---|
| LinearizeNet[*NeuralAR*(*X*), *x*] | linearizes *NeuralAR* (*X*) at the regressor value *x* |

Linearize a dynamic model.

The result is a neural AR(X) model with a linear FF network as described in Section 5.1.5, LinearizeNet and NeuronDelete.

### 8.1.5 NetPlot—Evaluate Model and Training

The command `NetPlot` can be used to illustrate the performance of a model, and to evaluate the training of it. The command can be used in the same way as in FF and RBF networks. Depending on how the option `DataFormat` is set, the command can be used in very different ways. Here, the possibilities that are interesting in connection with dynamic models are presented. Please see Section 5.1.4, NetPlot, for other possibilities.

| | |
|---|---|
| NetPlot[*NeuralARX*, *u*,*y*] | illustrates *NeuralARX* using input and output data *u* and *y* |
| NetPlot[*NeuralAR*, *y*] | illustrates *NeuralAR* using time series data *y* |
| NetPlot[*fitrecord*, *u*, *y*] | evaluates the training of a *neural ARX* model |
| NetPlot[*fitrecord*, *y*] | evaluates the training of a *neural AR* model |

The `NetPlot` function.

When `NetPlot` is applied to training records from `NeuralARXFit` it takes the same options as when it is applied to training records from `NeuralFit`. The following two options have different default values:

| *option* | *default value* | |
|---|---|---|
| DataFormat | Automatic | if a model is submitted, the default is HiddenNeurons; if a training record is submitted, the default is ParameterValues |
| Intervals | 5 | depending on how DataFormat is chosen, indicates the number of bars in a bar chart or the period of animation of the training result: if a bar chart is animated, then the option should be a list of two integers; the first indicates the number of bars and the second the animation period |

Options of `NetPlot`.

You can also submit options to modify the graphical output. Depending on the chosen option for `DataFormat` the graphic is created by `BarChart`, `MultipleListPlot`, `ListPlot`, `Plot3D`, or `Histogram`.

If a dynamic neural network model is submitted, then the option `DataFormat` can be given any of the following values.

- `HiddenNeurons` gives the output values of the hidden neurons as a function of time when the model is used for prediction on the data.
- `LinearParameters` linearizes the model at each time instant at the value of the regressor vector and displays the linear parameters versus data.
- `FunctionPlot` plots the mapping using the range of the supplied data. This option can only be used if the model has a one- or two-dimensional regressor.
- `ErrorDistribution` gives a bar chart of the prediction errors. You can modify the presentation of the result using any of the options applicable to `Histogram`.

If you submit a training record to `NetPlot`, then the default value of `DataFormat` is `ParameterValues`. This gives a plot of the parameters versus training iterations.

Except for the default value, `ParameterValues`, you can also give `DataFormat` any of the possible values. You then obtain animations of the corresponding results as a function of the number of training iterations. The frequency of the plotting can be set with the option `Intervals`, which indicates the number of iterations between each plot.

### 8.1.6 MakeRegressor

Usually you do not have to care about the regressor of the dynamic model. The only thing you have to do is to specify it by choosing the three indices $n_a$, $n_b$, and $n_k$ when the model is defined. However, in case you would like to compute the regressor explicitly, you can do so with the following command.

| | |
|---|---|
| `MakeRegressor[u, y, {na, nb, nk}]` | gives a list of two components: the regressor and the output using input and output data and the specifications $n_a$, $n_b$, and $n_k$ |
| `MakeRegressor[y, {na}]` | gives a list of two components: the regressor and the output using time series data according to specification $n_a$ |

The `MakeRegressor` function.

`MakeRegressor` returns a list of two variables. The first variable is the regressor matrix and the second is the matrix of the output values.

The difference between the returned output values and the output values submitted in the call is that the length of the data sequence has been shortened by as many samples as the number of lags in the regressor.

## 8.2 Examples

Following are examples using measured data from a DC motor and a hydraulic actuator. Two subsequent examples show how you can handle the bias-variance problem in two different ways. The first way is to use fewer parameters in your neural network model. The second possibility is to modify the model structure so that a more appropriate model may be obtained. These are, therefore, alternatives to the approach discussed in Section 7.5, Regularization and Stopped Search.

### 8.2.1 Introductory Dynamic Example

In this first example you will see how the different commands of the *Neural Networks* package can be used to identify and evaluate dynamic models. You do not obtain exactly the same result if you repeat the example. This is due to the randomly chosen input signal and the random initialization of the neural network weights.

Read in the *Neural Networks* package and two standard add-on packages.

```
In[1]:=  << NeuralNetworks`
```

```
In[2]:= <<Statistics`ContinuousDistributions`
        <<Graphics`MultipleListPlot`
```

Generate a data set using a function defining the true system.

```
In[4]:= Ndata=600;
        u=RandomArray[NormalDistribution[0,3], {Ndata,2}];
        x=FoldList[Function[{xs,uin},{(uin[[1]]+uin[[2]]+0.6*xs[[1]]+0.8*xs[[2]])/(1+xs[[3
        ]]^2),0.7*xs[[2]]+uin[[2]],xs[[1]]}],{0,0,5}, Drop[u,-1]];
        y=x[[All,{1,2}]];
```

The input data is placed in *u* and the output data in *y*.

In a real situation, the data is measured and an approximation of the unknown true function generating the data is estimated using a dynamic neural network. This situation is now imitated and a neural network approximating the data generating function is estimated.

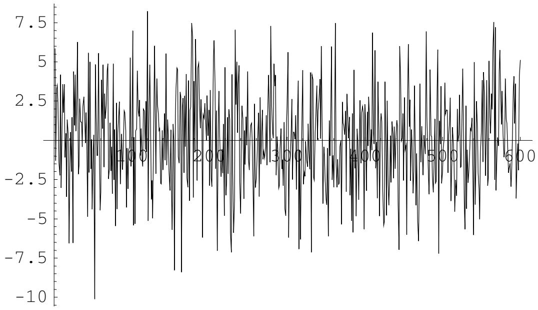
Check the dimensions of the data.

```
In[8]:= Dimensions[u]
        Dimensions[y]

Out[8]= {600, 2}

Out[9]= {600, 2}
```

There are 600 data samples available, and the plant has two inputs and two outputs. It is a good idea to look at the data before you start to estimate a model. From a plot you can see if the data look strange in some way that makes the training of a network hard.

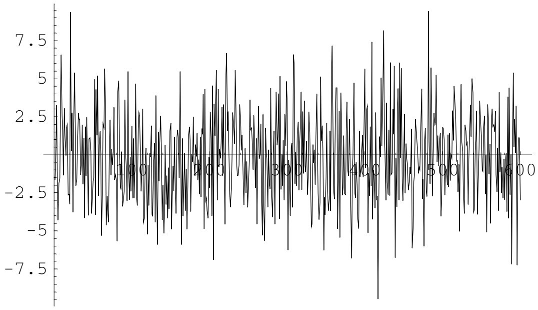
Look at the first input signal.

```
In[10]:= ListPlot[u[[All, 1]], PlotJoined → True, PlotRange → All]
```



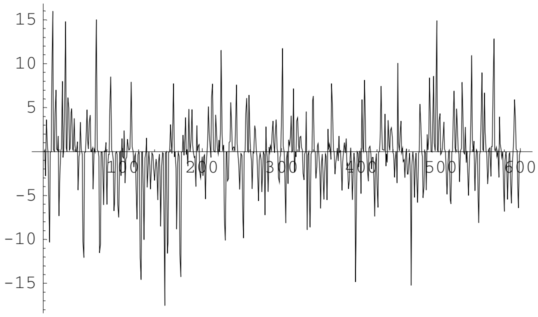
Look at the second input signal.

```
In[11]:= ListPlot[u[[All, 2]], PlotJoined → True, PlotRange → All]
```



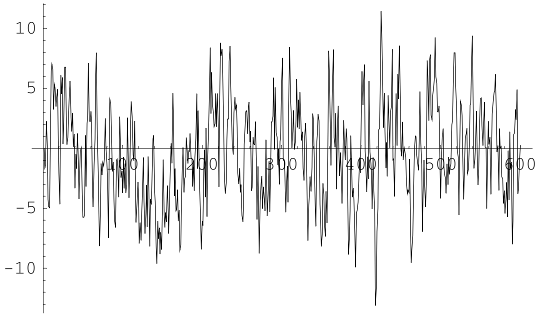
Look at the first output signal.

```
In[12]:= ListPlot[y[[All, 1]], PlotJoined → True, PlotRange → All]
```



Look at the second output signal.

```
In[13]:= ListPlot[y[[All, 2]], PlotJoined → True, PlotRange → All]
```



The first half of the data set is used for identification and the second half for validation of the model.

Divide the data into identification and validation data.

```
In[14]:= ue = u[[Range[Ndata / 2]]];
         ye = y[[Range[Ndata / 2]]];
         uv = u[[Range[Ndata / 2 + 1, Ndata]]];
         yv = y[[Range[Ndata / 2 + 1, Ndata]]];
```

In this example the true function is known and the regressor should be chosen to $x(t) = \{y_1(t-1), y_1(t-2), y_2(t-1)\, u_1(t-1), u_2(t-1)\}$, which is obtained by choosing $n_a = \{2, 1\}$, $n_b = \{1, 1\}$, and $n_k$ = {1, 1} as described in Section 2.6, Dynamic Neural Networks. In real situations, when the generating function is unknown, you usually have to find the best regressor by trial-and-error.

It is always good to start with a linear model. This is obtained by using an FF network without hidden neurons, as described in Section 5.1.1, InitializeFeedForwardNet. The performance of the linear model gives a quality measure that you want your nonlinear neural network model to beat.

Estimate a linear model.

```
In[18]:= {model1, fitrecord} = NeuralARXFit[ue, ye,
            {{2, 1}, {1, 1}, {1, 1}}, FeedForwardNet, {}, 0, CriterionPlot → False];
```

Find some information about the model.

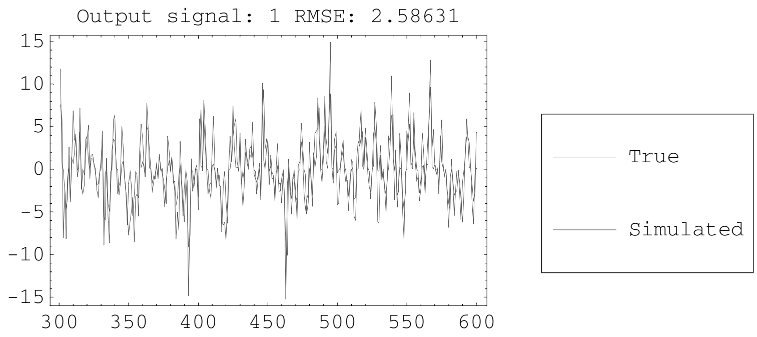
```
In[19]:= NetInformation[model1]
```

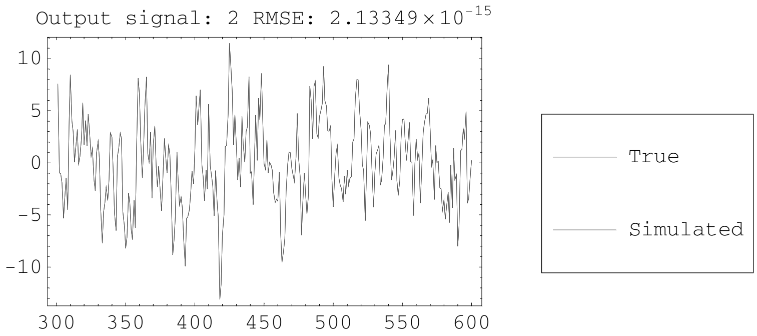
```
Out[19]= NeuralARX model with 2 input signals and 2 output signals. The regressor is
           defined by: na = {2, 1}, nb = {1, 1}, nk = {1, 1}. The mapping from
           regressor to output is defined by a FeedForward network created 2003-7-28
           at 21:14. The network has 5 inputs and 2 outputs. It has no hidden layer.
```

The command `NetComparePlot` is very convenient for evaluating dynamic models. Depending on which value you choose of the option `PredictionHorizon`, the model can be a simulator or a predictor. For each output, the model output is displayed together with the true output signal, and the root-mean-square error is given in the plot title. This type of test is only fair if you use fresh data, that is, the validation data for the comparison.

Obtain the one-step-ahead prediction with the linear model and compare it with the true output signal.

```
In[20]:= NetComparePlot[u, y, model1, ShowRange → {Ndata / 2 + 1, Ndata}, PredictHorizon → 1,
           PlotStyle → {Hue[.6], Hue[.8]}, PlotLegend → {"True", "Simulated"}]
```





By including the whole data set in the call, and then indicating the validation data with the option `ShowRange`, you avoid transients in the beginning of the plotted prediction.

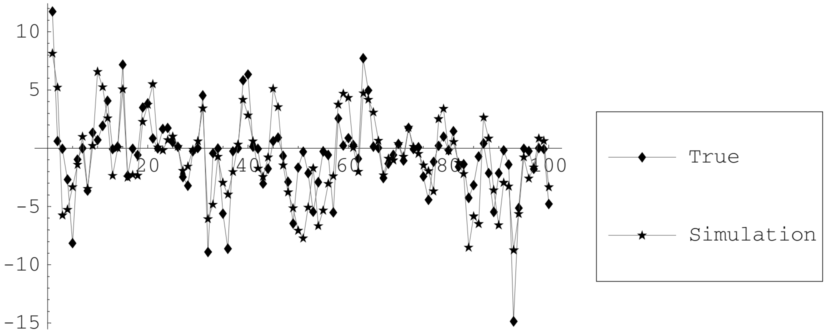
From the plot you see that the second output is described almost perfectly by the linear model, but there are inconsistencies in the first output. This is not surprising if you take a closer look at the true function generating the data. The second output can be described by a linear model but the first output cannot. To model the first output better, you need to make the neural network nonlinear by including some neurons. This will be done later in this section.

You can also use the command `NetSimulate` or `NetPredict` to perform simulations and predictions. They give you the simulated and predicted outputs in the same format as the original output signal. The obtained simulation and prediction can then, for example, be plotted with the true output.

Simulate the linear model and plot the first 100 values of the validation data together with the true output.

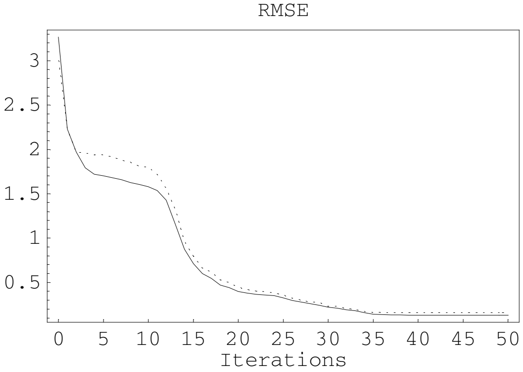
```
In[21]:= ys = NetSimulate[u, model1];
```

```
In[22]:= MultipleListPlot[y[[Ndata / 2 + Range[100], 1]],
          ys[[Ndata / 2 + Range[100], 1]], PlotJoined → True,
          PlotLegend → {"True", "Simulation"}, PlotStyle → {Hue[0.6], Hue[0.9]}]
```



Estimate a nonlinear model based on an FF network with 4 neurons and with the same regressor as the linear model.

```
In[23]:= {model2, fitrecord} =
           NeuralARXFit[ue, ye, {{2, 1}, {1, 1}, {1, 1}}, FeedForwardNet, {4}, uv, yv, 50];
```

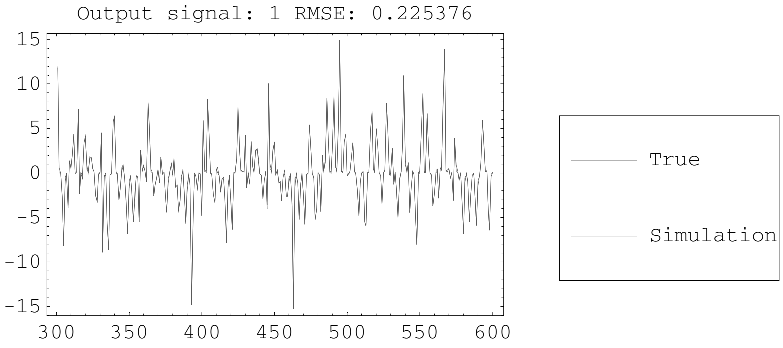


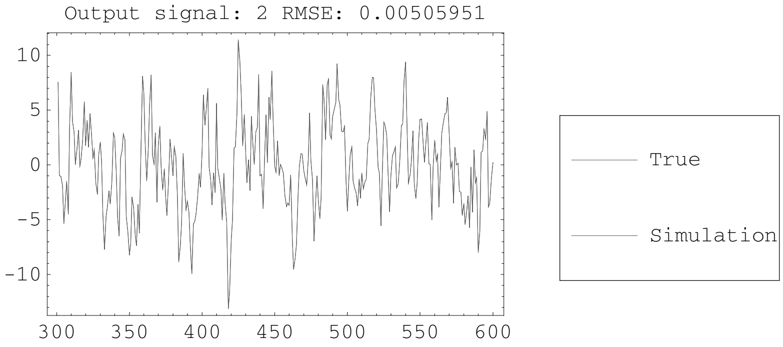
```
NeuralFit::StoppedSearch :
 The net parameters are obtained by stopped search using the supplied
   validation data. The neural net is given at the 39th training iteration.
```

Note that if you repeat the example, the result will turn out differently due to randomness in data and in the network initialization. You might need more training iterations, or you might get caught in a local minimum.

Compare the one-step-ahead prediction with the true output.

```
In[24]:= NetComparePlot[u, y, model2, ShowRange → {Ndata / 2 + 1, Ndata}, PredictHorizon → 1,
           PlotStyle → {Hue[.6], Hue[.8]}, PlotLegend → {"True", "Simulation"}]
```



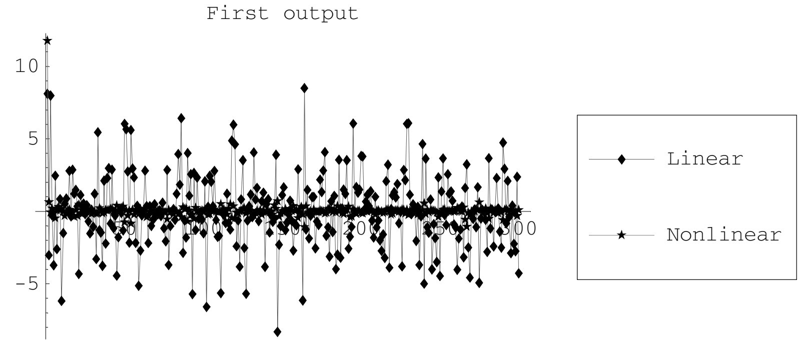


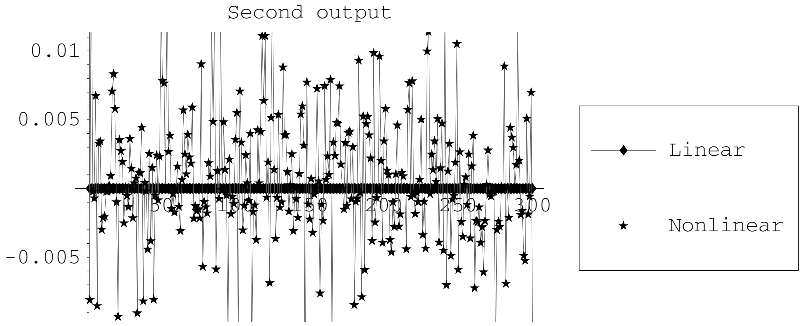
Compare this with the prediction of linear model. The first output is much better predicted, but the second is slightly worse.

The difference between the two models can be illustrated by comparing the prediction errors in common plots.

Compute and plot the prediction errors using the linear and nonlinear models.

```
In[25]:=  e1 = NetPredictionError[uv, yv, model1];
          e2 = NetPredictionError[uv, yv, model2];

In[27]:=  MultipleListPlot[e1[[All, 1]], e2[[All, 1]],
           PlotRange → All, PlotJoined → True, PlotStyle → {Hue[0.6], Hue[0.9]},
           PlotLabel → "First output", PlotLegend → {"Linear", "Nonlinear"}]
          MultipleListPlot[e1[[All, 2]], e2[[All, 2]],
           PlotJoined → True, PlotStyle → {Hue[0.6], Hue[0.9]},
           PlotLabel → "Second output", PlotLegend → {"Linear", "Nonlinear"}]
```





As you see from the plots, the errors of the linear model dominate the first output, but for the second output the nonlinear model gives the largest error. The scales of the prediction errors of the two ouputs are very different, however.

An analytic expression of dynamic neural network models is obtained by evaluating the neural network placed at the first position of the neural ARX model on a vector with length equal the number of regressors.

Express the linear ARX model analytically.

```
In[29]:= model1[[1]][{y1[t - 1], y1[t - 1], y2[t - 2], u1[t - 1], u2[t - 1]}][[1]]
```

```
Out[29]= 0.0272013 + 0.621573 u1[-1 + t] +
          0.529971 u2[-1 + t] - 0.0571367 y1[-1 + t] + 0.543105 y2[-2 + t]
```

The expression of the nonlinear model is much more complicated.

Express the nonlinear neural ARX model analytically.

```
In[30]:= model2[[1]][{y1[t - 1], y1[t - 1], y2[t - 1], u1[t - 1], u2[t - 1]}][[1]]
```

Out[30]=

$$
\begin{aligned}
&252.266 - 0.19221 / \left(1 + e^{-6.74106-0.122178\,\mathrm{u1}[-1+t]-0.0958901\,\mathrm{u2}[-1+t]-0.0656796\,\mathrm{y1}[-1+t]-0.239055\,\mathrm{y2}[-1+t]}\right) - \\
&694.073 / \left(1 + e^{0.561709+4.5562\times10^{-7}\,\mathrm{u1}[-1+t]+0.000317567\,\mathrm{u2}[-1+t]-1.15007\times10^{-7}\,\mathrm{y1}[-1+t]+0.000222354\,\mathrm{y2}[-1+t]}\right) + \\
&380694. / \left(1 + e^{-0.0183979-0.00648828\,\mathrm{u1}[-1+t]-0.00203839\,\mathrm{u2}[-1+t]+1.64912\,\mathrm{y1}[-1+t]+0.00916544\,\mathrm{y2}[-1+t]}\right) - \\
&380694. / \left(1 + e^{-0.018398-0.00647792\,\mathrm{u1}[-1+t]-0.00202864\,\mathrm{u2}[-1+t]+1.64913\,\mathrm{y1}[-1+t]+0.00917335\,\mathrm{y2}[-1+t]}\right)
\end{aligned}
$$

The symbolic expressions may be useful if you want to use general *Mathematica* commands to manipulate the neural network expression.

### 8.2.2 Identifying the Dynamics of a DC Motor

In this example you will see how the *Neural Networks* package can be used to model the dynamics of a DC motor. The input signal is the applied voltage and the output signal is the angular speed of the motor.

Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << dcmotor.dat;
```

The input data is placed in *u* and the output data in *y*.

Check the dimensions.

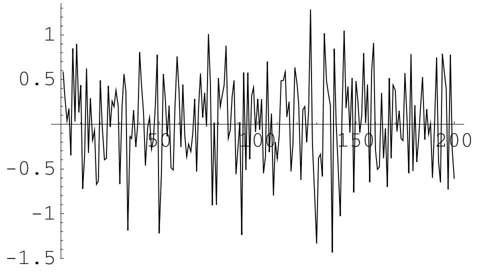
```
In[3]:=  Dimensions[u]
         Dimensions[y]

Out[3]=  {200, 1}

Out[4]=  {200, 1}
```

There are 200 data samples available, and the plant has one input and one output.

It is always a good idea to visually inspect the data before the modeling. This might allow you to detect any outliers in the data.

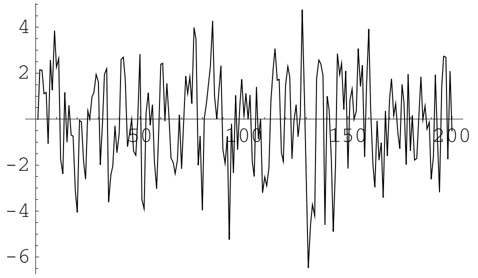
Show the input signal.

```
In[5]:=  ListPlot[Flatten[u], PlotJoined → True]
```



Show the output signal.

```
In[6]:=  ListPlot[Flatten[y], PlotJoined → True]
```



By inspecting the plots you might find outliers, which should be removed before the system identification procedure starts.

The first half of the data set is used for identification and the second half for validation of the model.

Divide the data into identification and validation data.

```
In[7]:= ue = u[[Range[100]]];
        ye = y[[Range[100]]];
        uv = u[[Range[101, 200]]];
        yv = y[[Range[101, 200]]];
```

It is a good idea to try a linear model first and then try to obtain a better nonlinear model. Using Maxwell's and Newton's laws, the following linear relationship for the DC motor could be expected.

$$\hat{y}(t) = ay(t-1) + bu(t-1) + c \qquad (1)$$

This means that the regressor should be chosen to $x(t) = \{y(t-1)\, u(t-1)\}$, which is obtained by choosing $n_a = 1$, $n_b = 1$, and $n_k = 1$.

The linear model has three parameters, $a$, $b$, and, $c$. You specify and train this linear model structure with the following call.

```
In[11]:= {model1, fitrecord} =
           NeuralARXFit[ue, ye, {1, 1, 1}, FeedForwardNet, {}, 0, CriterionPlot → False];
```

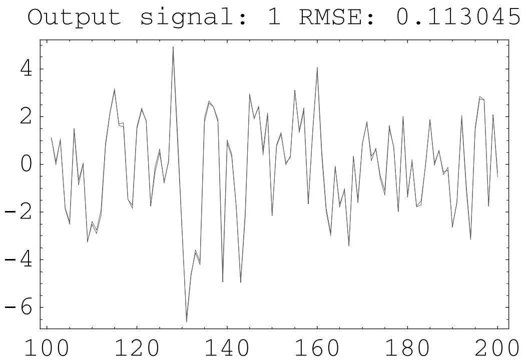
Find some information about the model.

```
In[12]:= NetInformation[model1]

Out[12]= NeuralARX model with 1 input signal and 1 output signal. The regressor
           is defined by: na = 1, nb = 1, nk = 1. The mapping from regressor to
           output is defined by a FeedForward network created 2002-4-3 at 13:
           56. The network has 2 inputs and 1 output. It has no hidden layer.
```

The command `NetComparePlot` is very convenient for evaluating dynamic models. Depending on which option you choose, the model can be a simulator or a predictor. The model output is displayed together with the true output signal. This type of test is only fair if you use fresh data, that is, the validation data for the comparison.

Simulate the linear model and compare it with the true output signal.

```
In[13]:= NetComparePlot[u, y, model1, PredictHorizon → Infinity,
          ShowRange → {101, 200}, PlotStyle → {Hue[.6], Hue[.8]}]
```



By including the whole data set in the call, and then indicating the validation data with the option `ShowRange`, you avoid transients in the beginning of the simulation.

You can also use the command `NetSimulate` or `NetPredict` directly instead of calling `NetComparePlot`.

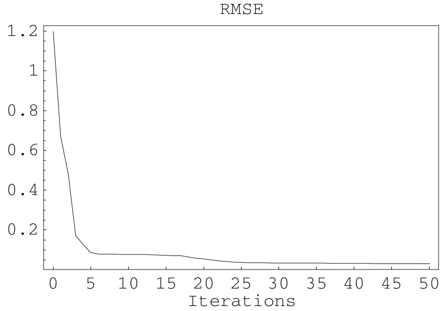
It is hard to see any difference between the true and the simulated output signal in the plot, and, obviously, the linear model is quite good at explaining the relationship in the data. A nonlinear model will now be trained to see if it becomes better than the linear model. The dynamic model can be described by

$$y(t) = g(\theta, y(t-1), u(t-1)) \tag{2}$$

where *g(θ, ·, ·)* is the neural network function whose parameters *θ* are to be trained.

Train an FF network on the DC-motor data.

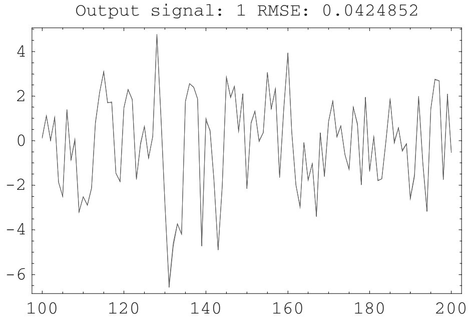
```
In[14]:= {model2, fitrecord} = NeuralARXFit[ue, ye, {1, 1, 1}, FeedForwardNet, {3}, 50];
```



Depending on the initialization you may end up in any one of many local minima. Notice that the result changes if you re-evaluate the training command due to the randomness in the initialization of the neural network. If the criterion at the end of the learning is larger than 0.05, you should repeat the command.

Simulate the nonlinear neural network model.

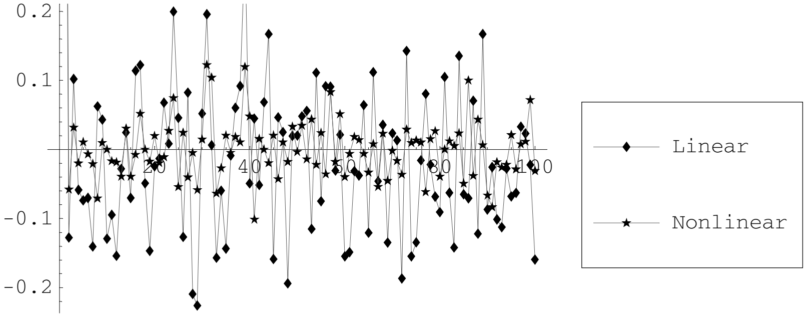
```
In[15]:= NetComparePlot[u, y, model2, PredictHorizon → Infinity,
          ShowRange → {100, 200}, PlotStyle → {Hue[.6], Hue[.8]}]
```



The nonlinear neural network model should give an RMSE of less than half of what you obtained for the linear model. However, because the models are so good it is hard to see any difference in the plots. Instead you can look at the prediction errors, that is, the difference between the true output and the model output. To do that you can use the command NetPredictionError.

Compute and plot the prediction errors using the linear and nonlinear models.

```
In[16]:= e1 = NetPredictionError[uv, yv, model1];
         e2 = NetPredictionError[uv, yv, model2];
         << Graphics`MultipleListPlot`;
         MultipleListPlot[Flatten[e1], Flatten[e2], PlotJoined → True,
          PlotStyle → {Hue[0.6], Hue[0.9]}, PlotLegend → {"Linear", "Nonlinear"}]
```

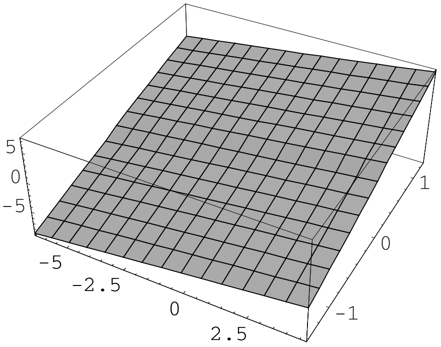


It should be evident from the plot that the prediction errors of the nonlinear model are much smaller than those of the linear model.

Since the model describing the DC motor has only two regressor components, it is possible to look at $g(\theta, x)$. But because the linear and nonlinear models are very similar, it is hard to see anything other than a linear relationship.

Plot the nonlinear model of the DC motor.

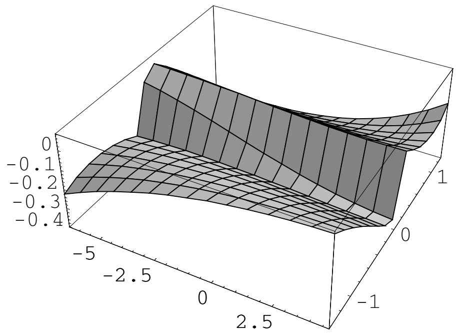
```
In[20]:= NetPlot[model2, uv, yv, DataFormat → FunctionPlot]
```



It is, however, fairly easy to plot the *difference* between the linear and nonlinear models. This can be done by extracting the linear parameters from the linear model and inserting them with opposite signs as a linear submodel in the nonlinear model. Section 13.1, Change the Parameter Values of an Existing Network, gives more details on how you can change the parameters in an existing neural network.

Plot the difference between linear and nonlinear models.

```
In[21]:= model3 = model2;
         model3[[1, 1]] = {model2[[1, 1, 1]], -model1[[1, 1, 1, 1, {1, 2}]]};
         NetPlot[model3, uv, yv, DataFormat → FunctionPlot]
```



Now it is easy to see that the relationship is far from being linear.

An analytic expression of the dynamic model can be obtained by using the neural net placed at the first position.

Express the neural ARX model analytically.

```
In[24]:= model2[[1]][{yy[t - 1], uu[t - 1]}][[1]]
```

$$\text{Out[24]}= 24.9618+\frac{20174.5}{1+e^{10.1785-0.647103\,uu[-1+t]-0.0877935\,yy[-1+t]}}-\frac{2301.18}{1+e^{4.46849+0.118863\,uu[-1+t]+0.0148835\,yy[-1+t]}}+\frac{0.385491}{1+e^{-3.22387-55.0474\,uu[-1+t]+0.313509\,yy[-1+t]}}$$

Similarly, it might be interesting to have the equation of the plane describing the first linear ARX model.

Describe the first linear ARX model.

```
In[25]:= model1[[1]][{yy[t - 1], uu[t - 1]}][[1]]

Out[25]= -0.137853 + 3.91426 uu[-1 + t] + 0.459061 yy[-1 + t]
```

### 8.2.3 Identifying the Dynamics of a Hydraulic Actuator

In this example the dynamics of a hydraulic actuator will be modeled. The data used was kindly provided by P. Krus at the Fluid Power Technology group of the Department of Mechanical Engineering, Linköping University, http://hydra.ikp.liu.se.

The input signal is the opening of a valve, which influences the flow of oil into a cylinder acting on a robot arm. The oil pressure is the output signal that relates to the position of the robot arm. The goal is to find a mathematical model describing how the valve opening influences the oil pressure.

Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`

In[2]:= << actuator.dat;
```

The input data is placed in *u* and the output data in *y*.

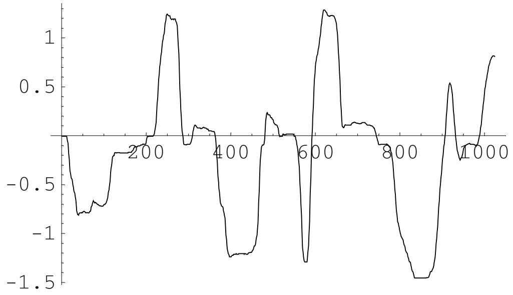
Check the dimensions.

```
In[3]:= Dimensions[u]
        Dimensions[y]

Out[3]= {1024, 1}

Out[4]= {1024, 1}
```

The 1024 input and output data samples involved are plotted.

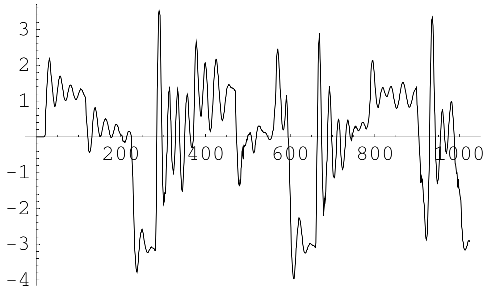
Plot the input signal.

```
In[5]:= ListPlot[Flatten[u], PlotJoined → True]
```



Plot the output signal.

```
In[6]:= ListPlot[Flatten[y], PlotJoined → True]
```



As in the previous example, the first half of the data set is used for identification and the second half for validation of the model.

Divide the data set into estimation and validation data.

```
In[7]:= ue = u[[Range[512]]];
        ye = y[[Range[512]]];
        uv = u[[Range[513, 1024]]];
        yv = y[[Range[513, 1024]]];
```

The first candidate model is a linear model so that you have something to compare the nonlinear model to. The chosen regressor indices are $n_a = 3$, $n_b = 2$, and $n_k = 1$, which give a regressor $x(t) = \{y(t-1),\ y(t-2),\ y(t-3),\ u(t-1),\ u(t-2)\}$. The linear model then becomes

$$y(t) = \theta^T x(t) + \theta_0 \tag{3}$$

where $\theta$ is a parameter vector of length 5 and $\theta_0$ is a level parameter.

A linear model is obtained by using a `FeedForwardNet` without any hidden layer. Since the parameter estimate of linear models can be computed exactly with only one training iteration, no iterative training is necessary.

Estimate a linear model of the hydraulic actuator.

```
In[11]:= {model1, fitrecord} =
           NeuralARXFit[ue, ye, {3, 2, 1}, FeedForwardNet, {}, 0, CriterionPlot → False];
```

Provide information about the model.

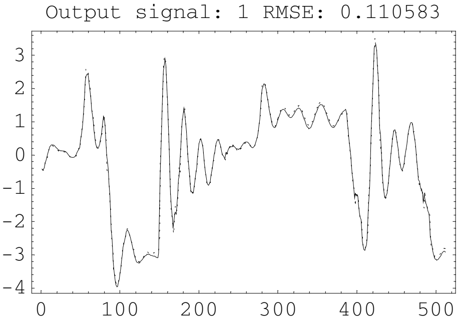
```
In[12]:= NetInformation[model1]

Out[12]= NeuralARX model with 1 input signal and 1 output signal. The regressor
           is defined by: na = 3, nb = 2, nk = 1. The mapping from regressor to
           output is defined by a FeedForward network created 2002-4-3 at 13:
           57. The network has 5 inputs and 1 output. It has no hidden layer.
```

Use `NetComparePlot` to produce a prediction and compare the result to the true output. Since the first half of the data was used for training, the second half is used for a fair validation.

A short prediction horizon will often yield good prediction performance if the signals are dominated by low frequencies. Therefore, it might be hard to evaluate the model's quality from a one-step prediction, which is shown here.

Compare a one-step prediction with the true output.

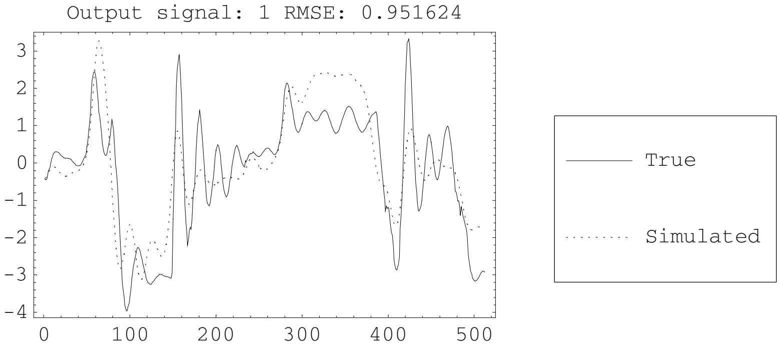
```
In[13]:= NetComparePlot[uv, yv, model1, PredictHorizon → 1]
```



The one-step prediction looks good, and it is hard to see the difference between the predicted and the true output. Often it is more interesting to look at the result using a larger prediction horizon or a pure simulation.

Compare a simulation with the true output.

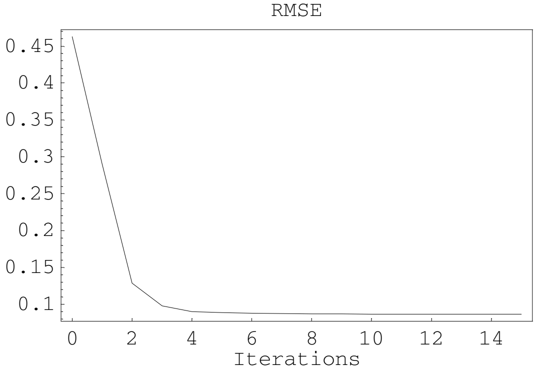
```
In[14]:= NetComparePlot[uv, yv, model1,
          PredictHorizon → Infinity, PlotLegend → {"True", "Simulated"}]
```



After having obtained a linear model, you can try to derive a better nonlinear one. This can be done by using the same regressor as earlier, but adding a hidden layer to the network.

Train an FF network with four neurons on the hydraulic actuator.

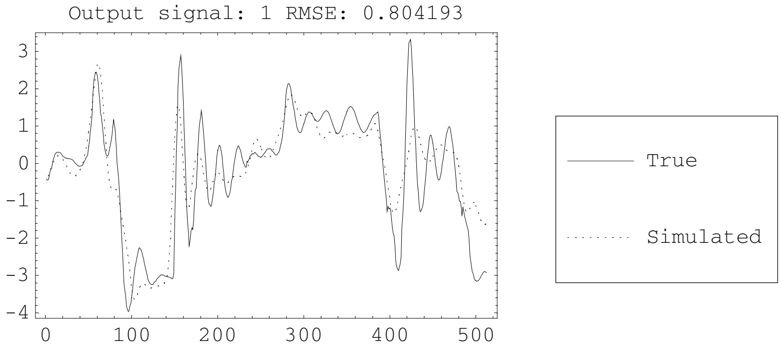
```
In[15]:= {model2, fitrecord} = NeuralARXFit[ue, ye, {3, 2, 1}, FeedForwardNet, {4}, 15];
```



Depending on the initialization, the training ends up in different local minima. If the result is not satisfactory, you can repeat the training; a new initialization is used each time.

Simulate the nonlinear model and compare the result with the true output.

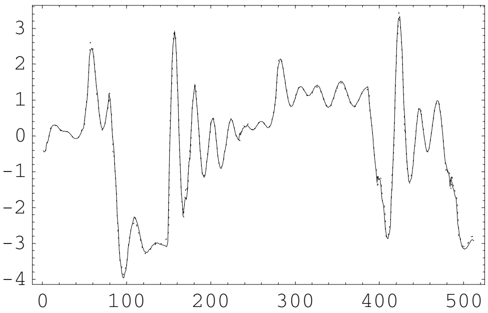
```
In[16]:= NetComparePlot[uv, yv, model2,
           PredictHorizon → Infinity, PlotLegend → {"True", "Simulated"}]
```



Evaluate the model on validation data using a one-step prediction.

```
In[17]:= NetComparePlot[uv, yv, model2, PredictHorizon → 1]
```

Output signal: 1 RMSE: 0.10438

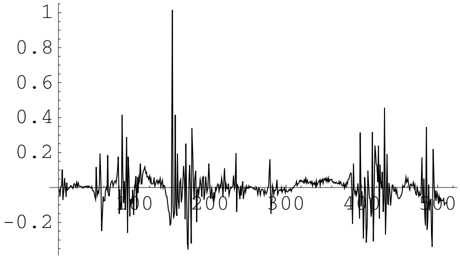


Is the performance better than the linear model?

It might be interesting to look at the prediction errors to see where on the data sequence the model performs well.

Compute and plot the prediction errors.

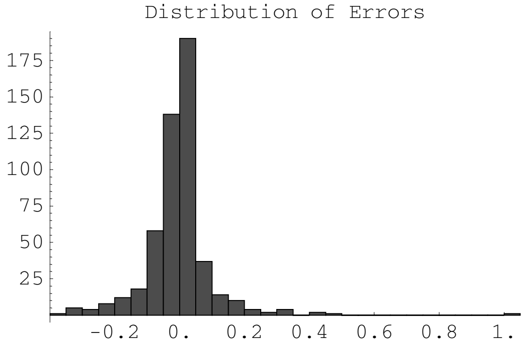
```
In[18]:= ep = NetPredictionError[uv, yv, model2, PredictHorizon → 1];
         ListPlot[Flatten[ep], PlotJoined → True, PlotRange → All]
```



You can also look at the distribution of the prediction errors. Such a plot can indicate if the model has problems explaining some individual data samples.

Display the distribution of the prediction errors with a histogram.

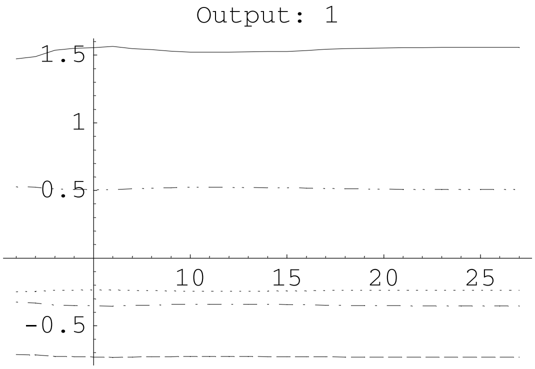
```
In[20]:= NetPlot[model2, uv, yv, DataFormat → ErrorDistribution]
```



You can plot the linearization at each sampling instant. The smaller the variations in the parameter values of the linear model over the data domain, the closer the underlying model is to a linear model. A curve that is close to zero over the whole data domain indicates that a smaller model with fewer regressors could be better (assuming the same range of all signals).

Display the linearization of the nonlinear model versus the data samples.

```
In[21]:= NetPlot[model2, uv[[Range[30]]], yv[[Range[30]]], DataFormat → LinearParameters]
```



### 8.2.4 Bias-Variance Tradeoff—Avoiding Overfitting

As described in Section 7.5, Regularization and Stopped Search, it is critical to find the appropriate type of model and the appropriate number of parameters of the model.

In this example three ways to avoid overfitting are demonstrated: choosing a network with a sufficiently low number of neurons, using stopped search for the minimum, and applying regularization.

The data from the hydraulic actuator from the previous example is used to demonstrate these alternative options. You may refer to the previous example if you want an introduction to this data set.

Load the *Neural Networks* package and the data.

```
In[1]:=  << NeuralNetworks`
```

```
In[2]:=  << actuator.dat;
```

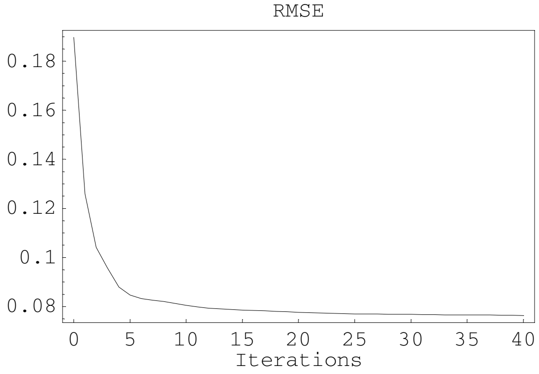
The first half of the data set is used for identification and the second half for validation of the model.

```
In[3]:=  ue = u[[Range[512]]];
         ye = y[[Range[512]]];
         uv = u[[Range[513, 1024]]];
         yv = y[[Range[513, 1024]]];
```

Train a nonlinear neural ARX model with many neurons on the data.

Train an FF network on the hydraulic actuator.

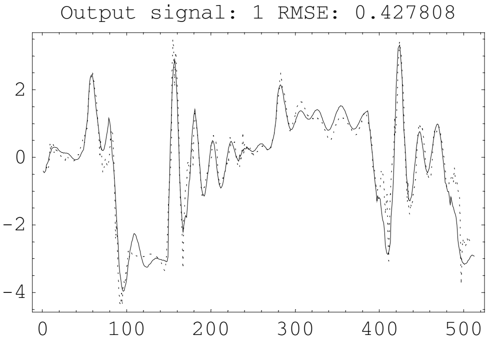
```
In[7]:=  {model1, fitrecord} = NeuralARXFit[ue, ye, {3, 2, 1}, FeedForwardNet, {8}, 40];
```



Depending on the initialization you end up in different local minima. If the result is not satisfactory you may repeat the training, which will use a new initialization.

Evaluate the model on validation data using a four-step prediction.

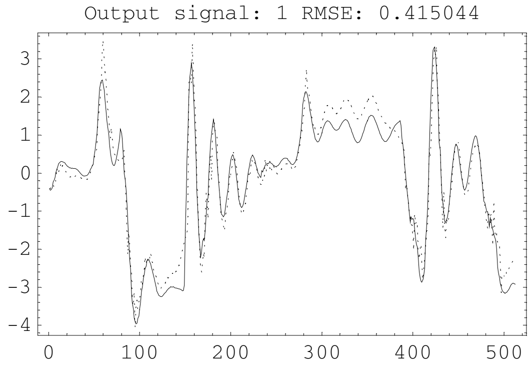
```
In[8]:= NetComparePlot[uv, yv, model1, PredictHorizon → 4]
```



The result of the prediction depends on which minimum the training converged to, but usually the result is worse than that of a linear model. You may try a linear model for comparison.

Estimate a linear model and display the four-step prediction.

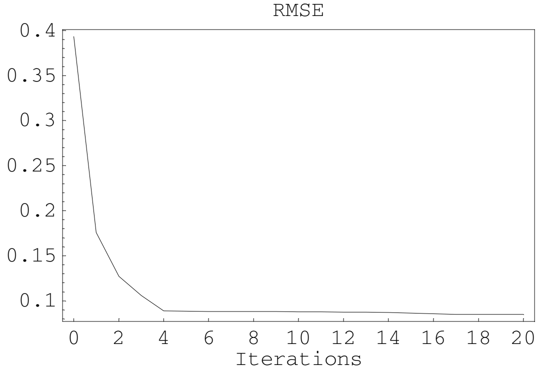
```
In[9]:= {model2, fitrecord} = NeuralARXFit[ue, ye, {3, 2, 1},
           FeedForwardNet, {}, 0, CriterionPlot → False, CriterionLog → False];
        NetComparePlot[uv, yv, model2, PredictHorizon → 4]
```



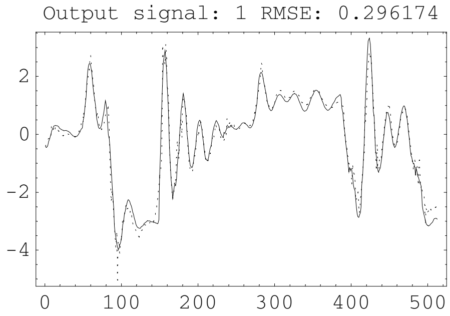
The reason why the nonlinear model was worse than the linear model is that it had more degrees of freedom than necessary; that is, it used more parameters than necessary. In contrast, the linear model used fewer parameters than necessary. By choosing somewhere between 0 and 8 hidden neurons, it might be possible to find a better model. This is the first way to handle the bias-variance tradeoff.

Estimate and predict a model with four hidden neurons.

```
In[11]:= {model3, fitrecord} = NeuralARXFit[ue, ye, {3, 2, 1}, FeedForwardNet, {4}, 20];
```



```
In[12]:= NetComparePlot[uv, yv, model3, PredictHorizon → 4]
```

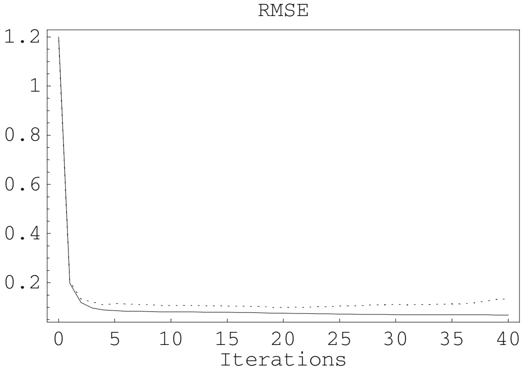


Is the result better than the linear model and the model with eight neurons? Try other numbers of neurons and repeat the training.

You can also submit the validation data to the training algorithm. The criterion is then evaluated after each iteration. As described in Section 7.5, Regularization and Stopped Search, the most important parameters are adapted in the beginning of the training and the least important at the end. The performance of the network model during the training is illustrated by the plot of the criterion evaluated on the validation data, which is shown at the end of the training. If the model starts to become worse after some number of training iterations then the model is *overtrained*.

Train a large network but supply the validation data.

```
In[13]:= {model4, fitrecord} = NeuralARXFit[ue, ye, {3, 2, 1}, FeedForwardNet,
            {8}, uv, yv, 40, RandomInitialization → LinearParameters];
```



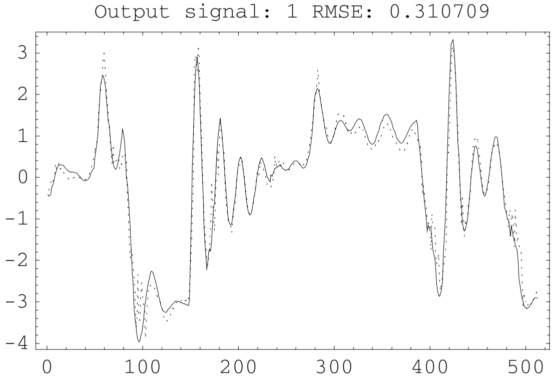
```
NeuralFit::StoppedSearch :
 The net parameters are obtained by stopped search using the supplied
   validation data. The neural net is given at the 20th training iteration.
```

Is there any overtraining? Usually there is when you use such a large network, but, depending on initialization, overtraining may not be an issue.

If you submit validation data, the desired model is *not* necessarily the one obtained at the end of the training process. The desired model is the one that gives the best match to the validation data, which could exist at an intermediate iteration in the training process. Therefore, by supplying validation data you can do the bias-variance tradeoff by stopped search; that is, stopping the training at the iteration where the prediction best matches the validation data.

Predict with the model obtained by stopped search.

```
In[14]:= NetComparePlot[uv, yv, model4, PredictHorizon → 4]
```



When validation data is submitted in the training, you automatically obtain a stopped search model. If you want the model at the last iteration instead, it is possible to get it by using the training record. It contains a list of the parameters after each training iteration.

Check the storing format of the training record.

```
In[15]:= fitrecord

Out[15]= NeuralFitRecord[NeuralARX, ReportFrequency → 1, CriterionValues → {-values-},
          CriterionValidationValues → {-values-}, ParameterRecord → {-parameters-}]
```

Note that the value of ReportFrequency indicates the iteration frequency with which parameter values are logged.

Extract the parameters versus training iterations and check the length of the parameter list.

```
In[16]:= parameters = ParameterRecord /. fitrecord[[2]];
         Length[parameters]

Out[17]= 41
```

The length of the parameter log equals the number of iterations, including one for the initial estimate. The model at the last iteration can now be compared to the one obtained with the preceding stopped search technique. This can be done in the following way. First check how the information is stored in the model.

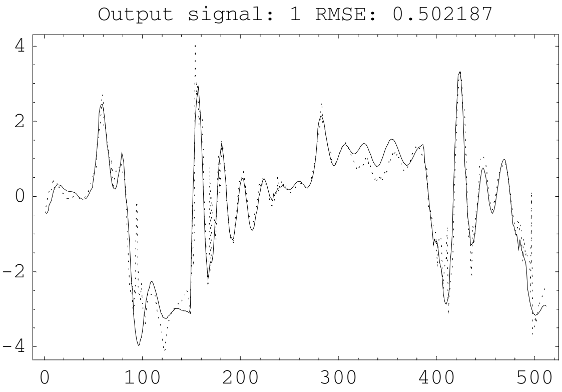
```
In[18]:=  model4

Out[18]=  NeuralARX[FeedForwardNet[{{w1, w2}},
             {AccumulatedIterations → 40, CreationDate → {2003, 7, 28, 21, 22, 0.7205328},
              Neuron → Sigmoid, FixedParameters → None,
              OutputNonlinearity → None, NumberOfInputs → 5}], Regressor → {3, 2, 1}]
```

Create a new model with the same structure as the previous one and insert the last parameter values from the training record.

```
In[19]:=  model5 = model4;
          model5[[1, 1]] = Last[parameters];
```

Predict using the final model.

```
In[21]:=  NetComparePlot[uv, yv, model5, PredictHorizon → 4]
```



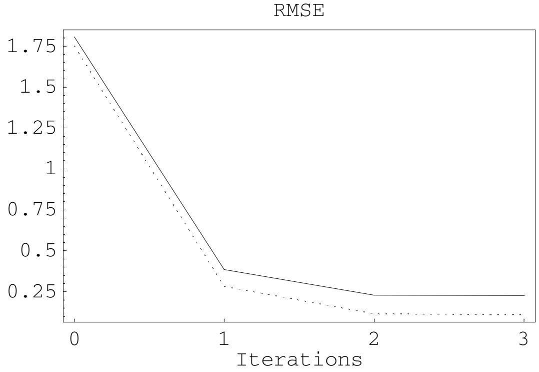
Now, compare the performance of this model with that of the stopped search model.

If you do not want to use the stopped search feature, it might be interesting to submit validation data in the training. If the performance measured on validation data increases toward the end of the training, then this indicates that the chosen model has too much flexibility. You should choose a neural network with fewer neurons (or use stopped search).

The third way to handle the bias-variance tradeoff is to use a large model but to minimize a regularized criterion, as described in Section 7.5, Regularization and Stopped Search.

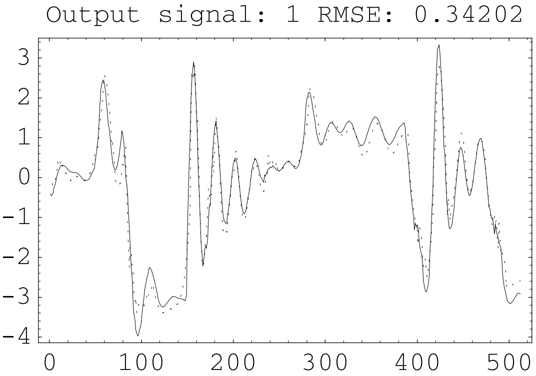
Train a neural network using regularization.

```
In[22]:= {model6, fitrecord} = NeuralARXFit[ue, ye, {3, 2, 1}, FeedForwardNet, {8}, uv, yv,
           15, RandomInitialization → LinearParameters, Regularization → 0.0001];
```



Evaluate the regularized network model.

```
In[23]:= NetComparePlot[uv, yv, model6, PredictHorizon → 4]
```



How does the regularized model perform compared to the other two ways of handling the bias-variance tradeoff?

In this example you have seen three ways to handle the bias-variance tradeoff: (1) use a sparse model, which does not have more parameters than necessary; (2) use a large model in connection with the validation data to apply stopped search, so that only a subset of the parameters are used; and (3) use a large model with regularization so that only a subset of the parameters are used.

### 8.2.5 Fix Some Parameters—More Advanced Model Structures

Sometimes it can be interesting to exclude some of the parameters from the training process. In this way, it is possible to obtain special model structures, where some features are built into the model at initialization. Addressed in the following is a special model for the hydraulic actuator problem discussed earlier. The option `FixedParameters` is used to exclude parameters from the fit, and it is explained in Section 13.2, Fixed Parameters.

There are reasons to believe that most of the nonlinear behavior of the hydraulic actuator is near the physical input of the system, where the oil streams into the cylinder. Hence, it could be interesting to include only the past input values of the regressor in the nonlinear part of the model, while keeping the model linear in the past output values. Using a regressor similar to that used in the previous two examples yields the following model:

$$\begin{aligned}&\hat{y}(t) = \\ &\quad a_1\, y(t-1) + a_2\, y(t-2) + a_3\, y(t-3) + g(\theta, u(t-1), u(t-2))\end{aligned} \qquad (4)$$

where $g(\theta, u(t-1), u(t-2))$ is a neural network.

Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << actuator.dat;
```

Divide the data into training and validation data.

```
In[3]:= ue = u[[Range[512]]];
        ye = y[[Range[512]]];
        uv = u[[Range[513, 1024]]];
        yv = y[[Range[513, 1024]]];
```

Before fixing the parameters, an initial model is obtained by invoking training with zero iterations. Note that you can repeat the example with different models by modifying the structure indices $n_a$, $n_b$, $n_k$, and $n_h$. Choose an FF network with two hidden layers.

Obtain an initial FF network model with two hidden layers.

```
In[7]:=  na = 3; nb = 2; nk = 1; nh = {3, 3};
         {model1, fitrecord} = NeuralARXFit[ue, ye, {na, nb, nk},
            FeedForwardNet, nh, 0, LinearPart → True, CriterionPlot → False];
```

Inspect the format of the model.

```
In[9]:=  model1

Out[9]=  NeuralARX[FeedForwardNet[{{w1, w2, w3}, χ},
           {AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 7, 32},
            Neuron → Sigmoid, FixedParameters → None,
            OutputNonlinearity → None, NumberOfInputs → 5}], Regressor → {3, 2, 1}]
```

It is the first $n_a$ rows of $w1$, corresponding to the past $y$ values, that have to be set to zero. Refer to the description in Section 2.6, Dynamic Neural Networks.

Take a look at the $w1$ matrix.

```
In[10]:=  w1 = model1[[1, 1, 1, 1]] ;
          MatrixForm[w1]

  Out[11]//MatrixForm=
         ⎛ -0.32458    -1.52618    0.322549 ⎞
         ⎜ -0.685248  -0.0795772   1.15006  ⎟
         ⎜ -0.628164   2.74448     0.129114 ⎟
         ⎜ 0.0613149  -3.40022     0.729567 ⎟
         ⎜  -0.1638   -0.0633633  -0.871552 ⎟
         ⎝  2.60667    4.00491     0.521309 ⎠
```

Set the first $n_a$ rows to zero.

```
In[12]:=  model1[[1, 1, 1, 1, {1, 2, 3}]] = Table[0, {na}, {nh[[1]]}];
```

Check that the manipulation is correct.

```
In[13]:= w1 = model1[[1, 1, 1, 1]] ;
         MatrixForm[w1]
```

```
Out[14]//MatrixForm=
 ⎛    0           0           0      ⎞
 ⎜    0           0           0      ⎟
 ⎜    0           0           0      ⎟
 ⎜0.0613149   -3.40022    0.729567   ⎟
 ⎜ -0.1638   -0.0633633 -0.871552    ⎟
 ⎝ 2.60667    4.00491    0.521309    ⎠
```

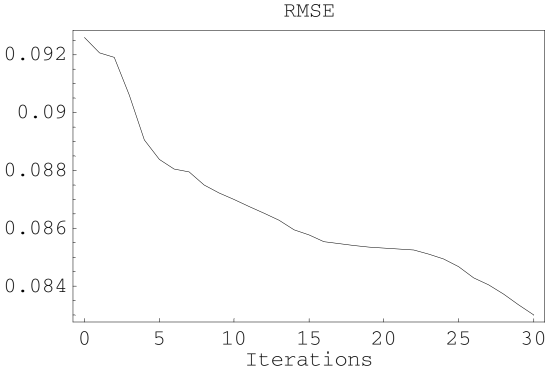
The parameters that have been set to zero also have to remain zero during the training. Therefore, the zeroed parameters have to be held fixed in the training. To do that you need to know their positions in the flattened parameter list of the model. You find the position easily by a search of the zeros.

Find the indices of the parameters to be held fixed.

```
In[15]:= fixparameters = Flatten[Position[Flatten[model1[[1, 1]]], 0]]

Out[15]= {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Train the network holding the zeroed parameters fixed.

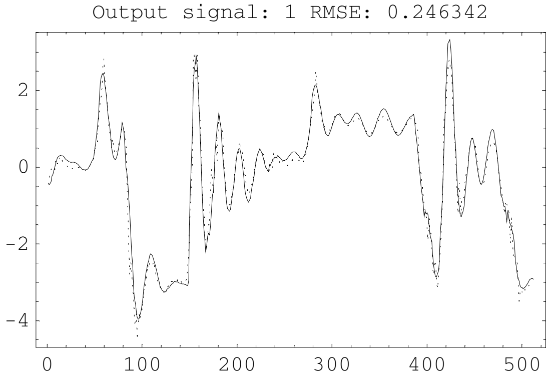
```
In[16]:= {model2, fitrecord} =
           NeuralARXFit[ue, ye, model1, 30, FixedParameters → fixparameters];
```



The trained model can now be tested on the validation data.

Compare four-step prediction with the true output on the validation data.

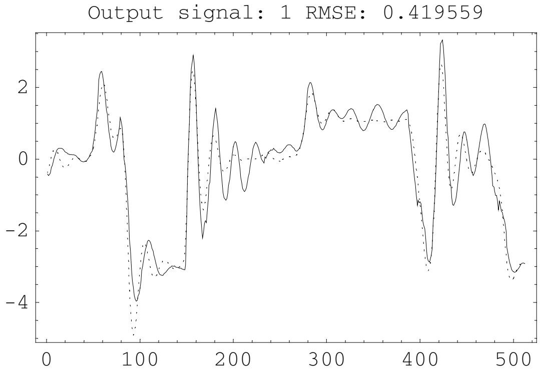
```
In[17]:= NetComparePlot[uv, yv, model2, PredictHorizon → 4]
```



Compare the result with those of the techniques for handling the bias-variance tradeoff, which were demonstrated in the preceding example.

Compare the simulation with the true output on the validation data.

```
In[18]:= NetComparePlot[uv, yv, model2]
```



You can go back and repeat the example, changing the size of the model. Notice that the result will change also if you do not change the network model, due to the randomness in the initialization.

The parameter-fixing feature can be used to incorporate prior knowledge in the network model. In this way a model with fewer parameters can be obtained, which performs well from a bias-variance perspective.

## 8.3 Further Reading

System identification and time series prediction are broad and diverse fields. This list is a small sampling of the vast literature available on these topics.

The following books are good introductions:

R. Johansson, *System Modeling and Identification*, Englewood Cliffs, NJ, Prentice Hall, 1993.

L. Ljung and T. Glad, *Modeling of Dynamic Systems*, Englewood Cliffs, NJ, Prentice Hall, 1994.

The following books are more thorough and they are used in graduate courses at many universities:

L. Ljung, *System Identification: Theory for the User*, 2nd ed., Englewood Cliffs, NJ, Prentice Hall, 1999.

T. Söderström and P. Stoica, *System Identification*, Englewood Cliffs, NJ, Prentice Hall, 1989.

The following article discusses possibilities and problems using nonlinear identification methods from a user's perspective:

J. Sjöberg et al., "Non-Linear Black-Box Modeling in System Identification: A Unified Overview", *Automatica*, **31** (12), 1995, pp. 1691–1724.

This book is a standard reference:

G. E. P. Box and G. M. Jenkins, *Time Series Analysis, Forecasting and Control*, Oakland, CA, Holden-Day, 1976.

Many modern approaches to time series prediction can be found in this book and in the references therein:

A. S. Weigend and N. A. Gershenfeld, "Time Series Prediction: Forecasting the Future and Understanding the Past", in *Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis held in Santa Fe, New Mexico, May 14–17, 1992*, Reading, MA, Addison-Wesley, 1994.

Standard books on neural networks might also be of some interest. The following are examples:

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, Macmillan, 1999.

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Reading, MA, Addison-Wesley, 1991.

# 9 Hopfield Networks

Hopfield networks, or *associative networks,* are typically used for classification. Given a distorted input vector, the Hopfield network associates it with an undistorted pattern stored in the network.

A short tutorial about Hopfield networks is given in Section 2.7, Hopfield Network. Section 9.1, Hopfield Network Functions and Options, defines the commands and the options. Section 9.2, Examples, contains demonstration examples illustrating the commands.

## 9.1 Hopfield Network Functions and Options

This section introduces the commands you need to train and use Hopfield networks. Examples can be found in Section 9.2, Examples.

### 9.1.1 HopfieldFit

Hopfield networks are defined with the function `HopfieldFit`.

| | |
|---|---|
| `HopfieldFit[`*x*`,` *opts*`]` | uses the data vectors *x* to create a discrete- or continuous-time Hopfield network depending on the options |

Training a Hopfield network.

`HopfieldFit` returns an object with head `Hopfield`. The information of the network is stored in the systematic way used by all neural networks of the package, as described in Section 3.2, Package Conventions. The first component is the matrix that contains the parametric weights, as described in Section 2.7, Hopfield Network.

`HopfieldFit` takes the following options.

| *options* | *default values* | |
|---|---|---|
| NetType | Discrete | indicates the discrete- or continuous-time Hopfield model |
| Neuron | SaturatedLinear | type of neuron for continuous-time Hopfield models |
| WorkingPrecision | 4 | indicates the precision of the solution of a continuous-time Hopfield network |

Options of HopfieldFit.

The option NetType takes the value Discrete or Continuous, indicating which type of Hopfield network you want to create. Continuous-time Hopfield networks can have two types of neurons, SaturatedLinear or Tanh. You use the option Neuron to indicate your preference. The option WorkingPrecision indicates the precision, the number of decimals, with which the differential Equation 2.28 in Section 2.7, Hopfield Network, should be solved for continuous-time Hopfield networks.

A continuous-time Hopfield network stores a few more information items than its discrete-time counterpart. The type of nonlinear activation function, Neuron, WorkingPrecision, and the step size, Dt, for the differential equation must be logged.

The network can be evaluated for the disturbed data vectors using the evaluation rule for Hopfield objects. This means that the equation describing the network, as given in Section 2.7, Hopfield Network, is simulated using the disturbed data vector as an initial state.

| | |
|---|---|
| *net*[*x*] | evaluates *net* on the input vector *x* |

Function evaluation of a Hopfield network.

The input argument $x$ can be a vector containing one input sample or a matrix containing one input sample on each row.

The evaluation rule for Hopfield networks has the option Trajectories. By setting this option to True, you obtain not only the final values of the state vectors, but also the trajectories of the states starting at the initial values, supported in the call, and finishing at the final values.

| *option* | *default value* | |
|---|---|---|
| `Trajectories` | `False` | indicates if the state trajectories should be returned or only the final state values |

Option of the evaluation rule for Hopfield networks.

### 9.1.2 NetInformation

Some information about a Hopfield network is provided when the command `NetInformation` is applied.

| | |
|---|---|
| `NetInformation[`*hop*`]` | gives information about a Hopfield network |

The `NetInformation` function.

### 9.1.3 HopfieldEnergy

The energy of a Hopfield network, at any value of the state vector, is given by `HopfieldEnergy`. The mathematical definition of the energy is provided in Section 2.7, Hopfield Network.

| | |
|---|---|
| `HopfieldEnergy[`*hop*`,` *x*`]` | computes the energy level for the given Hopfield network *hop* at the indicated state vector *x* |

Computing the energy of a state of a Hopfield network.

`HopfieldEnergy` has no options.

### 9.1.4 NetPlot

A Hopfield network can be evaluated directly on a data vector by applying the evaluation rule as shown earlier. The command `NetPlot` can also be used, and it gives some more information about the evaluation. It works in the following way.

| | |
|---|---|
| `NetPlot[`*hop*`,` *x*`,` *opts*`]` | plots the convergence path of a Hopfield network; presents the result in various ways by choosing different options |

Illustrate Hopfield networks.

NetPlot takes the following options when it is applied to a Hopfield network.

| *option name* | *default value* | |
|---|---|---|
| DataFormat | Trajectories | indicates how the classification result is illustrated |
| Compiled | True | use compiled version |

Option of NetPlot.

NetPlot simulates the Hopfield network, as described in Section 2.7, Hopfield Network, using the supplied disturbed data vector $x$ as an initial state. By giving different values to the option DataFormat you can obtain the result, which may be presented in different ways.

The default DataFormat→Trajectories gives a plot of the components of the state vector as a function of time. In addition to DataFormat, NetPlot also passes on any other options to its plot commands so that you can modify the plots.

Possible values for DataFormat include the following.

| | |
|---|---|
| Trajectories | plots the components of the state vectors as a function of time |
| Energy | plots the energy decrease as a function of time |
| ParametricPlot | an option only possible for two-dimensional problems; gives a parametric plot of the states together with a contour plot of the energy |
| Surface | an option only possible for continuous-time Hopfield nets in two dimensions; gives a parametric plot of the states together with a 3 D plot of the energy |

Possible values of DataFormat .

As described in Section 2.7, Hopfield Network, the convergence points of a Hopfield network are always local energy minima. That is why the energy is strictly decreasing.

For a continuous-time, two-dimensional Hopfield network, you can also get the energy surface together with the state trajectories by choosing the option `Surface`.

## 9.2 Examples

In this subsection, Hopfield networks are used to solve some simple classification problems. The first two examples illustrate the use of discrete-time Hopfield models, and the last two examples illustrate the continuous-time version on the same data sets.

### 9.2.1 Discrete-Time Two-Dimensional Example

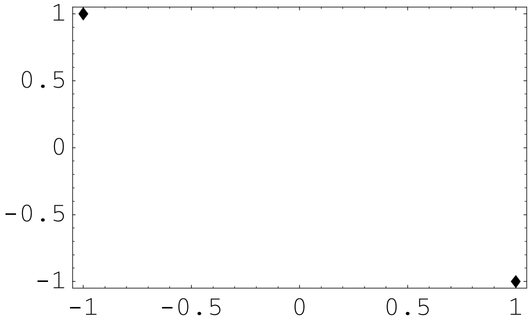
Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

In this small example there are two pattern vectors {1, −1} and {−1, 1}. Since the vectors are two-dimensional you can display the results to illustrate the outcome.

Generate and look at class vectors.

```
In[2]:= x={{1,-1},{-1,1}};
        NetClassificationPlot[x]
```



The two pattern vectors are placed in the corners of the plot. The idea is now that disturbed versions of the two pattern vectors should be classified to the correct undisturbed pattern vector.

Define a discrete-time Hopfield network.

```
In[4]:= hop = HopfieldFit[x]

Out[4]= Hopfield[W, {NetType → Discrete, CreationDate → {2002, 4, 3, 14, 9, 1}}]
```

Because the discrete Hopfield network is the default type, you do not have to specify that you want this type.

Some descriptive information is obtained by using `NetInformation`.

```
In[5]:= NetInformation[hop]

Out[5]= Discrete Hopfield model for pattern vectors with 2
          components. Created 2002-4-3 at 14:09. Type of neuron: Sign[x].
```

A new data pattern may be classified by processing it with the obtained model.

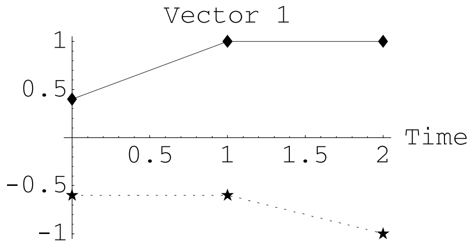
Evaluate the network for some disturbed data vectors.

```
In[6]:= hop[{0.4,-0.6}]

Out[6]= {{1, -1}}
```

More information about the evaluation of the Hopfield network on data vectors can be obtained by using `NetPlot`. The default is to plot the state trajectories as a function of time.

Plot the state vectors versus time.

```
In[7]:= NetPlot[hop, {{0.4, -0.6}}]
```



It might be interesting to obtain the trajectories for further manipulation. They can be obtained using the evaluation rule with the option `Trajectories→True`. Then the trajectories are returned instead of only the final value, which is the default.

Obtain the state trajectory.

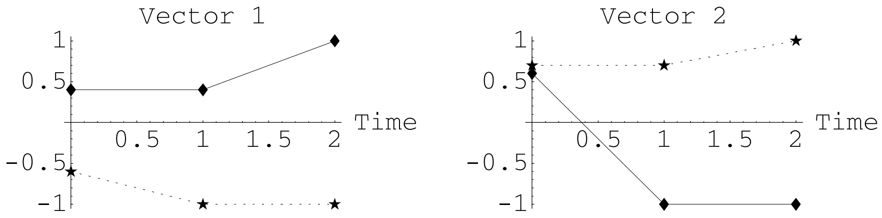
```
In[8]:= hop[{{0.4, -0.6}}, Trajectories → True]

Out[8]= {{{0.4, -0.6}, {1, -0.6}, {1, -1}}}
```

The trajectory is the numerical solution to Equation 2.26 describing the network; see Section 2.7, Hopfield Network.

`NetPlot` can also be used for several patterns simultaneously.

Evaluate two data vectors simultaneously.

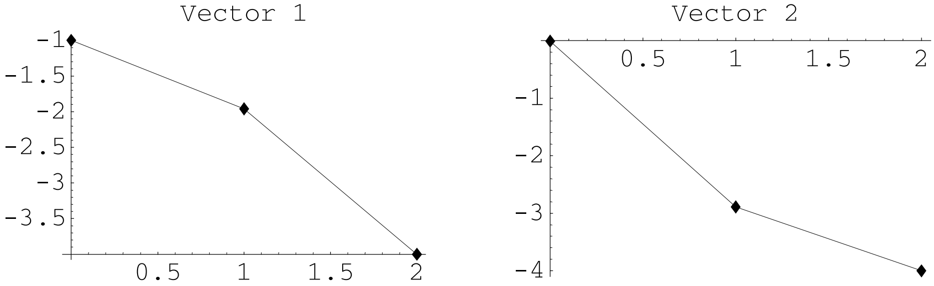
```
In[9]:= res = NetPlot[hop, {{0.4, -0.6}, {0.6, 0.7}}]
```



By giving the option `DataFormat→Energy`, you obtain the energy decrease from the initial point, the data vector, to the convergence point as a function of the time.

Look at the energy decrease.

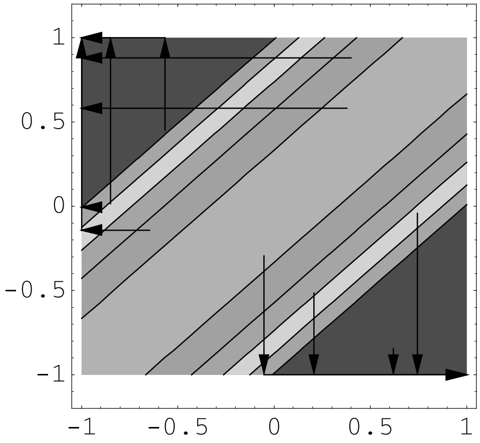
```
In[10]:= res = NetPlot[hop, {{0.4, -0.6}, {0.6, 0.7}},DataFormat→Energy]
```



Try 30 data pattern vectors at the same time. To avoid 30 trajectory and energy plots, you can instead choose the option `DataFormat→ParametricPlot`. You can use the command `RandomArray` from the standard add-on package `Statistics`ContinuousDistributions`` to generate random vectors.

Plot a contour plot with state vector trajectories.

```
In[11]:= << Statistics`ContinuousDistributions`
         x = RandomArray[UniformDistribution[-1, 1],  {10, 2}];
         NetPlot[hop,x,DataFormat→ParametricPlot,PlotRange→{-1.2,1.2}]
```



All trajectories converge to {−1, 1} or {1, −1}, which are the two pattern vectors used to define the Hopfield network with `HopfieldFit`.

### 9.2.2 Discrete-Time Classification of Letters

In this example, some letters will be automatically classified.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

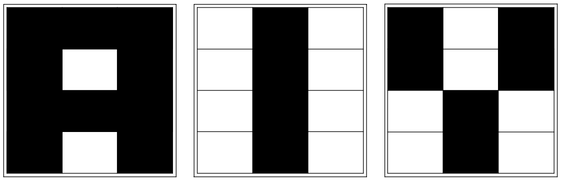
Generate patterns of the letters A, I, and Y. They are stored as matrices with 1 indicating black and −1 indicating white.

Generate the letters A, Y, and I in a list *x*.

```
In[2]:= x=-{{{1,-1,1},{1,1,1},{1,-1,1},{1,1,1}},
        {{-1,1,-1},{-1,1,-1},{-1,1,-1},{-1,1,-1}},
        {{-1,1,-1},{-1,1,-1},{1,-1,1},{1,-1,1}}};
```

Look at the letters.

```
In[3]:= xg=Map[ListDensityPlot[#,DisplayFunction→Identity,FrameTicks→None]&,x];
        Show[GraphicsArray[xg]]
```



Before you can construct a Hopfield network, the matrices have to be transformed into vectors.

Transform the matrices into pattern vectors.

```
In[5]:= xv = Map[Flatten,x,{1}];
```

The vectors containing the input pattern representing the three letters can now be used to create a Hopfield network.

Create a Hopfield network.

```
In[6]:= hopletter = HopfieldFit[xv]

Out[6]= Hopfield[W, {NetType → Discrete, CreationDate → {2002, 4, 3, 14, 10, 31}}]
```

The obtained Hopfield network can be tested in different ways. Start by determining if it can correctly classify noisy versions of the three letters. Noisy versions are generated when each pixel has a certain probability to change value. Since this is a random process you will receive different disturbed data vectors each time you repeat the following commands.

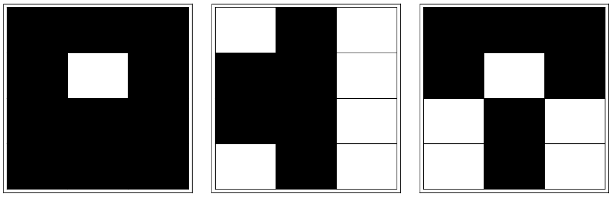
Create three disturbed data vectors from the three pattern vectors.

```
In[7]:= <<Statistics`ContinuousDistributions`
        y=Sign[xv*RandomArray[UniformDistribution[-0.1,1], {3,12}]];
```

You can look at the disturbed data vectors, but first they must be converted to matrices.

Look at the disturbed letters.

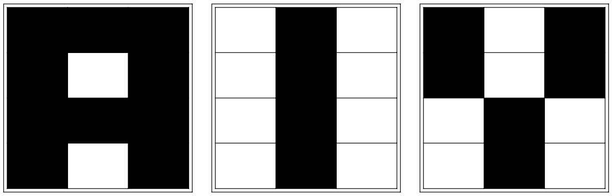
```
In[9]:= ym=Map[Partition[#,3] &, y];
        yg=Map[ListDensityPlot[#,DisplayFunction→Identity,FrameTicks→None] &,ym];
        Show[GraphicsArray[yg]]
```



It is now time to evaluate the Hopfield network against the disturbed data vectors. This is done by mapping the Hopfield object containing the Hopfield network on each of the data vectors describing the noisy letters. The result is converted back into matrices and plotted.

Evaluate the Hopfield network on the noisy letters and plot the result.

```
In[12]:= yh=Map[hopletter[#] &, y];
         yh=Apply[Partition[#,3] &, yh, 1];
         yhg=Map[ListDensityPlot[#,DisplayFunction→Identity,FrameTicks→None] &, yh];
         Show[GraphicsArray[yhg]]
```

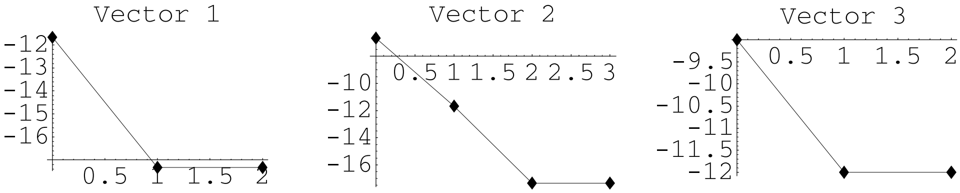


Is the result satisfactory? You can test other noisy letters by repeating the commands.

With `NetPlot` you can plot the energy decrease and the trajectories $x(t)$.

Look at the energy decrease during the evaluation.

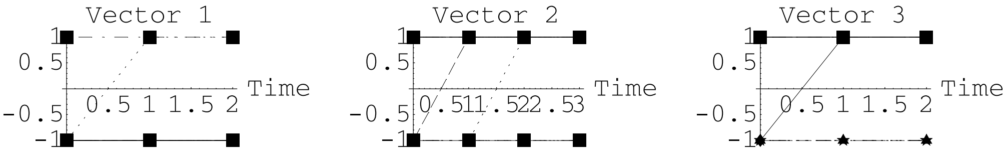
```
In[16]:= NetPlot[hopletter,y,DataFormat→Energy]
```



From the plot you can see that the Hopfield network converged after three discrete-time steps.

Look at the state vectors starting at the noisy letters.

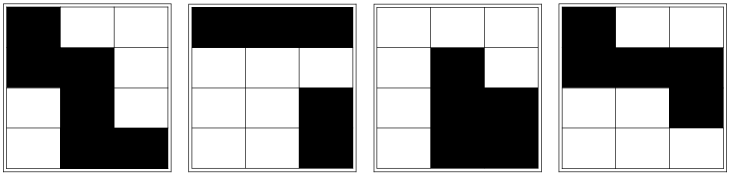
```
In[17]:= NetPlot[hopletter,y]
```



You can also try the Hopfield networks on some randomly generated patterns.

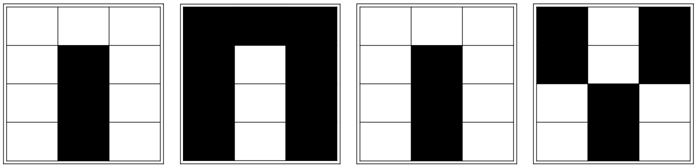
Generate and look at random patterns.

```
In[18]:= letterRand=Sign[RandomArray[UniformDistribution[-1,1], {4, 12}]];
         letterRandMatrix=Map[Partition[#,3] &,letterRand, 1];
         lg = Map[ListDensityPlot[#,DisplayFunction→Identity,FrameTicks→None]
         &,letterRandMatrix];
         Show[GraphicsArray[lg]]
```



Apply the network to each of these patterns and look at the patterns to which they converge.

```
In[22]:= lh=Map[hopletter[#] &, letterRand];
         lh=Apply[Partition[#, 3] &, lh, 1];
         lhg=Map[ListDensityPlot[#, DisplayFunction→Identity,FrameTicks→None] &, lh];
         Show[GraphicsArray[lhg]]
```



You can show that the mirror vectors to the pattern vectors also constitute minima to the energy function and, therefore, the mirror vectors are also possible convergence points of the Hopfield network. It is not uncommon for some of the randomly generated data vectors to converge to these inverses of some of the original letters.

### 9.2.3 Continuous-Time Two-Dimensional Example

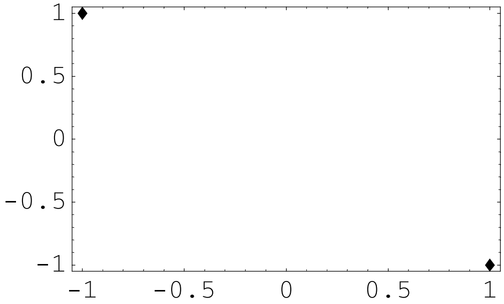
Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

Consider the same two-dimensional example as for the discrete-time Hopfield network. There are two pattern vectors {1, −1} and {−1, 1}, and the goal is to classify noisy versions of these vectors to the correct vector.

Generate and look at class pattern vectors.

```
In[2]:= x={{1,-1},{-1,1}};
        NetClassificationPlot[x]
```



The two pattern vectors are placed in the corners of the plot.

Define a continuous-time Hopfield network with a saturated linear neuron.

Create a continuous-time Hopfield network.

```
In[4]:= hop=HopfieldFit[x,NetType→Continuous,Neuron→SaturatedLinear]

Out[4]= Hopfield[W, {NetType → Continuous, WorkingPrecision → 4,
          CreationDate → {2002, 4, 3, 14, 11, 31}, Dt → 0.2, Neuron → SaturatedLinear}]
```

Provide some information about the Hopfield network.

```
In[5]:= NetInformation[hop]

Out[5]= Continuous Hopfield model for pattern vectors with 2 components. Created 2002-4-3
          at 14:11. Type of neuron: SaturatedLinear[x]. Precision in evaluation: 4
```

The obtained network can be used right away on any data vector by using the evaluation rule for Hopfield objects.

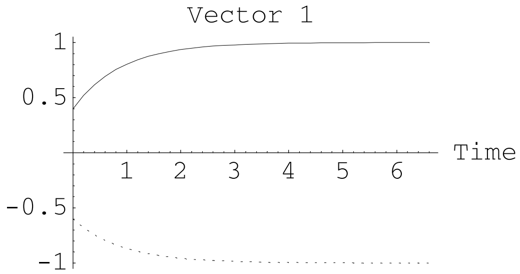
Evaluate the Hopfield network on a data vector.

```
In[6]:= hop[{0.4,-0.6}]

Out[6]= {{0.99962, -0.999746}}
```

Using `NetPlot` you can plot various information, for example, the state trajectories.

Plot the state trajectories.

```
In[7]:= NetPlot[hop,{{0.4, -0.6}}]
```



It might be interesting to obtain the state trajectories of the evaluation of the Hopfield network on the data vectors. This can be done by setting the option `Trajectories→True` in the evaluation of the Hopfield network.

Obtain the state trajectory.

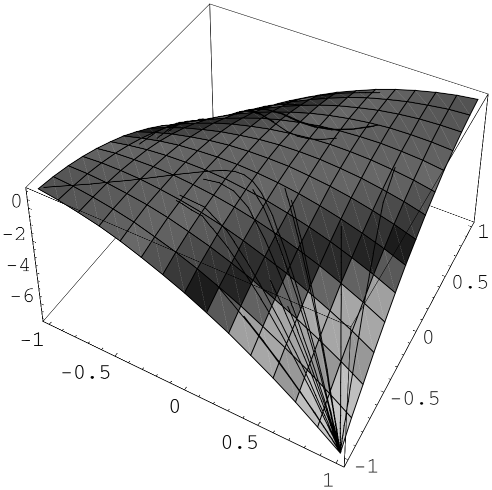
```
In[8]:= hop[{{0.99, -0.99}}, Trajectories → True]

Out[8]= {{{0.99, -0.99}, {0.992, -0.992}, {0.9936, -0.9936}, {0.99488, -0.99488},
          {0.995904, -0.995904}, {0.996723, -0.996723}, {0.997379, -0.997379},
          {0.997903, -0.997903}, {0.998322, -0.998322}, {0.998658, -0.998658},
          {0.998926, -0.998926}, {0.999141, -0.999141}, {0.999313, -0.999313},
          {0.99945, -0.99945}, {0.99956, -0.99956}, {0.999648, -0.999648}}}
```

The trajectory is the numerical solution to Equation 2.28 describing the network (see Section 2.7, Hopfield Network), computed with a time step given by the variable `Dt` in the Hopfield object. It was automatically chosen, when `HopfieldFit` was applied, to ensure a correct solution of the differential equation.

The energy surface and the trajectories of the data vectors can provide a vivid illustration of the classification process. This is only possible for two-dimensional, continuous-time Hopfield nets.

Plot the energy surface together with the trajectories of several data vectors.

```
In[9]:= << Statistics`ContinuousDistributions`
        x = RandomArray[UniformDistribution[-1, 1],  {30, 2}];
        NetPlot[hop,x,DataFormat→Surface]
```



### 9.2.4 Continuous-Time Classification of Letters

Consider classification of the same letters as in the example used with the discrete-time Hopfield network.

Read in the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

A continuous-time Hopfield network will now be used to classify noisy patterns of the letters A, I, and Y. First combine the three letters as three matrices in a list.

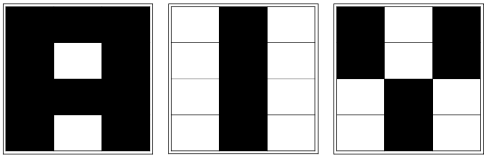
Generate three matrices containing the letters A, Y, and I.

```
In[2]:= x=-{{{1,-1,1},{1,1,1},{1,-1,1},{1,1,1}},
        {{-1,1,-1},{-1,1,-1},{-1,1,-1},{-1,1,-1}},
        {{-1,1,-1},{-1,1,-1},{1,-1,1},{1,-1,1}}};
```

Each matrix element contains the gray level of one pixel. The value 1 corresponds to entirely black and −1 to entirely white.

Look at the letters.

```
In[3]:= xg=Map[ListDensityPlot[#,DisplayFunction→Identity,FrameTicks→None]&,x];
        Show[GraphicsArray[xg]]
```



Before you can construct a Hopfield network, the matrices have to be transformed into vectors.

Transform the matrices to pattern vectors.

```
In[5]:= xv = Map[Flatten, x, {1}];
```

The network is obtained with `HopfieldFit`.

Obtain a continuous-time Hopfield network.

```
In[6]:= hopletter = HopfieldFit[xv, NetType → Continuous, Neuron → SaturatedLinear]

Out[6]= Hopfield[W, {NetType → Continuous,
          WorkingPrecision → 4, CreationDate → {2002, 4, 3, 14, 12, 39},
          Dt → 0.047619, Neuron → SaturatedLinear}]
```

To test if the network can correct noisy data vectors, some noisy versions of the three pattern vectors are created. If everything works out the way it ought to, then these three noisy data vectors will be classified correctly.

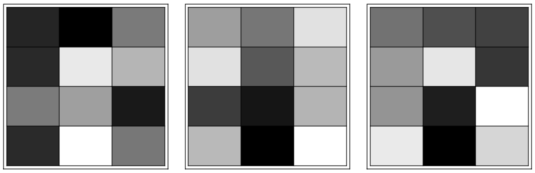
Create three distorted letters from the three given ones.

```
In[7]:= <<Statistics`ContinuousDistributions`
        y=xv*RandomArray[UniformDistribution[-0.3,1], {3,12}];
```

The disturbed letters take real values in the interval {−1, 1}. To look at the disturbed letters, they must first be retransformed into matrices.

Look at the disturbed letters.

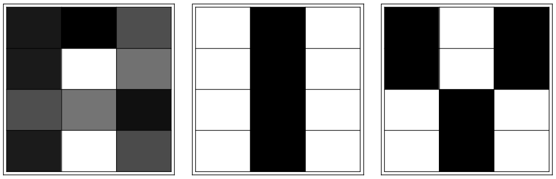
```
In[9]:= ym=Map[Partition[#,3] &, y];
        yg=Map[ListDensityPlot[#, DisplayFunction→Identity,FrameTicks→None] &, ym];
        Show[GraphicsArray[yg]]
```



The Hopfield network will now be used to “repair” the noisy patterns. The result is transformed into matrices and plotted.

Apply the Hopfield network to the noisy data vectors and plot the result.

```
In[12]:= yh=Map[hopletter[#] &, y];
         yh=Apply[Partition[#, 3] &, yh, 1];
         yhg=Map[ListDensityPlot[#, DisplayFunction→Identity,FrameTicks→None] &, yh];
         Show[GraphicsArray[yhg]]
```

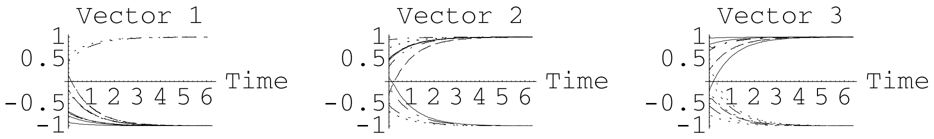


Was the result satisfactory? Because the noisy data vectors were generated randomly, the result may vary from time to time. Repeat the calculations a couple of times with different noisy letters to see the differences.

You can also use `NetPlot` to illustrate the evaluation on the noisy data vectors.

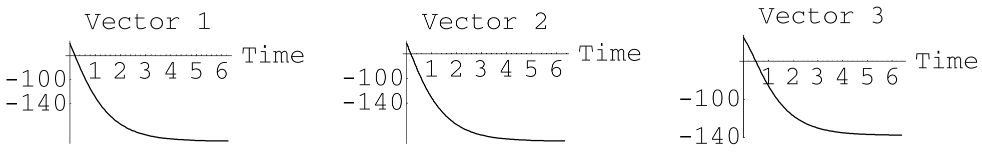
Check how the trajectories develop with time.

```
In[16]:= NetPlot[hopletter, y]
```



Check how the energy decreases with time.

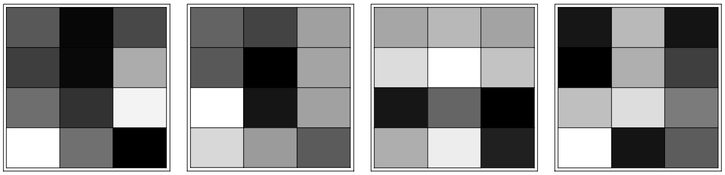
```
In[17]:= NetPlot[hopletter, y, DataFormat → Energy, Ticks → {Automatic, {-60, -100, -140}}]
```



You can also see how the Hopfield network deals with some randomly generated patterns.

Generate and look at random patterns.

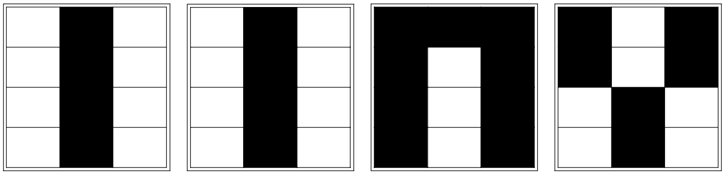
```
In[18]:= letterRand = RandomArray[UniformDistribution[-1, 1], {4, 12}];
         letterRandMatrix = Map[Partition[#, 3] &, letterRand, 1];
         lg = Map[ListDensityPlot[#, DisplayFunction → Identity, FrameTicks → None] &,
            letterRandMatrix];
         Show[GraphicsArray[lg]]
```



Do any of these random patterns look like any of the three letters? What does the Hopfield network say about them? To which energy minima are these random patterns attracted?

Apply the network to the random patterns and look at the result.

```
In[22]:= lh=Map[hopletter[#] &, letterRand];
         lh=Apply[Partition[#, 3] &, lh, 1];
         lhg=Map[ListDensityPlot[#, DisplayFunction →Identity,FrameTicks→None]&, lh];
         Show[GraphicsArray[lhg]]
```



Have the randomly generated patterns been classified as any of the letters? If so, do these results make sense; that is, do the original random patterns resemble these letters? As mentioned previously, it is possible at times to get the inverted versions of the letters. They are also attractors of the Hopfield network; that is, their inversions are also possible convergence points of the network. This is an unfortunate feature of Hopfield networks.

## 9.3 Further Reading

The following texts cover Hopfield networks:

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Reading, MA, Addison-Wesley, 1991.

M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, Cambridge, MA, The MIT Press, 1995.

J. J. Hopfield, “Neural Networks and Physical Systems with Emergent Collective Computational Abilities”, in *Proc. Natl. Acad. Sci. USA, 1982*, vol. 79, pp. 2554–2558.

J. J. Hopfield, “Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons”, in *Proc. Natl. Acad. Sci. USA, 1984*, vol. 81, pp. 3088–3092.

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, Macmillan, 1999.

# 10 Unsupervised Networks

Unsupervised neural networks employ training algorithms that do not make use of desired output data. They are used to find structures in the data, such as clusters of data points or one- or two-dimensional relationships among the data. When such structures are discovered, they can help describe the data in more compact ways.

A short tutorial on unsupervised networks is given in Section 2.8, Unsupervised and Vector Quantization Networks. Section 10.1, Unsupervised Network Functions and Options, describes the functions and their options to work with unsupervised networks. Examples of the use of the commands are given in Section 10.2, Examples without Self-Organizing Maps, and Section 10.3, Examples with Self-Organizing Maps. Section 10.4, Change Step Length and Neighbor Influence, describes how you can change the training algorithm by changing the step length and the neighbor feature.

## 10.1 Unsupervised Network Functions and Options

This section introduces the commands to deal with unsupervised networks, with and without a neighbor map. Examples illustrating the use of the commands follows in Section 10.2, Examples without Self-Organizing Maps, and Section 10.3, Examples with Self-Organizing Maps.

### 10.1.1 InitializeUnsupervisedNet

An unsupervised network is initialized with `InitializeUnsupervisedNet`.

| | |
|---|---|
| `InitializeUnsupervisedNet[`*x*`, `*size*`]` | initializes an unsupervised network of the indicated *size* using supplied data vectors *x* |

Initialize an unsupervised network.

The argument *size* should be a positive integer indicating the number of codebook vectors; the supplied data *x* should be a matrix as described in Section 3.2, Package Conventions.

Unsupervised networks are stored in objects with head UnsupervisedNet, on a format following the general standard of the package, as described in Section 3.2, Package Conventions. The first component is a list of the codebook vectors.

You may write your own initialization or training algorithm. In that case, Section 13.1, Change the Parameter Values of an Existing Network, describes how you insert the parameter values into an unsupervised network.

The options of InitializeUnsupervisedNet are divided into two groups. The first group defines the structure of the network, which could include a neighbor map when a SOM is desired. The second group of options influences the way the network is actually initialized.

The following options define the structure of the unsupervised network.

| *option* | *default value* | |
|---|---|---|
| SOM | None | specifies the neighbor map if changed to a list containing two integers |
| Connect | False | connects the neighbor map into a ring if set to False or a cylinder if True |

Options of InitializeUnsupervisedNet defining the structure of the unsupervised network.

If you want a neighbor map, you set the option SOM to a list of two integers. These two integers define the geometry of the neighbor map, and their product must equal the number of codebook vectors. A one-dimensional SOM is obtained by setting one of the integers to 1 and the second to the number of codebook vectors.

The second group of options influences the initialization procedure of the unsupervised network. They can be used in calls to InitializeUnsupervisedNet and also to UnsupervisedNetFit if no existing network is submitted. If the option UseSOM is set to True, then a SOM will be used for the initialization, and most of the options are only active in this case. Some of the options are more advanced, and they are explained further in the examples in Section 10.4, Change Step Length and Neighbor Influence.

| *option* | *default value* | |
|---|---|---|
| `UseSOM` | `False` | random initialization of the unsupervised net; if set to `True`, a SOM is used in the initialization |
| `Compiled` | `True` | applies the initial SOM training compiled |
| `Iterations` | `10` | number of iterations with the SOM |
| `InitialRange` | `0.01` | standard deviation of the normally distributed initial codebook vectors, normalized by the range of the data vectors |
| `Recursive` | `False` | applies the initial SOM training in batch mode |
| `StepLength` | `Automatic` | step-length function for the initialization algorithm |
| `NeighborStrength` | `Automatic` | positive value, or function, indicating the neighbor strength for the SOM algorithm |
| `Neighbor` | `NonSymmetric` | neighbor topology for the SOM algorithm |
| `CriterionPlot` | `False` | gives no plot presenting the result of the training with the SOM algorithm |
| `CriterionLog` | `False` | logs no information about the training with the SOM algorithm |
| `CriterionLogExtN` | `True` | if the `CriterionLog` option is set to `True`, then presents the training log in a separate notebook |
| `ReportFrequency` | `1` | if the `CriterionLog` option is set to `True`, then logs the performance with this interval during the training |
| `MoreTrainingPrompt` | `False` | prompts for more training iterations if set to `True` |

Options of `InitializeUnsupervisedNet` defining the initialization of the network.

The default initialization of unsupervised networks is to place the codebook vectors randomly around the mean of the data vectors. Then, `InitialRange` is the only option that influences the initialization.

The other possibility is to set `UseSOM→True`, and then apply a few training iterations with a neighbor feature. The risk that some of the codebook vectors might "die"—that is, they might not be used by the data—decreases by using this initialization. All options, except `InitialRange`, influence the initial training with the neighbor feature, and therefore, they only influence the initialization if they are used in combination with `UseSOM→True`.

The default of the options `StepLength` and `NeighborStrength` are the following two functions:

- `StepLength`: `Function[If[# < 3, 0.1, 0.5]]`
- `NeighborStrength`: `Function[If[# < 3, 0.1, (2*#)/10.] ]`

You can change these default values as described later in this section.

### 10.1.2 UnsupervisedNetFit

Unsupervised networks are trained with `UnsupervisedNetFit`. You can choose between submitting an already existing unsupervised model, or have a new network initialized by indicating the number of codebook vectors. You can also indicate the number of training iterations. If left out, the default number of iterations (30) will be applied.

| | |
|---|---|
| `UnsupervisedNetFit[`*x*`,` *size*`]` | initializes and trains an unsupervised net of indicated size the default number of iterations |
| `UnsupervisedNetFit[`*x*`,` *size*`,` *iterations*`]` | initializes and trains an unsupervised net of indicated size the specified number of iterations |
| `UnsupervisedNetFit[`*x*`,` *net*`]` | trains the supplied unsupervised net the default number of iterations |
| `UnsupervisedNetFit[`*x*`,` *net*`,` *iterations*`]` | trains the supplied unsupervised net the specified number of iterations |

Training an unsupervised network.

An existing network can be submitted for more training by setting *net* equal to the network or its training record. The advantage of submitting the training record is that the information about the first training is combined with the additional training.

UnsupervisedNetFit returns a list of two variables. The first output is the trained unsupervised network. It consists of an object with head UnsupervisedNet. The second output, the training record with head UnsupervisedNetRecord, contains logged information about the training. It can be used to analyze the progress of the training, and to validate the model using the command NetPlot. You can also extract intermediate information from the training as described in Section 7.8, The Training Record.

During the training, intermediate results are displayed in a separate notebook, which is created automatically. After each training iteration the mean distance between the data vectors and the closest codebook vector is written out. Using the options of UnsupervisedNetFit, as described in Section 7.7, Options Controlling Training Results Presentation, you can change the way the training results are presented.

The necessary number of training iterations is strongly dependent on the particular problem. Depending on the number of data vectors, their distribution, and the number of codebook vectors, you might need more iterations. At the end of the training, the decrease of the mean distance is shown in a plot. You can use this plot to decide if more training iterations are necessary.

Sometimes you also receive a warning at the end of the training saying that there is at least one codebook vector that is not used by the data. This indicates that there are nuisance codebook vectors, or *dead* neurons, that do not have any effect on the training data. In general you do not want any dead codebook vectors, and there are various measures you can take. For example, you can

- Re-initialize the unsupervised network using the option UseSOM→True. This usually gives a better initialization as described later.
- Repeat the training from a different initialization. The initialization and training contain some randomness and by repeating these commands you obtain a new realization that might be better.
- Change the size of the unsupervised network by changing the number of codebook vectors in the initialization.
- Identify the unused codebook vectors with UnUsedNeurons and remove them using NeuronDelete.

UnsupervisedNetFit takes basically the same options as InitializeUnsupervisedNet, but the default values are different.

| *option* | *default value* | |
|---|---|---|
| SOM | None | specifies the neighbor map if changed to a list containing two integers |
| Connect | False | connects the neighbor map into a ring if False or a cylinder if True |
| Compiled | True | uses compiled code |
| Recursive | True | trains in recursive mode |
| StepLength | Automatic | step-length function for the training algorithm |
| NeighborStrength | Automatic | positive value, or function, indicating neighbor strength |
| Neighbor | NonSymmetric | neighbor topology |
| CriterionPlot | True | plots the result of the training |
| CriterionLog | True | logs information about the training |
| CriterionLogExtN | True | presents the training log in a separate notebook |
| ReportFrequency | 1 | logs the performance with this interval during the training |
| MoreTrainingPrompt | False | prompts for more training iterations if set to True |

Options of UnsupervisedNetFit.

The options CriterionPlot, CriterionLog, CriterionLogExtN, ReportFrequency, and MoreTrainingPrompt are common in the other training commands in the *Neural Networks* package, and they are described in Section 7.7, Options Controlling Training Results Presentation.

By giving new values to SOM and Connect in the call to UnsupervisedFit, it is possible to change the neighbor map of an existing unsupervised network. Examples of how this is done can be found in Section 10.3.3, Adding a SOM to an Existing Unsupervised Network.

The options NeighborStrength and Neighbor only influence the algorithm if the unsupervised network has a neighbor map attached to it. Examples illustrating these options are given in Section 10.4, Change Step Length and Neighbor Influence.

The options Recursive, StepLength, NeighborStrength, and Neighbor are used to modify the training algorithm. They are of a more advanced nature and are further described in this section.

An unsupervised network can be evaluated on one data vector, or a list of data vectors, using the function evaluation rule. The output is a list containing the number of the codebook vector closest to the data vector. This evaluation rule is actually all you need to start using the unsupervised network.

| | |
|---|---|
| *net*[*x*] | evaluates the *net* on the input vector *x* |

Function evaluation of an unsupervised network.

The input argument *x* can be a vector containing one input sample or a matrix containing one input sample in each row.

The function evaluation rule also has an option.

| *option* | *default value* | |
|---|---|---|
| SOM | Automatic | indicates whether the output should be the number of the winning neuron or its coordinates within the SOM map |

Option of the evaluation of an unsupervised network.

The default Automatic is changed to True or False depending on whether or not the unsupervised network has a SOM feature. If it does, then the default gives the position of the winning codebook vector within the SOM structure. If you supply the option SOM→False, then the SOM feature is not used in the evaluation, and you receive the number of the winning codebook vector. This is illustrated in Section 10.3.1, Mapping from Two to One Dimensions.

### Details and Algorithms

Further described are more advanced options for UnsupervisedNetFit. They can be used to modify the training algorithm from the default version in a way that might better suit your problem.

The codebook vectors can either be adapted in a recursive manner, considering one data sample in each update, or in batch mode where all data is used at each step. The algorithm to be used is indicated by the Recursive option. Also, the algorithm will vary depending on whether or not a neighbor feature is applied.

**The recursive algorithm for unsupervised networks (Standard competitive learning rule):**

Given $N$ data vectors $\{x_k\}$, $k = 1, \ldots, N$, in each update, the following steps are performed.

1. $k$ is chosen randomly from a uniform integer distribution between 1 and $N$, where the whole range is considered each time this step is executed.
2. The codebook vector closest to $x_k$, called the *winning neuron* or the *winning codebook vector*, is identified. Its index is indicated by $i$.
3. The winning codebook vector is changed according to

$$\texttt{w}_\texttt{i} = \texttt{w}_\texttt{i} + \texttt{SL[n]} * (\texttt{x}_\texttt{k} - \texttt{w}_\texttt{i}) \qquad (1)$$

   where $n$ is the iteration number.

4. The described steps are repeated $N$ times in each iteration.

Abbreviations have been used; $SL[n]$ is the `StepLength` function, and it can be changed by the option with the same name.

If the unsupervised network contains a neighbor feature, then the following recursive algorithm applies.

**The recursive algorithm for SOM (Kohonen's algorithm):**

Given $N$ data vectors $\{x_k\}$, $k = 1, \ldots N$, in each update, the following steps are performed.

1. $k$ is chosen randomly from a uniform integer distribution between 1 and $N$, where the whole range is considered each time this step is executed.
2. The codebook vector closest to $x_k$, called the winning neuron or the winning codebook vector, is identified. Its index is indicated by $\{i_{win}, j_{win}\}$.
3. All the codebook vectors are changed according to

$$\begin{aligned}&\texttt{w}_\texttt{i,j} = \texttt{w}_\texttt{i,j} + \\ &\quad \texttt{SL[n]} * \texttt{Exp[-NS[n]} * \texttt{NM[[c}_1 - \texttt{i}_\texttt{win} + \texttt{i, c}_2 - \texttt{j}_\texttt{win} + \texttt{j]]]} * (\texttt{x}_\texttt{k} - \texttt{w}_\texttt{i,j})\end{aligned} \qquad (2)$$

   where $n$ is the iteration number and $\{c_1, c_2\}$ is the center position of the neighbor matrix, the value given with the `SOM` option.

4. The described steps are repeated $N$ times in each iteration.

Abbreviations have been used; $SL[n]$ is the StepLength function and $NS[n]$ is the NeighborStrength function. They can both be changed by the options with the same names. $NM$ is the neighbor matrix dictating which codebook vectors are neighbors, and it can also be chosen by the user. All codebook vectors are changed toward the data vector. The neighbor matrix $NM$ should have its minimum at its center element $\{c_1, c_2\}$, so that the winning neuron update is most pronounced. Typically, the elements of $NM$ further away from the center take larger values, so that codebook vectors further away from the winner are changed less. One *iteration* of the stochastic algorithm (that is, $n$ incremented by 1), consists of $N$ updates via Equation 10.2. Note that due to the fact that $k$ is chosen independently in each update, the data use is indeterminate.

With Recursive→False all data is used in each iteration. The training follows a deterministic scheme where the mean of the update Equation 10.1, or Equation 10.2, over all data $\{x_k\}$, $k = 1, \ldots, N$ is used. In this case, the unsupervised training without a neighbor feature becomes equivalent to what is called a *k-means clustering.*

The intended use of UnsupervisedNetFit is to employ the recursive algorithm at the beginning of the training, and then, possibly, take a few steps with the batch algorithm to fine tune the neurons. When UnsupervisedNetFit is used in other ways you should consider changing the two options StepLength and NeighborStrength.

**The StepLength option:**

The StepLength option has the default value Automatic. Depending on the value of the Recursive option, this is changed into one out of two different functions.

```
Recursive → True: Function[n, If[n<5, 0.01, 2./(3+n)]]

Recursive → False: Function[n, 1]
```

In the recursive case, the step length is small during the first iterations so that the codebook vectors find a good orientation. Then, the step length is increased to speed up the convergence. From this higher value, the step length is then slowly decreased again. Convergence can only be guaranteed if the step length converges toward zero.

For a batch update, the step length is set to one for all iterations. This is a good choice if the codebook vectors are close to their optimal values, so that the step becomes small anyway. The equation may become unstable if such a large step length is used when this is not the case.

You can choose other step lengths in two different ways. A constant step length is obtained by giving StepLength a numerical value in the range {0, 1}. The other possibility is to set the option to a function that takes the iteration number as input and delivers a numerical value as output. In Section 10.4, Change Step Length and Neighbor Influence, you find an example showing how the step length may be changed.

### The NeighborStrength option:

The `NeighborStrength` option works similarly to `StepLength`, but it is active only if there is a neighbor feature adapted to the network. Depending on the `Recursive` option the default `Automatic` is changed into:

```
Recursive → True: Function[n, If[n<5, 0.1, (n-4)/10.] ];

Recursive → False: Function[n, 1000.]
```

In the recursive case, during the first five iterations, the neighbors of the winning neuron are influenced strongly (low value > 0). This helps the network to conform to a nice structure and avoid "knots." Subsequently, the influence on the neighbors gradually decreases (value increases).

When the algorithm is applied in batch mode, the neighbor strength function has a constant value of 1000, which imparts only a negligible influence on the neighboring codebook vectors. Therefore, in batch mode, only the winning neurons are adapted. This is good when the batch mode is used to fine-tune the final positions of the codebook vectors, after the recursive training has been applied.

A positive constant neighbor strength can be specified using `NeighborStrength`. You can also use any function that takes the iteration number as input and gives the neighbor strength as output. In Section 10.4, Change Step Length and Neighbor Influence, you find an example showing how the `NeighborStrength` option can be changed.

### The Neighbor option:

As is the case with `NeighborStrength`, the `Neighbor` option also has no meaning unless a neighbor feature is attached to the unsupervised network. The `Neighbor` option lets you specify which neurons, or codebook vectors, are neighbors. There are two standard possibilities that are specified by setting the `Neighbor` option to `NonSymmetric` (default) or `Symmetric`. The nonsymmetric choice gives a stronger connection to the neighbors on one side than on the other side. This should make it easier to avoid "knots" on the map. With these standard choices, neighbor matrices of the correct dimensions are computed internally. The symmetric option gives a neighbor matrix

$$\begin{pmatrix} 2 \\ 1 \\ 0 \\ 1 \\ 2 \end{pmatrix}$$

for a one-dimensional network with three codebook vectors. The zero is at the center position, and it corresponds to the winning codebook vector. The matrix has larger values away from the center position in both directions. Since the size of the matrix elements indicates the distance between the codebook vector and the winning neuron, a larger value means that the distance is also larger. The nonsymmetric alternative gives

$$\begin{pmatrix} 4 \\ 2 \\ 0 \\ 1 \\ 2 \end{pmatrix}$$

in a one-dimensional network with three codebook vectors. For a two-dimensional network of size {3, 4} you obtain the following neighbor matrices with the nonsymmetric alternative

$$\begin{pmatrix} 10 & 8 & 6 & 4 & 5 & 6 & 7 \\ 8 & 6 & 4 & 2 & 3 & 4 & 5 \\ 6 & 4 & 2 & 0 & 1 & 2 & 3 \\ 7 & 5 & 3 & 1 & 2 & 3 & 4 \\ 8 & 6 & 4 & 2 & 3 & 4 & 5 \end{pmatrix}$$

and with the symmetric alternative

$$\begin{pmatrix} 5 & 4 & 3 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 0 & 1 & 2 & 3 \\ 4 & 3 & 2 & 1 & 2 & 3 & 4 \\ 5 & 4 & 3 & 2 & 3 & 4 & 5 \end{pmatrix}$$

You can use the `Neighbor` option and submit your own neighbor matrix. It should then have dimensions $\{2c_1 - 1, 2c_2 - 1\}$, where $\{c_1, c_2\}$ are the dimensions of the SOM map. In Section 10.4, Change Step Length and Neighbor Influence, you will find an example showing how the `Neighbor` option can be changed.

**The Connect option:**

If `Connect` is changed to `True`, then the neighbor matrix is changed so that the SOM network is connected to a ring in the one-dimensional case, and into a cylinder in the two-dimensional case (in the first of the two dimensions). This holds only if you use one of the two values `NonSymmetric` or `Symmetric` for the `Neighbor` option. If you instead supply your own neighbor matrix, then the `Connect` option does not have any meaning, and you have to specify the neighbor matrix directly so that it corresponds to a ring. The neighbor matrix generated by setting `Connect→True` and `Neighbor→Symmetric` for a one-dimensional SOM network with six codebook vectors is

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 1 \\ 0 \\ 1 \\ 2 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

The element values indicate the distance from the winning neuron, which is in the center, that is, it has distance zero from itself. The codebook vectors six positions away from the winner have distance one, the same as the codebook vector in the position next to the winner. Therefore, one end of the line is connected to the other end.

### 10.1.3 NetInformation

Some information about the unsupervised networks is presented in a string by the function `NetInformation`.

| | |
|---|---|
| `NetInformation[`*unsup*`]` | gives information about an unsupervised net |

The `NetInformation` function.

`NetInformation` takes no options.

### 10.1.4 UnsupervisedNetDistance, UnUsedNeurons, and NeuronDelete

The function `UnsupervisedNetDistance` gives the average Euclidian distance between the data vectors and the nearest codebook vector of the submitted unsupervised network.

| | |
|---|---|
| `UnsupervisedNetDistance[`*net*`, `*x*`]` | gives the mean Euclidian distance<br>between data *x* and nearest codebook vector |

Performance of an unsupervised network.

A small value indicates that all data vectors are close to a codebook vector, and then the clustering can be considered successful.

`UnsupervisedNetDistance` has one option.

| *option* | *default value* | |
|---|---|---|
| `Compiled` | `True` | use compiled version |

Option of `UnsupervisedNetDistance`.

The function `UnUsedNeurons` indicates the codebook vectors that are not closest to any data vector. These codebook vectors that are unused on the training data set can be considered to have "died," and do not contribute to the clustering. Instead, they can be considered as nuisance vectors. Therefore, it is always interesting to test which, if any, codebook vectors are not being used. You may use the command `NeuronDelete` to remove these codebook vectors from the network.

| | |
|---|---|
| `UnUsedNeurons[`*net*`,` *x*`]` | gives the numbers of codebook vectors not used by the data *x* |

Finding the unused codebook vectors.

The output is a list containing the numbers of the codebook vectors that are not closest to any of the supplied data.

Sometimes it might be of interest to remove some of the codebook vectors from an existing unsupervised network, for example, the ones pointed out by `UnUsedNeurons`. `NeuronDelete` can be used to do this.

| | |
|---|---|
| `NeuronDelete[`*net, pos*`]` | deletes the codebook vectors<br>indicated with *pos* in an existing network *net* |

Deleting codebook vectors in an existing unsupervised network.

`NeuronDelete` has one option.

| *option* | *default* | |
|---|---|---|
| `DeleteSOM` | `False` | removes the neighbor structure from the network |

Option of `NeuronDelete`.

The argument *pos* indicates the codebook vectors to be deleted. It is used differently depending on whether the network has a SOM.

If there is no SOM, or if `DeleteSOM→True`, then *pos* should be an integer indicating the number of the codebook vector to be deleted. Several codebook vectors can be removed by supplying a list of integers.

If the network has a neighbor structure, SOM, and if it is not removed using `DeleteSOM→True`, then you have to delete a whole row or a column of codebook vectors simultaneously. This is done by setting *pos* to a list with two elements. Both of these elements are also lists: the first one containing the numbers of the rows to be deleted and the second one containing the numbers of the columns to be deleted. That is, *pos* = {{*m*, *n*, …}, {*k*, *l*, …}} removes rows *m* and *n* and columns *k* and *l*. If you delete only rows or columns, then one of the elements becomes an empty list.

### 10.1.5 NetPlot

The `NetPlot` command is intended to support the illustration of unsupervised networks and their training.

An existing unsupervised network can be evaluated on a data vector directly using the evaluation rule. The output indicates the codebook vector to which the data vector is mapped. This evaluation rule is actually all you need for the unsupervised network. The command `NetPlot` merely complements the evaluation rule by graphically illustrating the result of the unsupervised network.

The function `NetPlot` supports the illustration of the clustering in several ways, depending on the choice of the option `DataFormat`. An unsupervised network or the training record can be submitted, the second argument given by `UnsupervisedNetFit`. If a training record is submitted, then the progress during the training is illustrated. Consequently, in the calls in the following table, *net* can be either a trained unsupervised network or a training record.

| | |
|---|---|
| `NetPlot[`*net*`, `*x*`, `*y*`]` | illustrates how *net* clusters data *x* to given classes *y* |
| `NetPlot[`*net*`, `*x*`]` | illustrates how *net* clusters data *x* |

Illustrating an unsupervised network or its training.

An output *y*, indicating valid classes of the data vectors, must be in the correct format as described in Section 3.2, Package Conventions.

The option `DataFormat` controls the way the clustering result is illustrated, and depending on its value, some of the other options may become obsolete.

The default in two-dimensional clustering problems is to plot the data together with the codebook vectors. The positions of the codebook vectors are indicated by their numbers.

`NetPlot` takes the following options.

| *option* | *default value* | |
|---|---|---|
| `DataFormat` | `Automatic` | indicates how the clustering should be illustrated;<br>the default depends on the dimension<br>of the data (different possibilities follow) |
| `Voronoi` | `True` | displays Voronoi cells; if set to `False`,<br>indicates the positions of the codebook vectors with crosses |
| `CbvSymbol` | `Automatic` | changes the mark indicating the codebook vectors |
| `Intervals` | `5` | interval of training iterations between plots in the graphics array |
| `SOM` | `Automatic` | uses the net’ s neighbor feature if one exists |
| `Compiled` | `True` | uses compiled version |

Options of `NetPlot`.

The option `DataFormat` takes the following values.

| | |
|---|---|
| `DataMap` | the default for two-dimensional problems;<br>gives a plot of data together with the codebook vectors |
| `Table` | gives a table with one box for each codebook vector;<br>each box contains the number of data vectors<br>from each class assigned to this codebook vector |
| `DataMapArray` | gives a graphics array of the<br>progress of the clustering during training;<br>applies only to two-dimensional problems |

Possible values of `DataFormat`.

In addition to these options, you can submit others to modify the graphical output. Depending on the chosen option for `DataFormat`, the graphical output is created by `BarChart`, `BarChart3D`, `MultipleListPlot`, `ListPlot`, or `Plot3D`.

If the unsupervised network has no neighbor feature, then the classification boundaries among the different codebook vectors are marked with Voronoi cells. From such a plot, it is easy to see codebook vectors that are not being used, that is, whether they can be considered dead. If `Voronoi→True`, then the default of `CbvSymbol` is to indicate each codebook vector with its number in the network. You may change this by submitting a list of symbols. You may also use this option to include options of `Text`, which then modify only the codebook's marks and not the plot label and the axes.

If `Voronoi→False`, then the positions of the codebook vectors are indicated with crosses. You may change the size of the crosses by setting `CbvSymbol` to a list of two numbers indicating the size in each dimension.

If the submitted unsupervised network has a neighbor feature (that is, if it is a SOM network), then the default is to plot the neighbor map instead of the Voronoi cells. This can be avoided by setting `SOM→False`.

In a two-dimensional problem, submitting a training record, instead of the network itself, results in a plot that shows how the codebook vectors are changed from their initial positions to the final ones. The final classification boundaries are also shown.

## 10.2 Examples without Self-Organizing Maps

In this section some examples are given where unsupervised networks are used to perform clustering. The first example is in a two-dimensional space so that the result can be visualized. The second example is in a three-dimensional space. The last example illustrates potential difficulties you could encounter in using unsupervised networks.

The possibility of using a neighbor feature in an unsupervised network, and turning it into a Kohonen network, is illustrated in the next subsection.

Notice that if you reevaluate the examples you will not obtain exactly the same results due to the randomness in the initialization.

### 10.2.1 Clustering in Two-Dimensional Space

Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << sixclasses.dat;
```

The data set $x$ is a matrix where each row is a data vector following the standard format described in Section 3.2, Package Conventions. The number of columns indicates the dimensionality of the input space.

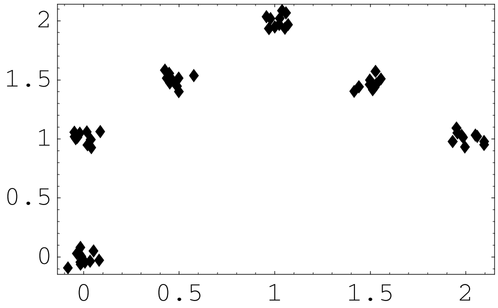
Check the number of data vectors and the dimensionality of the data space.

```
In[3]:= Dimensions[x]

Out[3]= {60, 2}
```

There are 60 data vectors with a dimensionality of two. Consequently, you can visualize the data by using `NetClassificationPlot`.

Look at the data.

```
In[4]:= NetClassificationPlot[x]
```



The six clusters can easily be recognized. Because the clusters are seen to follow an arc along which they appear to be separable, they obviously share a one-dimensional relationship. The goal of using an unsupervised network is to automatically find these clusters. If you also want to automatically find the one-dimensional relationship among the clusters, then you should give the unsupervised network a one-dimensional neighbor structure.

Although the unsupervised training algorithms do not use any output data, it can still be interesting to keep track of the clusters to which the various data vectors belong. This information is contained in the variable $y$, which has the general structure of an output data matrix in a classification problem, described in Section 3.2, Package Conventions. This means that it has one row for each data vector and one column for each cluster.

Check to which cluster data vector 43 belongs.

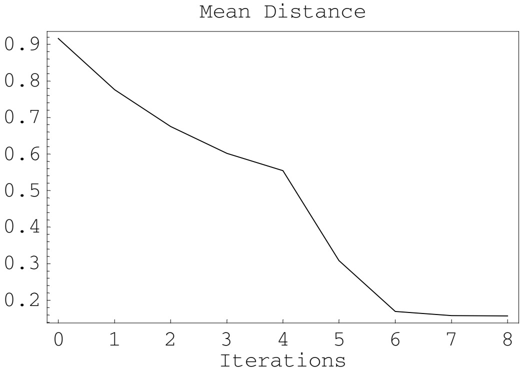
```
In[5]:= y[[43]]

Out[5]= {0, 0, 0, 0, 1, 0}
```

It belongs to cluster 5.

In the initialization of the unsupervised network you specify a number of neurons that correspond to the number of clusters you want to fit to the data.

Initialize and estimate an unsupervised network with eight neurons.

```
In[6]:= unsup = InitializeUnsupervisedNet[x, 8];
        {unsup, fitrecord} = UnsupervisedNetFit[x, unsup, 8, ReportFrequency → 1];
```



```
UnsupervisedNet::DeadNeuron :
 Some codebook vectors are not used by the data. They
   have 'died' during the training. Use UnUsedNeurons to find
   out which they are and decide if you want to remove them.
```

A warning about dead neurons means that some of the codebook vectors were not used to cluster the data. This is no surprise since you know that there are six clusters and you have eight codebook vectors. Unused neurons are often a nuisance, and you will soon learn how they can be removed using NeuronDelete.

The obtained unsupervised network can be used right away to classify new data vectors to one of the clusters. This is done by using the evaluation rule of the network.

Classify a new data vector with the trained unsupervised network.

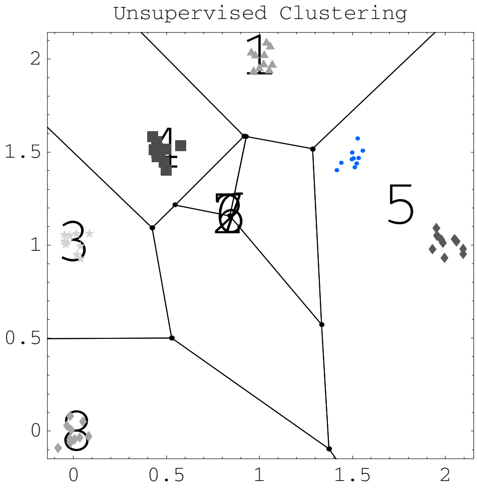
```
In[8]:= unsup[{0.2, 0.6}]

Out[8]= {3}
```

The output is the number of the codebook vector that is closest to the data vector.

You can also illustrate the division of the space into the different clusters using `NetPlot`.

Illustrate the trained unsupervised network together with the data.

```
In[9]:= NetPlot[unsup, x, y,
          SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]},
          CbvSymbol → {TextStyle → {FontSize → 30}}]
```



The position of the codebook vectors are indicated with their numbers together with the data vectors. You may submit any `Text` options in `CbvSymbols` to modify numbers indicating the codebook vectors; in the preceding example, a large font was used. The classification boundaries form Voronoi cells. From this plot it is easy to see which codebook vectors are not being used, that is, whether they can be considered dead. They can also be obtained with UnUsedNeurons.

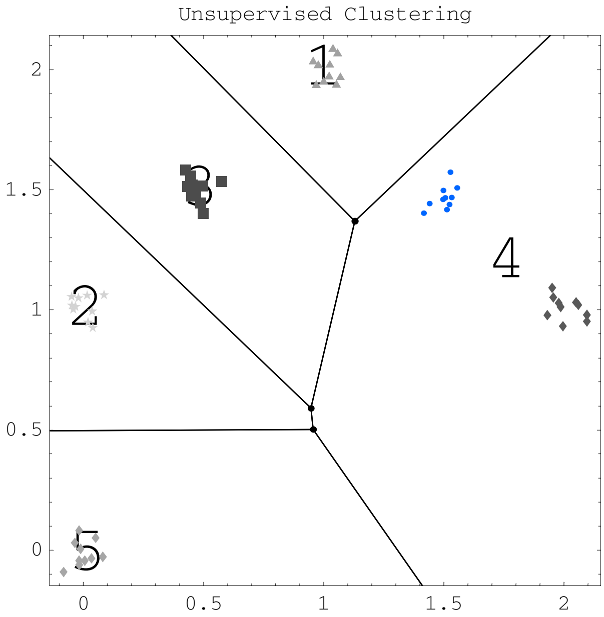
Find and remove the unused codebook vectors.

```
In[10]:= UnUsedNeurons[unsup, x]

Out[10]= {2, 6, 7}

In[11]:= unsup = NeuronDelete[unsup, %]

Out[11]= UnsupervisedNet[{-Codebook Vectors -}, {CreationDate → {2002, 4, 3, 14, 19, 42},
           AccumulatedIterations → 8, SOM → None}]
```

Due to the randomness in the initialization and in the training, it is likely that the result will change if you reevaluate the example.

Look at the clustering of the modified network.

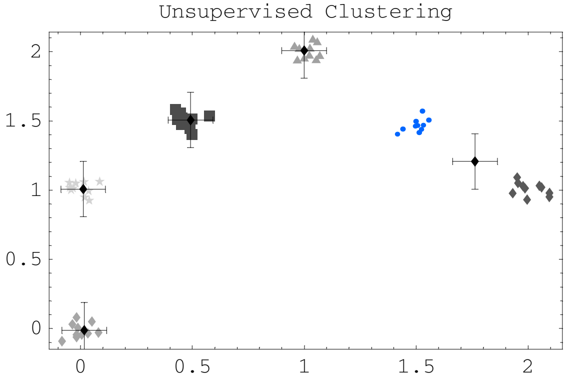
```
In[12]:= NetPlot[unsup, x, y,
          SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]},
          CbvSymbol → {TextStyle → {FontSize → 30}}]
```



You can change the options and plot the result in a different way. For example, by setting `Voronoi→False` the positions of the codebook vectors are indicated with crosses. You can change the size of the crosses with `CbvSymbol`.

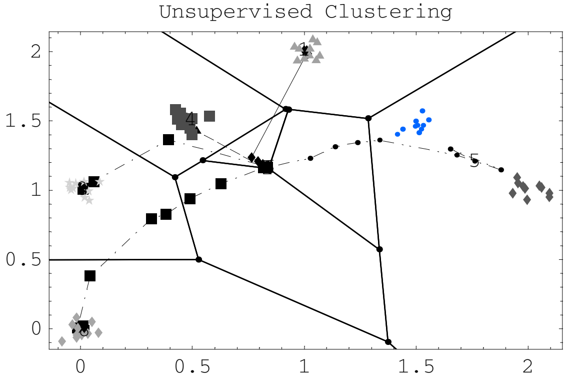
Plot the positions of the estimated clusters.

```
In[13]:= NetPlot[unsup, x, y, Voronoi → False, CbvSymbol → {0.1, 0.2},
           SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



By submitting a training record to `NetPlot`, you obtain an illustration as to how the training proceeded.

```
In[14]:= NetPlot[fitrecord, x, y,
           SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



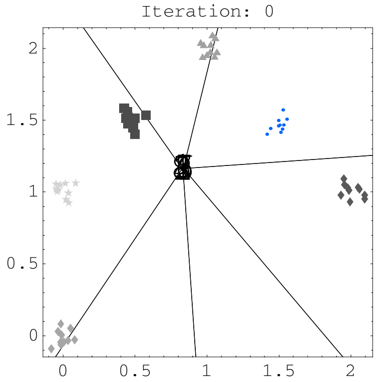
The plot shows how the codebook vectors are changed from their initial positions to the final ones. The final positions of the classification boundaries are also shown.

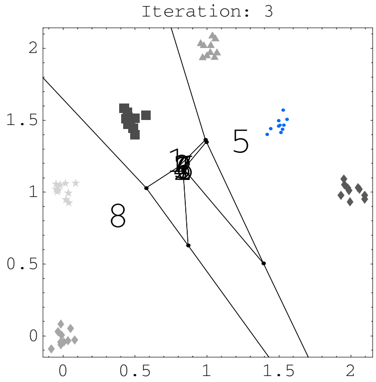
You can also obtain intermediate plots during the training.

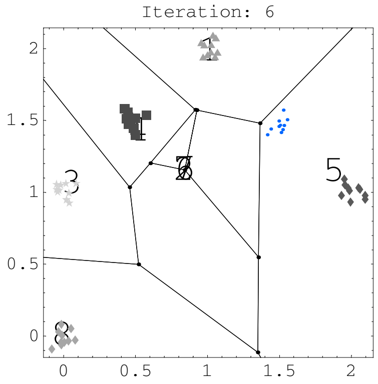
Plot the clustering during the training.

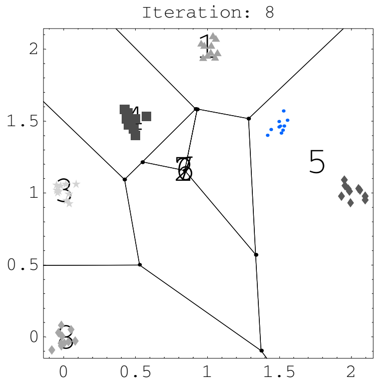
```
In[15]:= NetPlot[fitrecord, x, y, DataFormat → DataMapArray,
           SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]},
           Intervals → 3, CbvSymbol → {TextStyle → {FontSize → 20}}]
```

```
Unsupervised Clustering after
```


Iteration: 0


Iteration: 3





If you prefer an animation of the training progress, you can load `<<Graphics`Animation`` and then change the command to `Apply[ShowAnimation,NetPlot[fitrecord,x,y,Intervals→1, DataFormat→DataMapArray,DisplayFunction→Identity]]`.

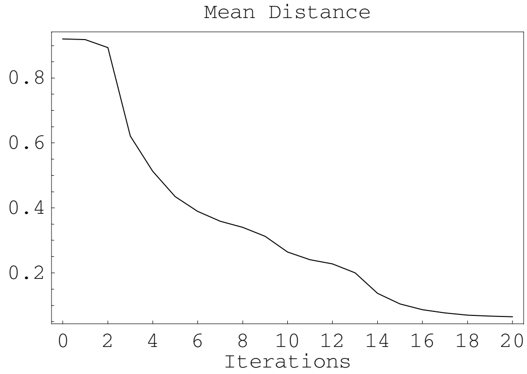
You can also check the mean distance between the data vectors and the codebook vectors.

```
In[16]:= UnsupervisedNetDistance[unsup, x]

Out[16]= 0.156935
```

Initialization is random, by default. Instead, you can use a SOM network to initialize the unsupervised network, which usually avoids the problem with dead neurons. You can also submit options to `InitializeUnsupervisedNet` so that the result of the initial fitting with SOM is reported.

Initialize an unsupervised network with six codebook vectors.

```
In[17]:= unsup = InitializeUnsupervisedNet[x, 6, UseSOM → True,
           CriterionPlot → True, CriterionLog → True, Iterations → 20]
```

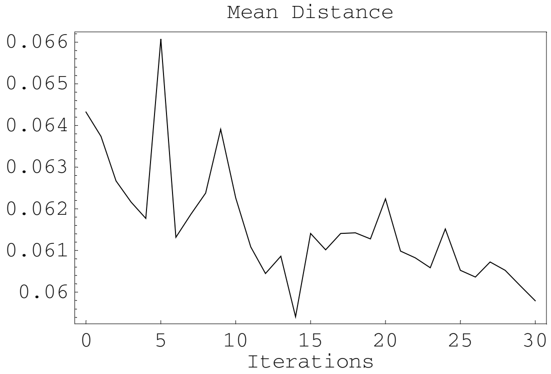


```
Out[17]= UnsupervisedNet[{-Codebook Vectors-}, {CreationDate → {2002, 4, 3, 14, 19, 47},
           AccumulatedIterations → 0, SOM → None, Connect → False}]
```

Note that although it was specified that SOM should be used in the initialization, the obtained network has no SOM. If you want the network to have a SOM, you must specify this with the option `SOM`. See Section 10.1.1, InitializeUnsupervisedNet, to learn how this is done.

If `UseSOM` is set to `True`, then the initialization usually gives a fairly good result, but the performance can be improved by further training.

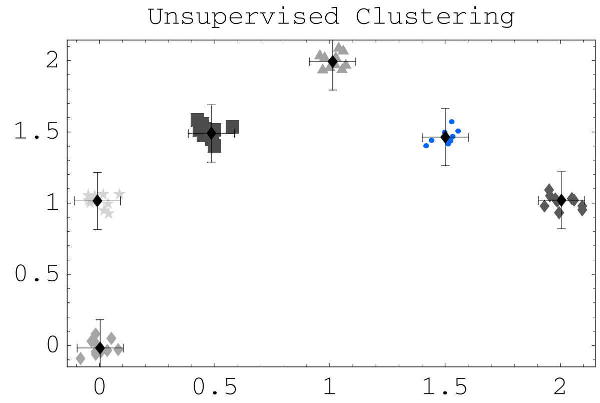
```
In[18]:= {unsup, fitrecord} = UnsupervisedNetFit[x, unsup];
```



This time there are no warnings about dead neurons. Look at the result.

Plot the obtained unsupervised network together with the data.

```
In[19]:= NetPlot[unsup, x, y, Voronoi → False, CbvSymbol → {0.1, 0.2},
          SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



The clustering is more successful than earlier. There is one codebook vector at each cluster.

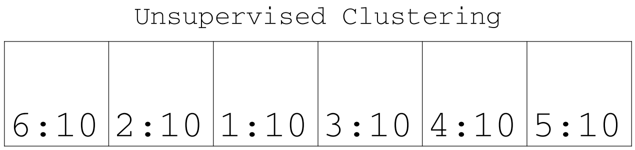
Check if the mean distance between the data vectors and the codebook vectors is smaller now.

```
In[20]:= UnsupervisedNetDistance[unsup, x]

Out[20]= 0.0597949
```

The clustering result can also be illustrated in a table in the following way.

Verify to which codebook vector the different data vectors are mapped.

```
In[21]:= NetPlot[unsup, x, y, DataFormat → Table]
```



```
              Unsupervised Clustering
 6:10 | 2:10 | 1:10 | 3:10 | 4:10 | 5:10
```

There is one box for each codebook vector, and in each box the number of data vectors from each class belonging to that codebook vector is indicated. If you do not submit any data indicating the correct class, all data will be considered belonging to the same class. This gives the following, less informative table, where only the number of data vectors belonging to each cluster is indicated.

```
In[22]:= NetPlot[unsup, x, DataFormat → Table]
```

```
                 Unsupervised Clustering
┌─────┬─────┬─────┬─────┬─────┬─────┐
│     │     │     │     │     │     │
│1:10 │1:10 │1:10 │1:10 │1:10 │1:10 │
└─────┴─────┴─────┴─────┴─────┴─────┘
```

### 10.2.2 Clustering in Three-Dimensional Space

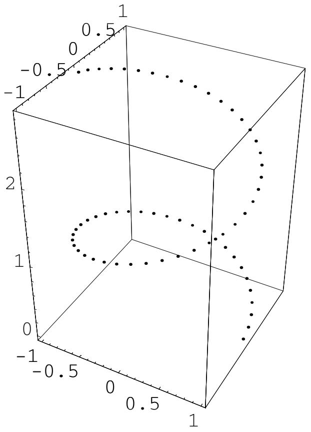
In this example, the unsupervised network is used to cluster three-dimensional data vectors. You can modify the data generating commands and rerun the example to test other data sets.

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

Generate and plot the data along a curve.

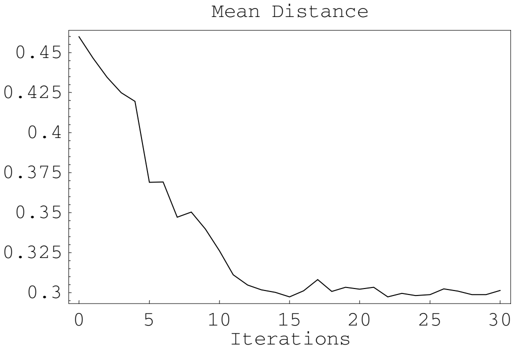
```
In[2]:= << Graphics`Graphics3D`;
        x = N[Table[{ Cos[t],  Sin[t], 0.3 t},
             {t, 0, 3 Pi, Pi / 20}]];
        xgraphics = ScatterPlot3D[x, AxesEdge → {{-1, -1}, Automatic, Automatic}]
```



The data set does not consist of several clusters. Instead, the data is distributed along a curve. With an unsupervised network it should be possible to detect and describe the data along the curve.

Initialize and train an unsupervised network with ten codebook vectors.

```
In[5]:= unsup = InitializeUnsupervisedNet[x, 10, UseSOM → True];
        {unsup, fitrecord} = UnsupervisedNetFit[x, unsup];
```

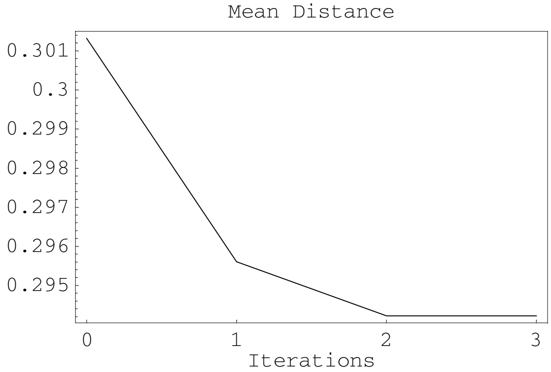


```
UnsupervisedNet::DeadNeuron :
 Some codebook vectors are not used by the data. They
   have 'died' during the training. Use UnUsedNeurons to find
   out which they are and decide if you want to remove them.
```

If the displayed average distance has not converged, it could be worth continuing with additional training iterations. This can be done with the batch training algorithm, which typically allows faster convergence if the codebook vectors are close to the minimum.

Apply three training iterations with the batch training algorithm.

```
In[7]:= {unsup, fitrecord} =
          UnsupervisedNetFit[x, unsup, 3, ReportFrequency → 1, Recursive → False];
```



```
UnsupervisedNet::DeadNeuron :
 Some codebook vectors are not used by the data. They
   have 'died' during the training. Use UnUsedNeurons to find
   out which they are and decide if you want to remove them.
```

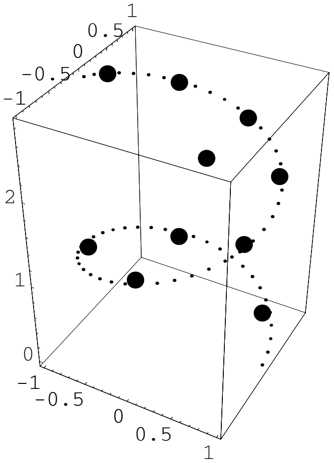
```
In[8]:= unsup
```

```
Out[8]= UnsupervisedNet[{-Codebook Vectors-}, {CreationDate → {2002, 4, 3, 14, 22, 17},
          AccumulatedIterations → 33, SOM → None}]
```

The codebook vectors are stored in the first argument of the `UnsupervisedNet` object. They can easily be extracted and plotted.

Extract and plot the codebook vectors together with the original data vectors.

```
In[9]:= cbv = unsup[[1]];
        cbvgraphics =
          ScatterPlot3D[cbv, PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
        Show[cbvgraphics, xgraphics, DisplayFunction → $DisplayFunction,
         AxesEdge → {{-1, -1}, Automatic, Automatic}]
```



## 10.2.3 Pitfalls with Skewed Data Density and Badly Scaled Data

In this section two examples are given whose data distributions are shown to impede successful clustering. To some extent, this difficulty may be avoided by preprocessing the data using an appropriate linear transformation.

Read in the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

### Clusters with Uneven Data Density

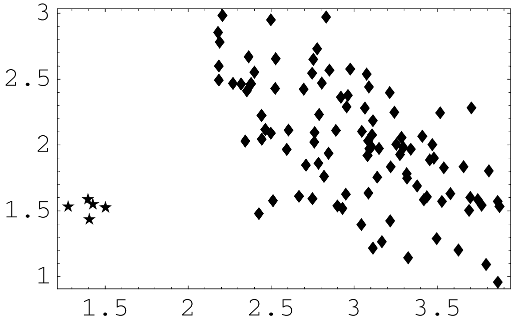
The data used consists of two clusters with very different numbers of data samples. Also, the data of the larger cluster is more widespread.

Load the data for the example.

```
In[2]:= << unevendensity.dat;
```

Look at the data.

```
In[3]:= NetClassificationPlot[x, y]
```

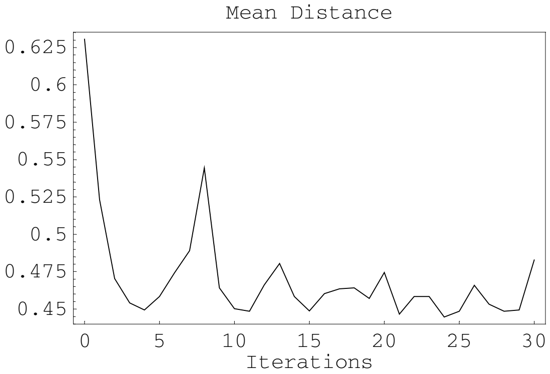


The smaller cluster is situated in the lower-left corner. All its data vectors are close together. The other, larger cluster exhibits a larger spread in its data vectors.

You will see that it is hard to detect a small cluster near a large one.

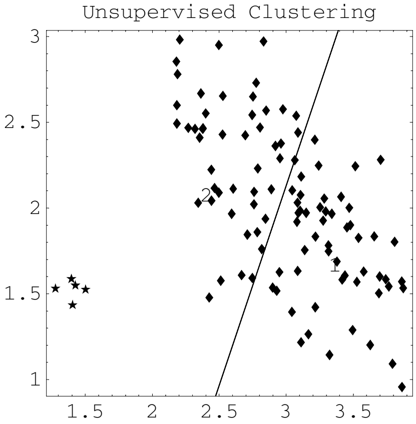
Train an unsupervised network with two codebook vectors.

```
In[4]:= {unsup, fitrecord} = UnsupervisedNetFit[x, 2];
```



Display the trained network.

```
In[5]:= NetPlot[unsup, x, y]
```



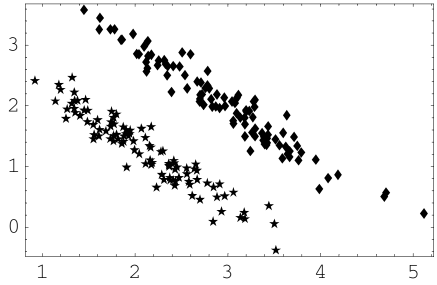
Both codebook vectors have been placed in the large cluster. This can be avoided if you compensate for the skewed data distributions of the clusters. For example, it might be possible to remove some of the data vectors of the larger cluster. It might also be possible to use more codebook vectors so that several of them can be used to explain the large cluster.

### Skewed Data Distributions

Consider now the following case where the data distribution is very skewed.

Load and look at the data.

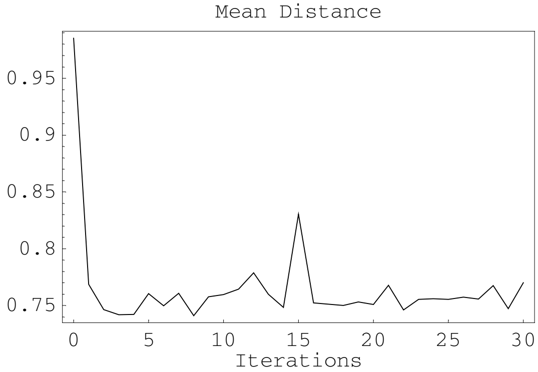
```
In[6]:= << skewed.dat;
        NetClassificationPlot[x, y]
```



The two clusters have very unsymmetrical distributions and there is a clear “direction” in the data.

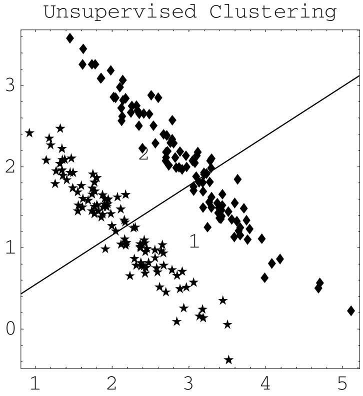
Train an unsupervised network with two codebook vectors.

```
In[8]:= {unsup, fitrecord} = UnsupervisedNetFit[x, 2];
```



Look at the result.

```
In[9]:= NetPlot[unsup, x, y]
```



Instead of one codebook vector appearing in each cluster, they have both converged to points in between the clusters, dividing the clusters in the middle. This is due to the skewed data distribution. If the data is preprocessed with a linear transformation so that the clusters become symmetric, then this problem may be avoided.

## 10.3 Examples with Self-Organizing Maps

If the unsupervised network is supplied with a neighbor feature so that not only the distance between the data vectors and the closest codebook vector is minimized, but also the distance between the codebook vectors, then you have a SOM network—a self-organizing map. SOM networks are often also called *Kohonen* networks.

The aim of a SOM network is to find a mapping from the space of dimension equal to the number of components of the data vectors to a one- or two-dimensional space. The mapping should preserve "closeness" between data vectors; that is, two data vectors that are close to one another in the original space should be mapped to points (codebook vectors) of the new space that are also close to one another. This idea will be illustrated with some examples.

Notice that if you re-evaluate the examples you will not obtain exactly the same results due to the randomness in the initialization. There are several local minima where the training may converge to.

### 10.3.1 Mapping from Two to One Dimensions

If not done already, make the *Neural Networks* package available.

Read in the *Neural Networks* package and the two-dimensional data in six different clusters.

```
In[1]:=  << NeuralNetworks`
```

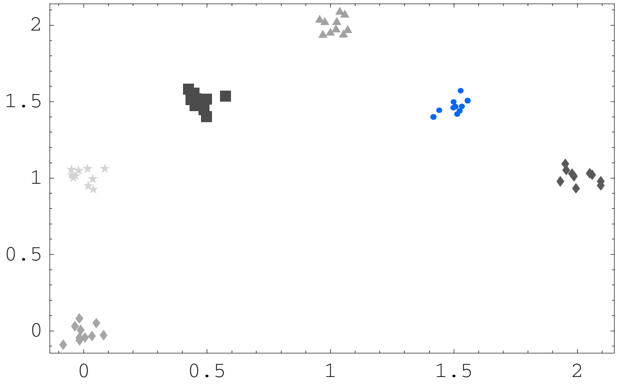
```
In[2]:=  << sixclasses.dat;
```

The data set $x$ is a matrix with one data vector on each row. The number of columns indicates the dimensionality of the input space.

The SOM training algorithm does not use any output data, but it can still be interesting to keep track of the clusters to which the various data vectors belong. This information is stored in $y$, following the standard format of the package as described in Section 3.2, Package Conventions.

Look at the data indicating the different clusters.

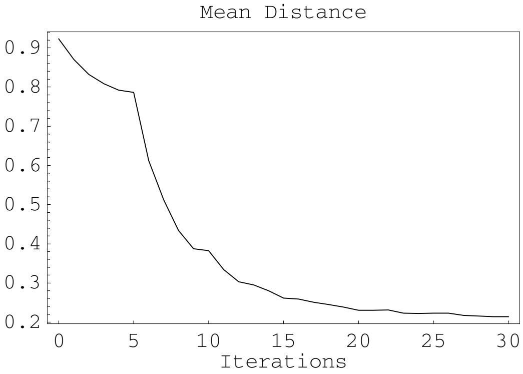
```
In[3]:= NetClassificationPlot[x, y,
         SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



The aim of the SOM network is now to find a one-dimensional relationship among the clusters. Since there are six clusters, at least six neurons are needed. The neighbor structure is indicated with the option SOM, and by setting it to {6, 1} you obtain a one-dimensional structure with six clusters in a column. As opposed to this small example, the true number of clusters is usually unknown. Then you have to experiment with networks of different sizes. Often it can be advantageous to have more codebook vectors than clusters. If you re-evaluate this example with a different number of codebook vectors you will see that there are typically fewer problems with local minima if you add a couple more codebook vectors.

Define and fit a SOM network with six clusters.

```
In[4]:= {somnet, fitrecord} = UnsupervisedNetFit[x, 6, SOM → {6, 1}];
```



The specified neighbor feature is now stored as a rule in the second element of the unsupervised network.

```
In[5]:=  somnet

Out[5]=  UnsupervisedNet[{-Codebook Vectors -}, {CreationDate → {2002, 4, 3, 14, 25, 14},
            Connect → False, SOM → {6, 1}, AccumulatedIterations → 30}]
```

Provide information about the network.

```
In[6]:=  NetInformation[somnet]

Out[6]=  Unsupervised net with a self-organizing map with 6 codebook vectors organized as
            a 6 x 1 array. Takes data vectors with 2 components. Created 2002-4-3 at 14:25.
```

The obtained SOM network can be evaluated directly on new input data vectors. The neurons are related to a structure as indicated by the option `SOM`, and the outputs are the coordinates of the winning neurons within this structure.

Evaluate the SOM network on a new input data vector.

```
In[7]:=  somnet[{0.1, 1.2}]

Out[7]=  {{5, 1}}
```

The two coordinates give the position of the winning neuron within the SOM structure. In this example, only one of the coordinates varies since a one-dimensional SOM structure was chosen. You can also evaluate a SOM network using the option `SOM→False`. Then you obtain the number of the winning codebook vector.

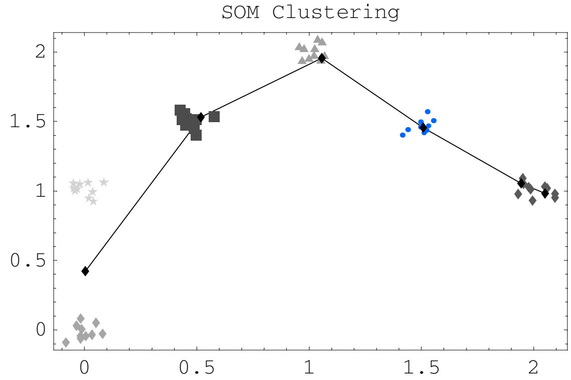
Evaluate the SOM network obtaining the number of the winning codebook vector.

```
In[8]:=  somnet[{0.1, 1.2}, SOM → False]

Out[8]=  {5}
```

Since the data space has two dimensions, the result can be displayed with `NetPlot`.

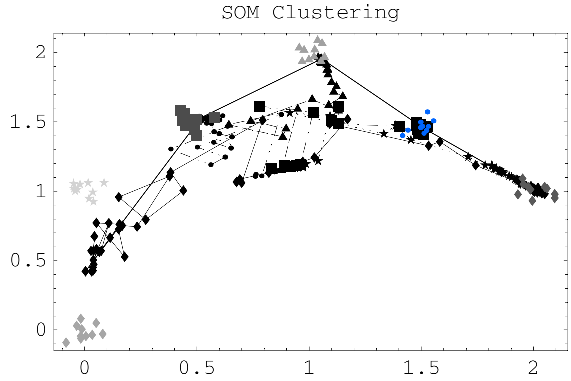
Plot the fitted SOM network.

```
In[9]:= NetPlot[somnet, x, y,
          SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



You can also check how the training process developed. This is done by submitting the training record.

```
In[10]:= NetPlot[fitrecord, x, y,
           SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```

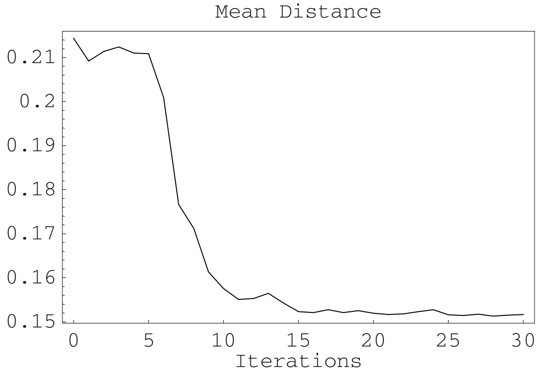


In the plot you can see how the codebook vectors change from the initial position in the center toward their final positions during the training.

If the training has not converged, you can call `UnsupervisedNetFit` once again. It will start with the submitted model so that you do not have to redo the earlier iterations.

Continue the training further.

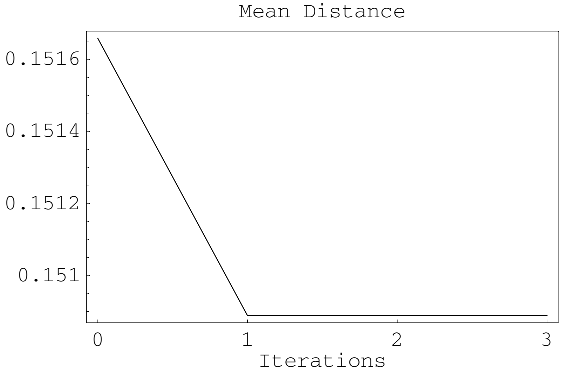
```
In[11]:= {somnet, fitrecord} = UnsupervisedNetFit[x, somnet];
```



If the codebook vectors have gotten fairly close to their minimum, it might be better to apply a few training iterations to the batch algorithm rather than to the recursive one.

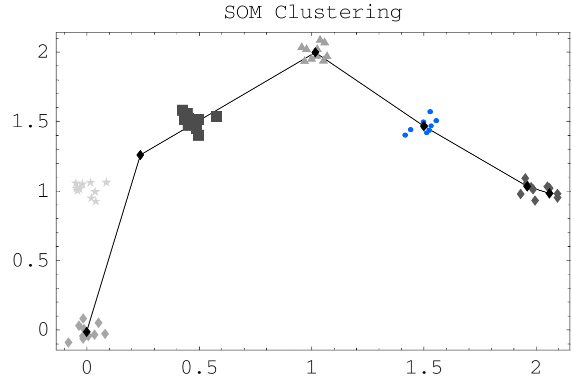
Apply three steps to the batch training algorithm.

```
In[12]:= {somnet, fitrecord} = UnsupervisedNetFit[x, somnet, 3, Recursive → False];
```



Display the trained SOM network.

```
In[13]:= NetPlot[somnet, x, y,
          SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



If `NetPlot` is evaluated with the option `DataMap→Table`, then a table is given that indicates which data vectors are mapped to which codebook vectors. If the dimension of the data space is higher than two, then this is the default (and only) way for `NetPlot` to display the result.

Verify to which codebook vector the different data vectors are mapped.

```
In[14]:= NetPlot[somnet, x, y, DataFormat → Table]
```

SOM Clustering

| | | | | | |
|---|---|---|---|---|---|
| 6:5 | 6:5 | 5:10 | 4:10 | 3:10<br>2:10 | 1:10 |

The table shows that both codebook vectors one and two are placed at the sixth cluster, and that the fifth codebook vector is used for both clusters two and three. This could also be seen from the earlier plots with the data and the codebook vectors. Notice that you will obtain different results if you re-evaluate the example.

The obtained SOM network can be used to map any number of data vectors of the correct dimension. The outputs are the indices of the codebook vector that is closest to the data vector.

Map two vectors with the SOM network.

```
In[15]:= somnet[{{1, 2}, {0, 0}}]

Out[15]= {{4, 1}, {6, 1}}
```

The mean Euclidian distance between a set of data vectors and their nearest codebook vector is given by the command `UnsupervisedNetDistance`.

Determine the mean distance between data and codebook vectors.

```
In[16]:= UnsupervisedNetDistance[somnet, {{1, 2}, {0, 0}}]

Out[16]= 0.0153328
```

This gives a measure of how well the submitted data set is described by the SOM network.

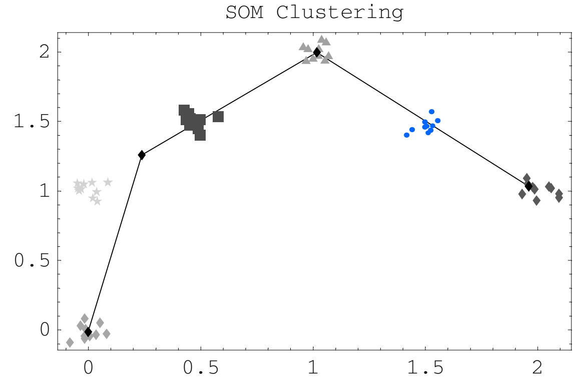
You may remove some of the codebook vectors from the SOM network using NeuronDelete. You have to indicate the rows and the columns to be removed. In this example there is only one column; that is, there is only one codebook vector on each row. Remove the first and the third rows. Notice, however, that dead neurons are not necessarily a nuisance for a SOM network. In the training, the positions of the codebook vectors are determined by both the closeness to the data and closeness to each other. Therefore, a codebook vector which is not used by any data can still form a bridge between two clusters. You can see this by re-evaluating this example with SOM networks and more codebook vectors.

Remove the first and the third codebook vectors.

```
In[17]:= somnet = NeuronDelete[somnet, {{1, 3}, {}}]

Out[17]= UnsupervisedNet[{-Codebook Vectors -}, {CreationDate → {2002, 4, 3, 14, 25, 16},
           Connect → False, SOM → {4, 1}, AccumulatedIterations → 63}]
```

Look at the clustering of the modified network.

```
In[18]:= NetPlot[somnet, x, y,
          SymbolStyle → {Hue[.1], Hue[.2], Hue[.0], Hue[.4], Hue[.6], Hue[.8]}]
```



## 10.3.2 Mapping from Two Dimensions to a Ring

A one-dimensional SOM network can be connected into a ring. This might be favorable in some cases where you might be trying to find a one-dimensional cyclic behavior in the data in a high-dimensional space. For clarity, this is demonstrated in a two-dimensional space.

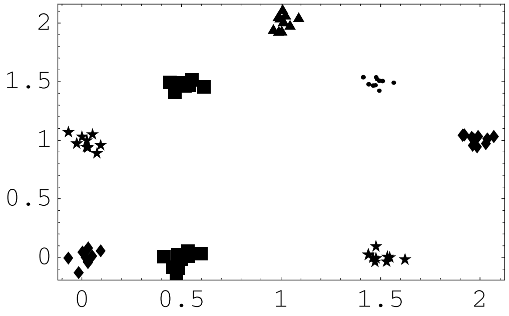
Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << classesinring.dat;
```

Look at the data.

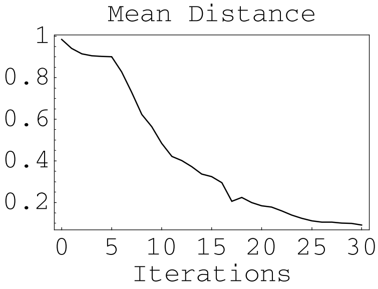
```
In[3]:= NetClassificationPlot[x, y]
```



From the data plot you can see that the data consists of eight clusters, and they can be connected into a ring. A SOM network will be used to find this relation. By giving the option `Connect→True` when the network is initialized, you indicate that the codebook vectors should be connected into a ring.

Initialize and train a SOM network with eight codebook vectors that are connected into a ring.

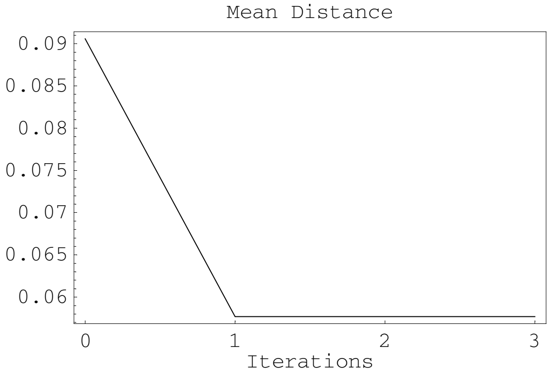
```
In[4]:= {somnet, fitrecord} =
          UnsupervisedNetFit[x, 8, 30, SOM → {8, 1}, ReportFrequency → 5, Connect → True];
```



A few steps with the batch training is recommended if the average distance has not converged in the recursive training.

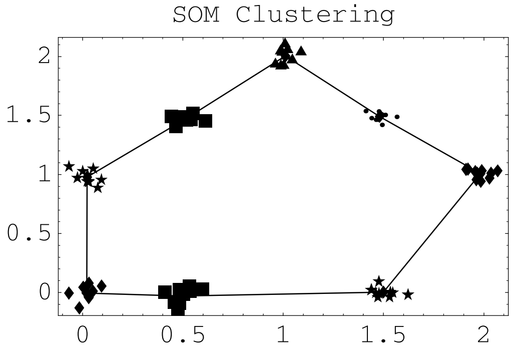
Apply three training iterations in batch mode.

```
In[5]:= {somnet, fitrecord} = UnsupervisedNetFit[x, somnet, 3, Recursive → False];
```



Plot the SOM network and the data.

```
In[6]:= NetPlot[somnet, x, y]
```



Display the result in a table.

```
In[7]:= NetPlot[somnet, x, y, DataFormat → Table]
```

SOM Clustering

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5:10 | 4:10 | 3:10 | 2:10 | 1:10 | 8:10 | 7:10 | 6:10 |

Ideally there should be only one data cluster at each codebook vector, but often the algorithm is caught in a local minimum, and then there might be several data clusters mapped on a common codebook vector. Notice that you will obtain a different result if you repeat the example, due to the randomness in the algorithms.

It is also possible to use more codebook vectors than you have data clusters. Then there will be several codebook vectors describing some of the clusters. You can go back and change the number of codebook vectors and repeat the example.

This example considered a one-dimensional SOM mapping. Two-dimensional SOM networks can also be given the option `Connect`. They are then connected only in the first of the two dimensions, and a topological cylinder is formed.

### 10.3.3 Adding a SOM to an Existing Unsupervised Network

It is possible to change the neighbor structure of an already-existing unsupervised network. The example demonstrates how this is done.

Read in the *Neural Networks* package and load a data set.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << sixclasses.dat;
```

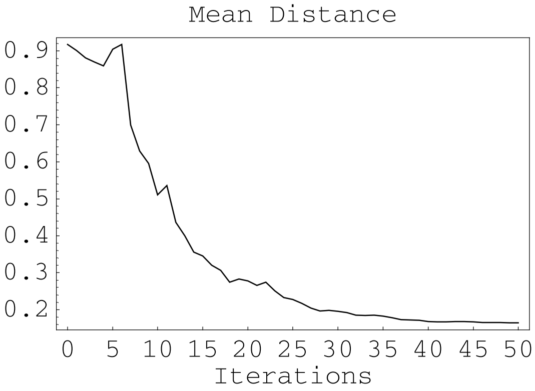
Initialize an unsupervised network without any neighbor structure.

```
In[3]:= unsup = InitializeUnsupervisedNet[x, 6]

Out[3]= UnsupervisedNet[{-Codebook Vectors -}, {CreationDate → {2002, 4, 3, 14, 29, 19},
          AccumulatedIterations → 0, SOM → None, Connect → False}]
```

The neighbor structure is now specified in the call to `UnsupervisedNetFit` using the options `SOM` and `Connect`.

Train the unsupervised network and specify the neighbor structure.

```
In[4]:= {somnet, fitrecord} = UnsupervisedNetFit[x, unsup, 50, SOM → {6, 1}, Connect → True];
```



```
In[5]:= somnet

Out[5]= UnsupervisedNet[{-Codebook Vectors-}, {CreationDate → {2002, 4, 3, 14, 29, 24},
          Connect → True, SOM → {6, 1}, AccumulatedIterations → 50}]
```

The neighbor structure is now stored in the network with the replacement rule `SOM→{ }` .

### 10.3.4 Mapping from Two to Two Dimensions

In this example a SOM network is used to quantize a two-dimensional space. The data space contains two dimensions and a two-dimensional SOM network is used. Hence, there is no reduction in the dimensionality of the data, but the mapping could be used to quantize the information in the original data. In addition to the quantization, there is also the neighbor effect of the SOM network.

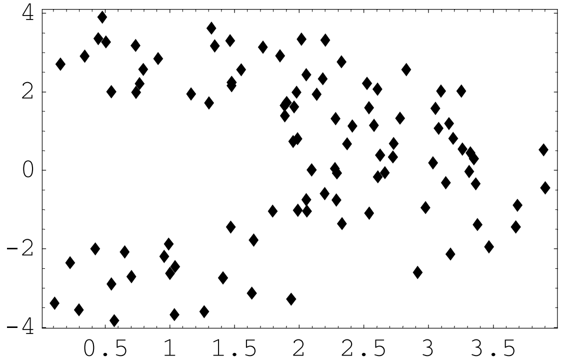
Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`

In[2]:= << two2twosom.dat;
```

The data vectors are placed in matrix *x* with one input vector on each row.

Look at the data.

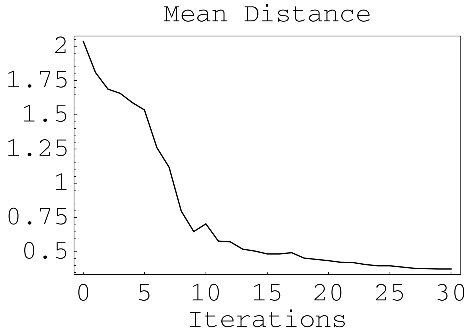
```
In[3]:= NetClassificationPlot[x]
```



The data vectors are unevenly distributed as shown in the plot. The training of the SOM network will place more codebook vectors in regions where data is most concentrated.

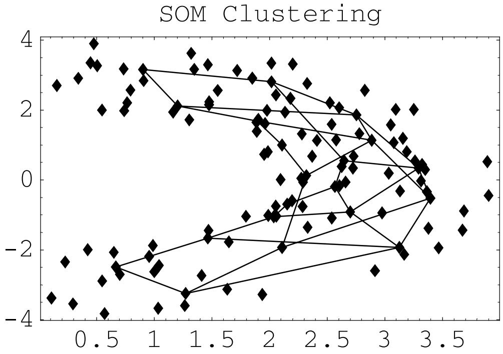
Define and train a SOM network with 5 × 4 codebook vectors.

```
In[4]:= {somnet, fitrecord} = UnsupervisedNetFit[x, 20, 30, SOM → {5, 4}, ReportFrequency → 5];
```



Plot the trained SOM network.

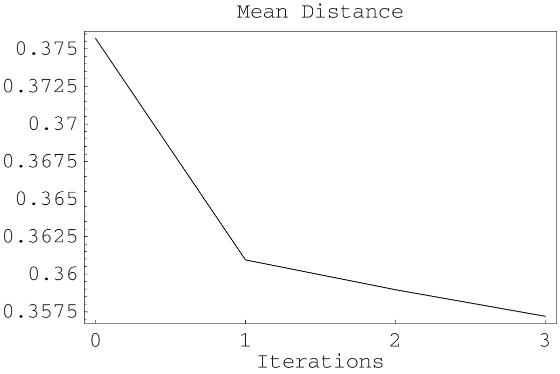
```
In[5]:= NetPlot[somnet, x]
```



When the codebook vectors have gotten fairly close to their minima, it is often advantageous to apply a few training iterations with the batch algorithm.

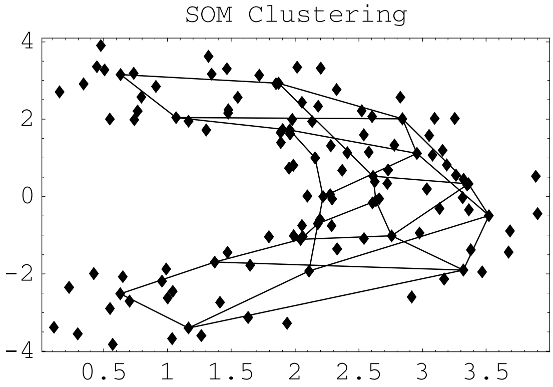
Perform three additional iterations with the batch training algorithm.

```
In[6]:= {somnet, fitrecord} =
          UnsupervisedNetFit[x, somnet, 3, ReportFrequency → 1, Recursive → False];
```



Plot the final SOM network.

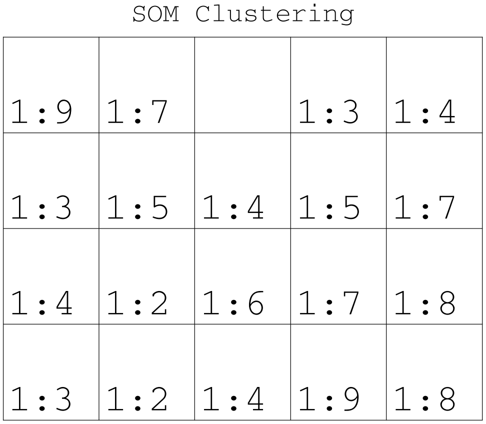
```
In[7]:= NetPlot[somnet, x]
```



As illustrated in the plot, the codebook vectors are placed closer where data concentrations are higher.

If `NetPlot` is called with `DataMap→Table`, you can see how many data vectors are mapped to each of the codebook vectors.

```
In[8]:= NetPlot[somnet, x, DataFormat → Table]
```



SOM Clustering

| | | | | |
|---|---|---|---|---|
| 1:9 | 1:7 | | 1:3 | 1:4 |
| 1:3 | 1:5 | 1:4 | 1:5 | 1:7 |
| 1:4 | 1:2 | 1:6 | 1:7 | 1:8 |
| 1:3 | 1:2 | 1:4 | 1:9 | 1:8 |

### 10.3.5 Mapping from Three to One Dimensions

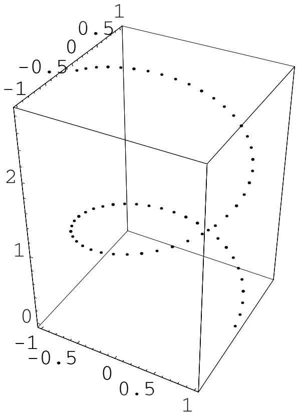
This example illustrates how a SOM network can be used to find a one-dimensional relationship in a three-dimensional space. First, data samples are generated along a curve. You can modify the example by making changes in the data generating commands and rerun the commands.

Load the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

Generate data along a curve.

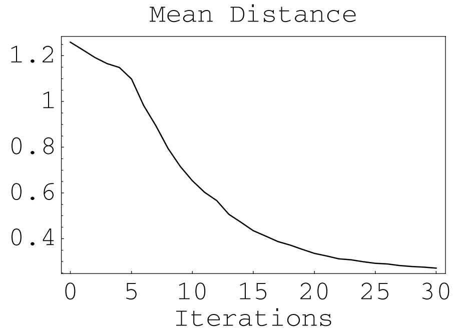
```
In[2]:= << Graphics`Graphics3D`;
        x = N[Table[{ Cos[t],  Sin[t],  0.3 t},
             {t,  0,  3 Pi,  Pi / 20}]];
        xgraphics = ScatterPlot3D[x, AxesEdge → {{-1, -1}, Automatic, Automatic}]
```



A knot on the chain of codebook vectors may occur, but the risk of this is typically smaller when the dimension of the data space is larger than two. A nonsymmetric neighbor function, which is the default, reduces the risk for knots. Here, because the data space is of dimension three, the option `Neighbor→Symmetric` is chosen.

Train a SOM network with ten codebook vectors.

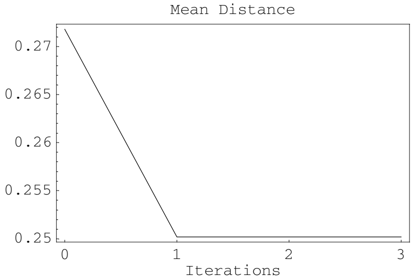
```
In[5]:= {somnet, fitrecord} = UnsupervisedNetFit[x, 10, SOM → {10, 1}, Neighbor → Symmetric];
```



If the displayed average distance has not converged, it could be worth continuing with additional training iterations. Doing this with the batch training algorithm is often advantageous when you are close to the minimum.

Perform three training iterations with the batch training algorithm.

```
In[6]:= {somnet, fitrecord} = UnsupervisedNetFit[x, somnet, 3, Recursive → False];
```

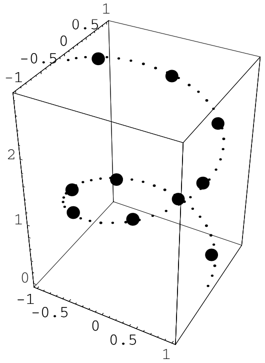


```
In[7]:= somnet

Out[7]= UnsupervisedNet[{-Codebook Vectors -}, {CreationDate → {2002, 4, 3, 14, 30, 39},
          Connect → False, SOM → {10, 1}, AccumulatedIterations → 33}]
```

The codebook vectors are stored in the first element of the SOM object. They can easily be extracted and plotted.

Extract and plot codebook vectors together with the original data vectors.

```
In[8]:= cbv = somnet[[1]];
        cbvgraphics =
          ScatterPlot3D[cbv, PlotStyle → AbsolutePointSize[8], DisplayFunction → Identity];
        Show[cbvgraphics, xgraphics, DisplayFunction → $DisplayFunction,
         AxesEdge → {{-1, -1}, Automatic, Automatic}]
```



## 10.3.6 Mapping from Three to Two Dimensions

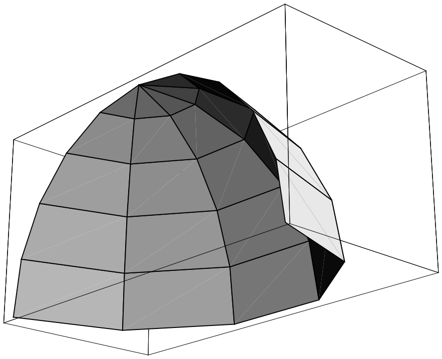
Load the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

This example illustrates how a SOM network can be used to find and model a surface in a three-dimensional space.

Generate data and look at the surface spanned by the data vectors.

```
In[2]:= << Graphics`Graphics3D`;
        apts = N[Table[{Cos[t] Cos[u], Sin[t] Cos[u],
         Sin[u]}, {t, 0, Pi, Pi / 5}, {u, Pi / 2, Pi, Pi / 10}]];
        xgraphics = ListSurfacePlot3D[apts, ViewPoint -> {-2.325, 2.146, -1.200}]
        x = Flatten[apts, 1];
```



There is one data vector at each vertex in the plot.

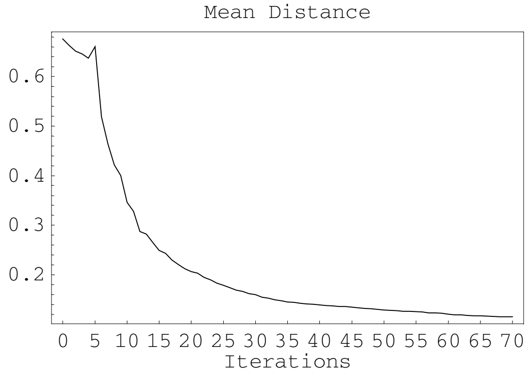
Look at the dimensionality of the data vectors.

```
In[6]:= Dimensions[x]

Out[6]= {36, 3}
```

There are 36 data vectors in a three-dimensional space.

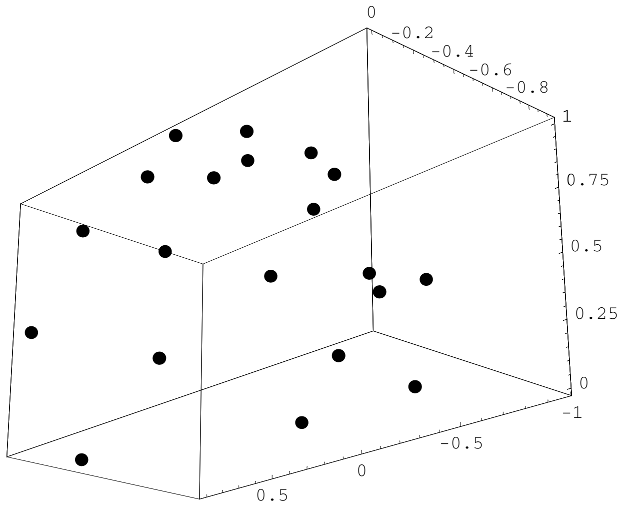
Initialize and fit a SOM network with 4 × 5 codebook vectors.

```
In[7]:= {somnet, fitrecord} = UnsupervisedNetFit[x, 20,
           70, SOM → {4, 5}, ReportFrequency → 5, Neighbor → Symmetric];
```



Extract the codebook vectors, which are stored in the first element of the SOM network, and plot them.

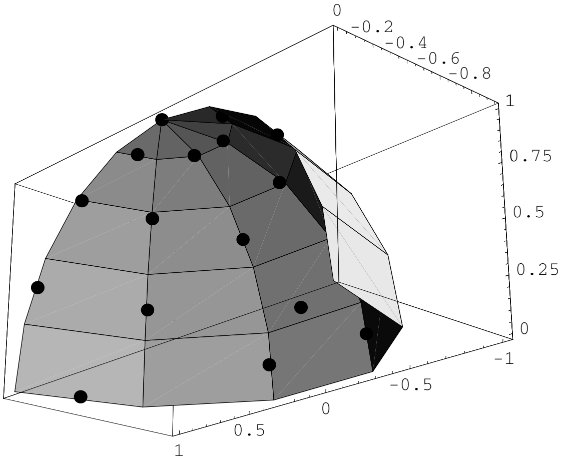
```
In[8]:= cbv = somnet[[1]];
        cbvgraphics = ScatterPlot3D[cbv, PlotStyle → AbsolutePointSize[7],
          ViewPoint -> {-2.325, 2.146, -1.200}, AxesEdge → {{-1, -1}, Automatic, Automatic}]
```



This plot does not show very much. It becomes a bit clearer if the codebook vectors are plotted together with the original surface representing the data vectors.

Plot the codebook vectors together with the surface.

```
In[10]:= Show[cbvgraphics, xgraphics]
```



## 10.4 Change Step Length and Neighbor Influence

There is a default function guiding the step length in the training of unsupervised networks, and another function for the neighbor strength. These defaults were described in connection with the function UnsupervisedNetFit, and they have been chosen to suffice for a large variety of different problems. Both have been chosen so that the step length and the neighbor strength become smaller when the number of training iterations increase. Sometimes, however, depending on the data vectors and the chosen size of the unsupervised network, it might be advantageous to apply other functions than the defaults. The easiest way to modify the step length and neighbor strength is to submit numerical values in the two options `StepLength` and `NeighborStrength`. That gives constant values to these parameters, and that might suffice in many cases. The following describes more advanced changes involving functions for these options.

Read in the *Neural Networks* packages and load a data set.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << two2twosom.dat;
```

The step length can be chosen to be any function that takes an integer indicating the number of the training iteration and delivers a step size in the interval {0, 1}.

Define a new step length function.

```
In[3]:= step = Function[n, 1 / (5. + n)]
```

```
                      1
Out[3]= Function[n, ------]
                    5. + n
```

The neighbor strength function and the neighbor matrix together dictate how much the neighbors of the winning codebook vector are changed. With a strong neighbor strength, the codebook vectors will be kept close to one another, while loose coupling will let them adapt more independently of each other. The neighbor strength function enters the algorithm as a negative exponent, so that a larger value indicates a weaker neighbor coupling (see Section 10.1.2, UnsupervisedNetFit). The function takes the training iteration number as an input argument and returns a positive real value. Typically, the neighbor strength should have a low value at the beginning of the training so that the codebook vectors obtain the correct orientation with respect to one another. Then, as the training progresses, the coupling strength between the codebook vectors should become weaker, which can be achieved if the neighbor strength function returns a larger value.

Here, a neighbor strength function is defined to return twice the iteration number. This yields a much faster decrease in the coupling between the neighbors than the default.

Define a new function guiding the neighbor strength.

```
In[4]:= neighborstrength = Function[n, n * 2.]
```

```
Out[4]= Function[n, n 2.]
```

With the neighbor option you can specify the codebook vectors, or neurons, that are close to each other. Using the default option `NonSymmetric`, or the other prespecified alternative `Symmetric`, makes `UnsupervisedNetFit` create a neighbor matrix automatically with the correct dimensions. A user-specified neighbor matrix should have $2c - 1$ components in each dimension, where $c$ is the number of codebook vectors in the corresponding dimension. The center element, which corresponds to the winning codebook vectors, should be zero. Then the winning codebook vector will be updated with the step length given by the step length function. The other codebook vectors will also be updated, and their change is influenced by the neighbor strength function; if the neighbor strength function gives larger values, then the neighbors are changed less, as described in Section 10.1.2, UnsupervisedNetFit.

Specify a neighbor matrix for a SOM network with three codebook vectors in one direction and four in the other. This means that the network has 12 codebook vectors. The elements are chosen to become larger in all directions from the winning codebook vectors, which means that distant codebook vectors are influenced less than those that are close to the winner.

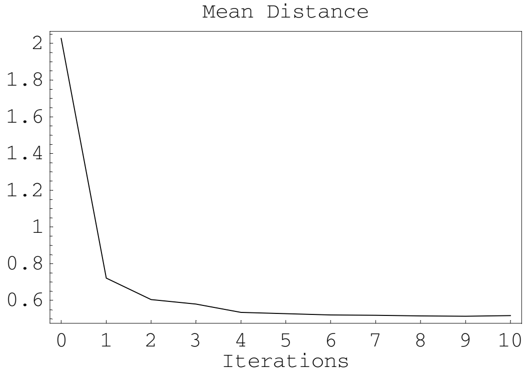
Define a neighbor matrix for a SOM network with 3 × 4 codebook vectors.

```
                                 ⎛10  8  6  4  5  6  7⎞
                                 ⎜ 8  6  4  2  3  4  5⎟
In[5]:=  neighborMatrix =        ⎜ 6  4  2  0  1  2  3⎟;
                                 ⎜ 7  5  3  1  2  3  4⎟
                                 ⎝ 8  6  4  2  3  4  5⎠
```

The new choices of step length, neighbor strength, and neighbor matrix can now be submitted as options to `UnsupervisedNetFit`.

Train the SOM network with the modified algorithm.

```
In[6]:=  {somnet, fitrecord} = UnsupervisedNetFit[x, 12, 10, SOM → {3, 4}, StepLength → step,
            NeighborStrength → neighborstrength, Neighbor → neighborMatrix];
```



The trained SOM network can be evaluated in the same way as shown earlier in the other examples.

## 10.5 Further Reading

There are many general books on neural networks, and most of them cover unsupervised methods. Here are some suggestions:

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, Macmillan, 1999.

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Reading, MA, Addison-Wesley, 1991.

T. Kohonen, *Self-Organizing Maps*, Berlin, Germany, Springer-Verlag, 1995.

# 11 Vector Quantization

Vector quantization networks are intended to be used for classification. Like unsupervised networks, the VQ network is based on a set of *codebook vectors*. Each class has a subset of the codebook vectors associated to it, and a *data vector* is assigned to the class to which the closest codebook vector belongs. In the neural network literature, the codebook vectors are often called the *neurons* of the VQ network. In contrast to unsupervised nets, VQ networks are trained with supervised training algorithms. This means that you need to supply output data indicating the correct class of any particular input vector during the training.

Section 2.8, Unsupervised and Vector Quantization Networks, gives a short tutorial on VQ networks. Section 11.1, Vector Quantization Network Functions and Options, describes the functions, and their options and examples are given in Section 11.2, Examples. Section 11.3, Change Step Length, describes how you can change the training algorithm by changing the step length.

## 11.1 Vector Quantization Network Functions and Options

This section introduces the commands to deal with VQ networks. Examples of the use of the commands follow in Sections 11.2, Examples.

### 11.1.1 InitializeVQ

VQ networks are initialized with `InitializeVQ`. The initialization algorithm can be influenced by a number of options.

`InitializeVQ` is called in the following way.

| | |
|---|---|
| `InitializeVQ[`*x*`,`*y*`,`*size*`]` | initializes a VQ net of indicated size using input data *x* and output data *y* |

Initializing a vector quantization network.

VQ networks are stored in objects with head `VQ`, on a format following the general standard of the package, as described in Section 3.2, Package Conventions. The first component is a list of the codebook vectors.

The supplied data matrices *x* and *y* should have the format described in Section 3.2, Package Conventions. The parameter *size* should be an integer or a list of integers with one component for each class. It specifies the number of codebook vectors in the following way:

- The integer indicates the total number of codebook vectors of the VQ network, which are distributed uniformly over the classes. The first classes receive one vector more than the others if the number is not a multiple of the number of classes.
- The list of integers indicates the number of codebook vectors for each class. The total number of codebook vectors is given by the sum of the components in this list.

A VQ network can be initialized in two different ways. The default is a random initialization, which sets all codebook vectors to random values around the mean value of the data vectors in *x*. In most cases it is better to initialize by using an unsupervised network for each class of the data. This is done by setting the option `UseUnsupervisedNet→True`, and most of the options of `InitializeVQ` are used to control the training of the unsupervised networks. Since an unsupervised network can be initialized using a SOM neighbor map, it could also be interesting to change the behavior of the SOM training by passing on some options to `InitializeUnsupervisedNet`. This can be done using the option `SOMOptions`, where you can submit the options in a list. More details about the initialization may be found in Section 10.1.2, UnsupervisedNetFit.

`InitializeVQ` takes the following options. Notice that most of the options control the initial competitive training with an unsupervised network. Therefore, they remain inactive if you do not set `UseUnsupervisedNet→True`.

| *option* | *default value* | |
|---|---|---|
| `UseUnsupervisedNet` | `False` | if `True`,<br>initializes the VQ net using an unsupervised net;<br>otherwise, random initialization is true |
| `Compiled` | `True` | applies the unsupervised training compiled |
| `Iterations` | `6` | number of iterations with the unsupervised net |
| `Recursive` | `False` | applies the competitive training in batch mode |
| `InitialRange` | `0.01` | initial range of the parameters<br>if the net is randomly initialized |
| `StepLength` | `Automatic` | step length function for the competitive training |
| `SOMOptions` | `{}` | list of options to be used in<br>the initial training with the SOM net |
| `CriterionPlot` | `False` | gives no plot to present the result of the<br>training with the competitive algorithm |
| `CriterionLog` | `False` | logs no information about the<br>training with the competitive algorithm |
| `CriterionLogExtN` | `True` | if the `CriterionLog` option is set to `True`,<br>then presents the training<br>log in a separate notebook |
| `ReportFrequency` | `2` | if the `CriterionLog` option is set to `True`,<br>then the performance logs with<br>this frequency during the training |
| `MoreTrainingPrompt` | `False` | prompts for more training iterations if set to `True` |

Options of `InitializeVQ`.

Some of the options are set differently from those of `InitializeUnsupervisedNet` when used directly. For example, using the default, no training results are reported. The default value of the `StepLength` option is `Function[If[#<3, 0.1, 0.5]]`.

A random initialization is much faster than using the initialization based on unsupervised networks. However, it is often worth the effort to use the unsupervised initialization since it will usually reduce the training required when VQFit is subsequently applied. This is illustrated in Section 11.2.1, VQ in Two-Dimensional Space.

You can also include any set of vectors of appropriate dimension as initial codebook vectors in an existing VQ network. In this way you can start the training at any point you wish, or write your own training algorithm. How this is done is described in Section 13.1, Change the Parameter Values of an Existing Network.

The default of the option StepLength depends on whether the training is performed recursively or in batch. The defaults are given in the following table.

| | |
|---|---|
| 1 | if Recursive→False |
| Function[If[#<5, 0.01, 2./(3+#)]] | if Recursive→True |

Default values of the StepLength option.

The step length can be modified with the StepLength option by submitting a numerical constant or a function of choice. This is done the same way as for unsupervised networks and is also illustrated in Section 11.3, Change Step Length.

### 11.1.2 VQFit

An initialized VQ network can be trained with VQFit. It is also possible to start directly with VQFit without an already initialized VQ network, but then you cannot influence the initialization with any options.

| | |
|---|---|
| VQFit[*x*,*y*, *size*] | initializes and trains a VQ network of indicated size the default number of iterations, which is six |
| VQFit[*x*,*y*, *size*, *iterations*] | initializes and trains a VQ network of indicated size a specified number of iterations |
| VQFit[*x*,*y*, *net*] | trains a supplied VQ network the default number of iterations, which is six |
| VQFit[*x*,*y*, *net*, *iterations*] | trains a supplied VQ network a specified number of iterations |

Training a vector quantization network.

An existing network can be submitted for more training by setting *net* equal to the network or its training record. The advantage of submitting the training record is that the information about the earlier training is combined with the additional training.

The input data vectors are indicated by $x$ and the output by $y$.

`VQFit` returns a list of two variables. The first variable is the trained vector quantizer and the second variable is a training record. The training record, with head `VQRecord`, can be used to analyze the progress of the training and for validation, using the command `NetPlot`. You can also extract intermediate information about the training from the training record, as described in Section 7.8, The Training Record.

Intermediate results during the training are displayed in a separate notebook, which is created automatically. After each training iteration, the fraction of misclassified data is written out. The value 1 means that all data vectors are incorrectly classified, and 0 means that they are all correctly classified. Using the options of `VQFit`, as described in Section 7.7, Options Controlling Training Results Presentation, you can change the way the training results are presented.

If the number of training iterations is not supplied, the VQ learning algorithm will terminate in six iterations. The necessary number of training iterations can vary substantially from one problem to another. If you did not apply enough iterations, you can submit the VQ network to `VQFit` a second time to train further. Consequently, you do not have to start from the beginning.

At the end of the training, the fraction of misclassified data versus the training iteration is displayed in a plot, assuming this facility is not switched off with the option `CriterionPlot`.

A derived VQ network can be evaluated on data using the function evaluation rule. The output will indicate the class to which the input is classified with a 1 in the corresponding column.

| | |
|---|---|
| *net* [ *x* ] | evaluates *net* on the input vector *x* |

Function evaluation of a vector quantization network.

The input argument $x$ can be a vector containing one input sample or a matrix containing one input sample on each row.

VQ networks are stored in a format following the general standard of the package, as described in Section 3.2, Package Conventions. The first component contains a list of the codebook vectors, which you may change directly as described in Section 13.1, Change the Parameter Values of an Existing Network.

`VQFit` takes the following options.

| *option* | *default value* | |
|---|---|---|
| `StepLength` | `Automatic` | positive numerical value or a function, which determines the adaptation step length as a function of the number of iterations |
| `Recursive` | `True` | recursive or batch training |
| `Method` | `LVQ1` | training algorithm; alternative: `Competitive` |
| `Compiled` | `True` | uses the compiled version of the training algorithm |
| `CriterionPlot` | `True` | displays the fraction of misclassified data versus the number of iterations in a plot after the training is finished |
| `CriterionLog` | `True` | logs intermediate results during training |
| `CriterionLogExtN` | `True` | displays the fraction of misclassified data in a separate notebook |
| `ReportFrequency` | `1` | report interval, in training iterations, of the intermediate results logged and displayed during training |
| `MoreTrainingPrompt` | `False` | prompts for more training iterations if set to `True` |

Options of `VQFit`.

The options `CriterionPlot`, `CriterionLog`, `CriterionLogExtN`, `ReportFrequency`, and `MoreTrainingPrompt` are common in the other training commands in the *Neural Networks* package, and they are described in Section 7.7, Options Controlling Training Results Presentation.

There are two possible training algorithms for VQ networks, and the choice is made with the option `Method`. Both training algorithms can be applied recursively or in batch mode. The batch version just adds up the contribution over the whole data set before the codebook vectors are changed. The default training algorithm is the *Learning Vector Quantization 1* (LVQ1) by Kohonen.

LVQ1 is the recursive version of the algorithm and is described by the following steps:

1. Begin with a given $N$ input-output data vectors $\{x_k, y_k\}$, $k = 1, \ldots, N$, in each update.
2. $k$ is chosen randomly from a uniform integer distribution between 1 and $N$, inclusively.
3. The codebook vector closest to $x_k$, called the winning neuron, or the winning codebook vector, is identified. Its indices are indicated by $\{i, j\}$, where $j$ is the class it belongs to and $i$ is the number of the codebook vector within this class.
4. The winning codebook vector is then changed in different ways depending on whether or not the winning codebook vector is from the correct class according to $y_k$. If $x_k$ is correctly classified, then the winning codebook is changed as

   $$w_{i,j} = w_{i,j} + SL[n] * (x_k - w_{i,j}) \qquad (1)$$

   otherwise the winning codebook vector is changed according to

   $$w_{i,j} = w_{i,j} - SL[n] * (x_k - w_{i,j}) \qquad (2)$$

   where $n$ is the iteration number.
5. The described steps are repeated $N$ times in each iteration.

Abbreviations have been used; SL[n] is the StepLength function, which may be changed by the option of the same name.

With Recursive→False, all data is used in each iteration, and the training follows a deterministic scheme where the mean value of the update given by Equation 11.1 and Equation 11.2 over all data $\{x_k, y_k\}$, $k = 1, \ldots N$ is used.

The second possible training algorithm, Competitive, is the same training algorithm used for unsupervised networks, but applied to one class at the time. The data is divided into the different classes according to the information in the submitted output data, and in each training iteration the codebook vectors of each class are updated using its data with the same algorithms as UnsupervisedNetFit.

### 11.1.3 NetInformation

Some information about a VQ network is presented in a string by the function NetInformation.

| | |
|---|---|
| `NetInformation[`*vq*`]` | gives information about a VQ net |

The `NetInformation` function.

### 11.1.4 VQDistance, VQPerformance, UnUsedNeurons, and NeuronDelete

The three following functions are used to test and validate VQ networks with respect to different aspects.

The function `VQDistance` gives the average Euclidian distance between the data vectors and the nearest codebook vector of the supplied VQ network. There is no test when the data vectors are correctly classified.

| | |
|---|---|
| `VQDistance[`*vq*`,`*x*`]` | mean Euclidian distance between data *x* and the nearest codebook vector |

Average distance to the nearest codebook vector for a vector quantization network.

`VQDistance` has one option.

| *option* | *default value* | |
|---|---|---|
| `Compiled` | `True` | uses compiled code |

Option of `VQDistance`.

If `VQDistance` returns a small value, all the data vectors are close to a codebook vector, in which case the quantization can be considered successful.

The function `VQPerformance` tests the classification performance of the VQ network. The output is the fraction of misclassified data vectors. Therefore, 0 means that all vectors are correctly classified.

| | |
|---|---|
| `VQPerformance[`*vq*`,`*x*`,`*y*`]` | gives the fraction of data incorrectly classified by the VQ net |

Fraction of misclassified data vectors.

The input data vectors are indicated by $x$ and the output by $y$.

Sometimes there might be codebook vectors that are never used; that is, no data vectors are mapped to them. These neurons are called *dead neurons*, or *dead codebook* vectors. Since these codebook vectors are not used by the data, they make the network more complicated than it has to be. It might be of interest to remove them using `NeuronDelete`. The following command points out the dead neurons.

| | |
|---|---|
| `UnUsedNeurons[`*vq*`,`*x*`]` | gives the numbers of the unused codebook vectors for each class |

Find the unused codebook vectors.

The output is a list with one sublist for each class including the numbers of the unused neurons.

`NeuronDelete` is used to remove codebook vectors from an existing VQ network.

| | |
|---|---|
| `NeuronDelete[`*net*, {*m*, *n*}`]` | deletes codebook vectors *n* of class *m* |

Delete codebook vectors in an existing vector quantization network.

The second argument can also contain a list of codebook vectors to be removed.

### 11.1.5 NetPlot

The function `NetPlot` can be used to illustrate a VQ network and its training. Depending on the value of option `DataFormat`, the result is displayed in different ways. If the argument *net* is chosen to be a trained VQ network, you get an illustration of the network. If it is chosen to be a training record, the second output argument of `VQFit`, then an illustration of the training is obtained.

| | |
|---|---|
| `NetPlot[`*net*`,`*x*`,`*y*`]` | information about a VQ net and how it classifies data of given classes |
| `NetPlot[`*net*`,`*x*`]` | information about a VQ net and how it classifies data |

Illustrate a vector quantization network or its training.

The input data vectors are indicated by $x$ and the output by $y$.

For two-dimensional data vectors, and with the default options, `NetPlot` plots the data, indicates the positions of the codebook vectors with a number corresponding to the number of the class, and plots the Voronoi cells of the codebook vectors.

NetPlot takes the following options.

| *option* | *default value* | |
|---|---|---|
| DataFormat | Automatic | indicates how the data is to be illustrated; default depends on the dimension of the data, with different possibilities described in the following |
| Voronoi | True | displays Voronoi cells |
| CbvSymbol | Automatic | changes the symbol indicating the codebook vectors |
| Intervals | 5 | interval of training iterations between plots in the graphics array of NetPlot |
| Compiled | True | uses compiled version |

Options of NetPlot.

Depending on how the option DataFormat is set, one of the functions MultipleListPlot, BarChart, or BarChart3D is used. Any options of these functions can be given in the call to PlotVQ, and they are then passed on to the plot command.

The option DataFormat takes the following values.

| | |
|---|---|
| DataMap | as the default for two-dimensional problems, gives a plot of the data together with the codebook vectors |
| BarChart | illustrates the classification result with a bar chart |
| Table | gives a table with one box for each class; in each box indicates the number of data vectors from each class classified to this class |
| ClassPerformance | as the default for training records when the input dimension is larger than two, plots the classification performance versus training iterations |
| DataMapArray | gives a graphics array of the progress of the clustering during training; applies only to two-dimensional problems |

Possible values of DataFormat.

The two last possibilities only apply when `NetPlot` is applied to a training record.

If the dimension of the codebook vectors is higher than two, then the default way to illustrate the classification is to use a bar chart.

A three-dimensional bar chart plots the VQ-derived classifications of the input data against their actual classifications known from the original given output data. If the two match perfectly, the chart consists of three-dimensional bars along a diagonal, the heights of which are the class populations. Misclassifications show up as nondiagonal columns.

If no output is supplied, the result is given in a two-dimensional bar chart. It illustrates the distribution of the classification of the supplied data.

If the training record, the second output argument from `VQFit`, is submitted instead of the VQ network, then the improvements of the VQ network during the training is illustrated using the logged intermediate results at the frequency given by the `ReportFrequency` option of `VQFit`.

By supplying the training record and by choosing `DataFormat→ClassPerformance`, the classification result for each class can be plotted versus the number of training iterations. The other possible values of `DataFormat` give graphics arrays of the corresponding result when `NetPlot` is applied to a VQ network. Each plot in the array illustrates the result at a certain stage of the training.

If `Voronoi→True`, then the default of `CbvSymbol` is to indicate each codebook vector with its number in the network. You can change this by submitting a list of symbols. You can also use this option to include options of `Text`, which then modifies only the codebook's marks, and not the plot label and the axes.

Instead of displaying the Voronoi cells, you can have a cross at the position of each codebook vector. This is done by choosing `Voronoi→False`. The size of the crosses is automatically set, but you can change it by setting the option `CbvSymbol` to a list of numbers indicating the desired size.

## 11.2 Examples

In this section VQ networks are used for classification in some small examples. The first two examples illustrate the commands. Then, a series of examples follows, illustrating some possible problems you may encounter using VQ nets. The last example illustrates the different initialization algorithms.

It is always easier to illustrate the data and the classifier in two-dimensional problems; so, most of the examples are constrained to that dimensionality for didactic reasons.

### 11.2.1 VQ in Two-Dimensional Space

Load the *Neural Networks* package.

```
In[1]:=  << NeuralNetworks`
```

The file vqthreeclasses.dat contains two-dimensional data divided into three classes, consisting of two clusters.

Load the data vectors and output indicating class.

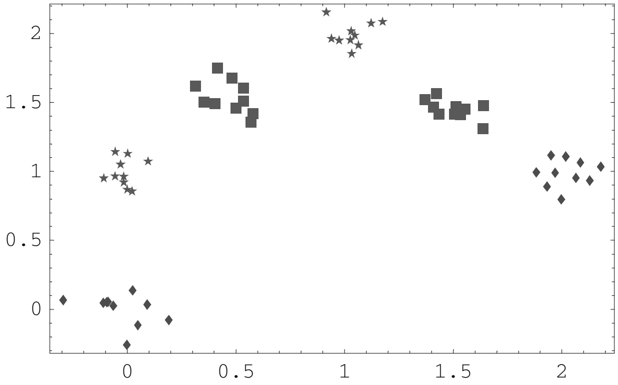
```
In[2]:=  << vqthreeclasses.dat;
```

The data vectors are stored in the matrix *x* with one data vector per row. The correct classes of the data vectors are indicated in the output data *y*. The data format follows the general standard of the package as described in Section 3.2, Package Conventions.

It is always instructive to look at data. By visually inspecting the plot you may understand the classification problem better, which will help you make better decisions in the problem-solving process. In this case, you have two-dimensional data that can be plotted right away.

Plot the data.

```
In[3]:=  NetClassificationPlot[x, y, SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]}]
```

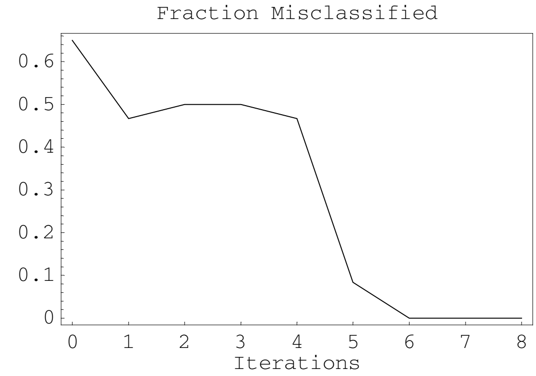


From the plot, it is evident that the data is divided into six clusters. Also the clusters form pairs so that there are three classes with two clusters in each.

Initialize and train a VQ network with seven codebook vectors for eight iterations. The codebook vectors will be distributed among the three classes, giving three for the first class and two each for the two last classes. This is obviously one more than necessary, which will be illustrated in the plots.

Initialize and train a VQ network.

```
In[4]:= {vq, fitrecord} = VQFit[x, y, 7, 8];
```



```
VQFit::DeadNeuron :
 Some codebook vectors are not used by the data. They have 'died'
   during the training. Use UnUsedNeurons to find out
   which they are and decide if you want to remove them.
```

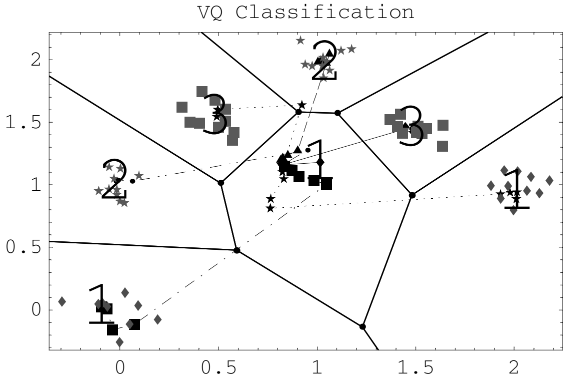
At the end of the training, the criterion, the fraction of misclassified data vectors, is plotted. From the plot you can decide if more training iterations are necessary.

The warning message `VQFit::DeadNeuron` indicates that there are codebook vectors not being used in the classification. This will be investigated in the following section.

The training can be illustrated using the training record and the function `NetPlot`. In the two-dimensional case the default is to plot the evolution of the codebook vectors during the training, the Voronoi cells of the final classifier, and the data, all together.

Plot the development of the classifier.

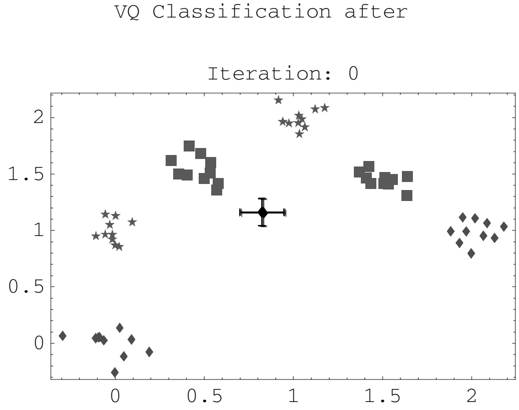
```
In[5]:= NetPlot[fitrecord, x, y, SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]},
          CbvSymbol → {TextStyle → {FontSize → 30}}]
```

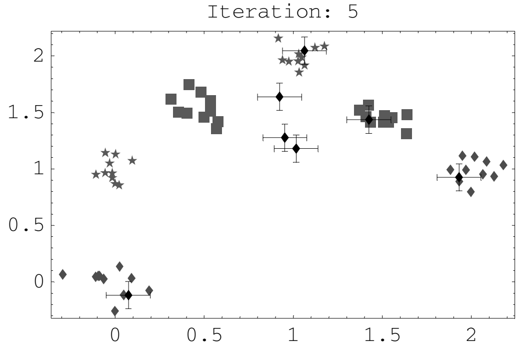


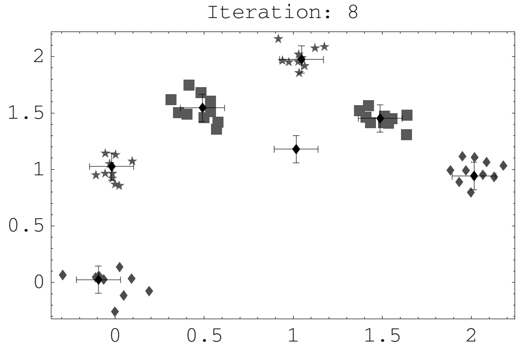
From this plot you can see the unused codebook vector.

Illustrate the training with a series of plots.

```
In[6]:= NetPlot[fitrecord, x, y, DataFormat → DataMapArray,
          SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]}, Voronoi → False]
```







If you prefer, you can animate the results as described in Section 5.2.1, Function Approximation in One Dimension.

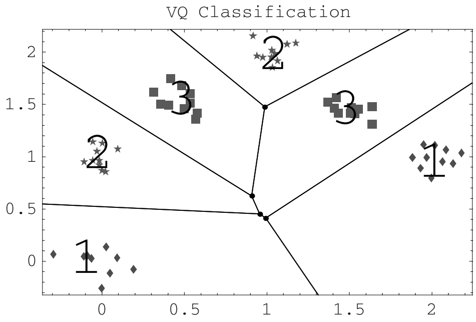
The unused codebook vector can be identified from the previous plots, or with the help of `UnUsedNeurons`. Then, you can remove it using `NeuronDelete`. Notice that the result given from `UnUsedNeurons` has to be modified slightly to fit the input format of `NeuronDelete`.

Identify the unused codebook vector and remove it.

```
In[7]:= UnUsedNeurons[vq, x]
        {1, Flatten[%][[1]]}
        vq = NeuronDelete[vq, %]

Out[7]= {{1}, {}, {}}

Out[8]= {1, 1}

Out[9]= VQ[{-Codebook Vectors-},
         {CreationDate → {2002, 4, 3, 14, 33, 3}, AccumulatedIterations → 8}]
```

Look at the classification with the modified network.

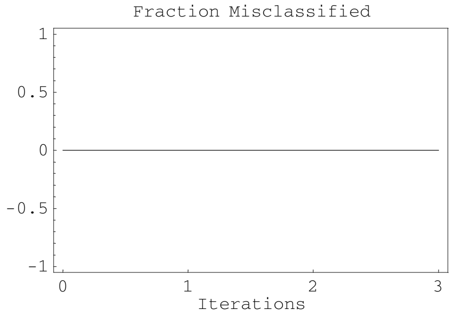
```
In[10]:= NetPlot[vq, x, y, SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]},
          CbvSymbol → {TextStyle → {FontSize → 30}}]
```



It is often good to end the training with a few iterations of the training algorithm in batch mode. You should also always consider applying more training after the network structure has been modified.

Apply three additional training iterations with the batch algorithm.

```
In[11]:= {vq, fitrecord2} = VQFit[x, y, vq, 3, Recursive → False];
```



Although the classification performance is perfect already at the beginning of this additional training, in general, this training will fine-tune the positions of the codebook vectors.

The obtained VQ network can be evaluated on any (new) data vectors by just using the evaluation rule for VQ objects.

Evaluate the VQ network on two data vectors.

```
In[12]:= vq[{{1, 1}, {3, 3}}]

Out[12]= {{0, 0, 1}, {0, 0, 1}}
```

The class to which the data vectors are classified is indicated by the columns that contain a 1. The evaluation rule is actually everything you need to use the VQ network. You can use all available *Mathematica* commands to illustrate the result. There are, however, some commands available to facilitate the illustration and use of VQ networks.

Find the percentage of misclassified data vectors.

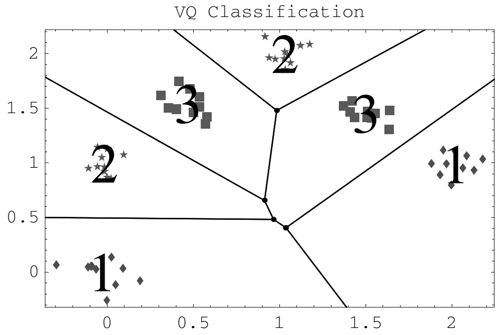
```
In[13]:= VQPerformance[vq, x, y] * 100

Out[13]= 0
```

Find the average distance between the data vectors and the closest codebook vector.

```
In[14]:= VQDistance[vq, x]

Out[14]= 0.116431
```

Display the obtained classifier together with the data.

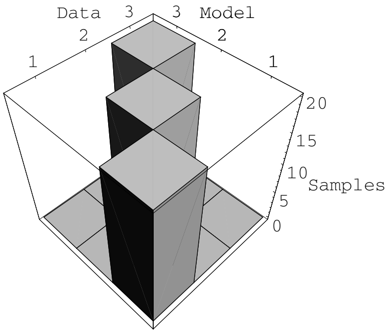
```
In[15]:= NetPlot[vq, x, y, SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]},
          CbvSymbol → {TextStyle → {FontSize → 30, FontFamily → "Times"}}]
```



Using the command NetPlot, the classification result can be presented in several ways. If the DataFormat option is set to BarChart, you obtain a three-dimensional bar chart where the diagonal bars correspond to correctly classified data vectors, and the off-diagonal ones correspond to misclassified data vectors.

Display the result with bar charts.

```
In[16]:= NetPlot[vq, x, y, DataFormat → BarChart]
```

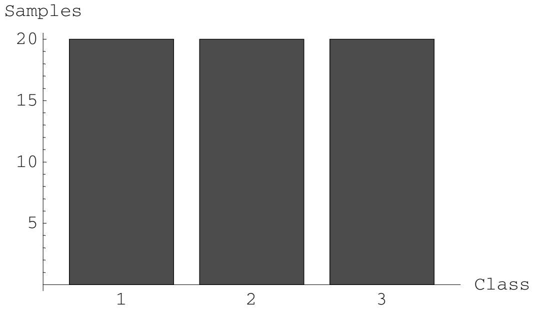


The heights of the columns in the bar chart correspond to the number of data vectors. When the classification is exact, the bar chart has only diagonal columns, each of which represents a class and has a height equal to that class's population. If some vectors of a given class are classified incorrectly by the VQ network model, an off-diagonal column will be shown with the appropriate height.

If no output is supplied, the result is given in a two-dimensional bar chart. It illustrates the distribution of the classification of the supplied data.

Illustrate the classification result without output data.

```
In[17]:= NetPlot[vq, x, DataFormat → BarChart]
```



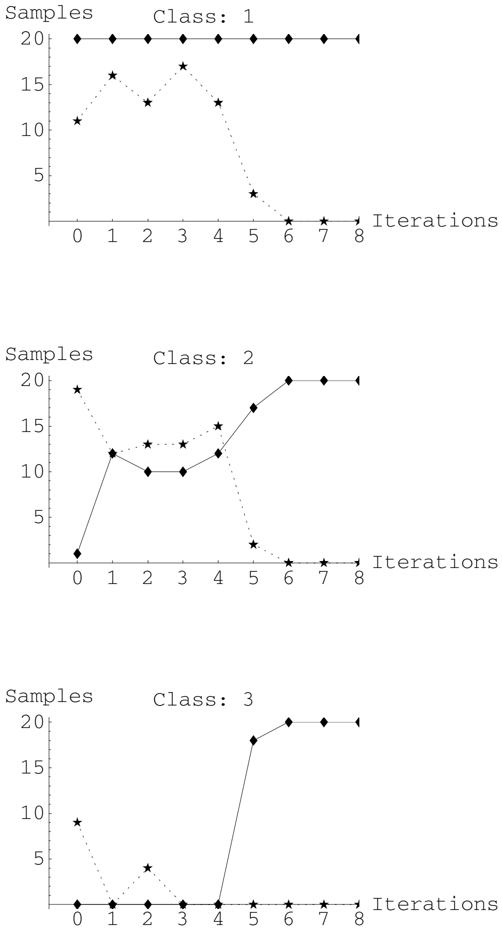
If the training record, the second output argument from `VQFit`, is submitted instead of the VQ network, then the improvements of the VQ network during the training can be illustrated. The intermediate result during the training was logged by the frequency given by the `ReportFrequency` option of `VQFit`.

By choosing `DataFormat→ClassPerformance` the classification result for each class is plotted.

Plot the classification versus training iteration.

```
In[18]:= NetPlot[fitrecord, x, y, DataFormat → ClassPerformance]
```

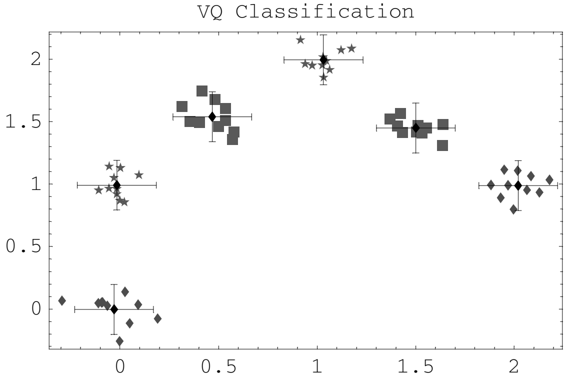
```
Correctly/incorrectly classified data
```



You can have a cross at the position of each codebook vector instead of the Voronoi cells. This is done by choosing `Voronoi→False`. The size of the crosses is automatically set, but you can change it by setting the option `CbvSymbol` to a list of numbers indicating the size desired.

Illustrate the classification with crosses of specified size instead of Voronoi cells.

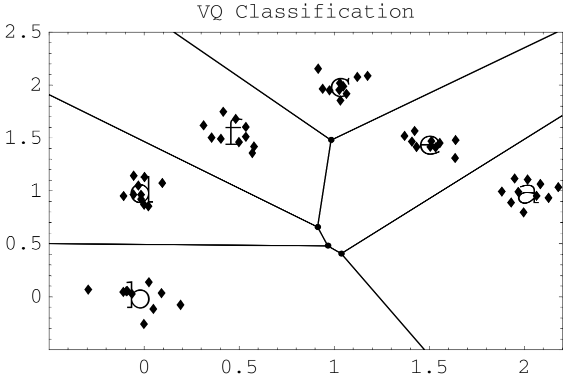
```
In[19]:= NetPlot[vq, x, y, Voronoi →False,
          SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]}, CbvSymbol → {0.2, 0.2}]
```



The label of the codebook vectors can be changed by using the option `CbvSymbol`, and you can also submit any option of `MultipleListPlot`. Here the plot range is changed.

Choose other labels for the codebook vectors and modify the plot range.

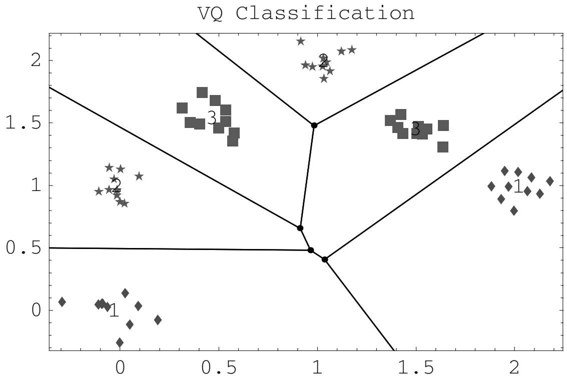
```
In[20]:= NetPlot[vq, x, PlotRange → {{-0.5, 2.2}, {-0.5, 2.5}},
          CbvSymbol → {"a", "b", "c", "d", "e", "f", TextStyle → {FontSize → 20}}]
```



In the plot above there was no true output signal submitted in the call. Then all data are marked identically in the plot. If you do not have the true output available, and you want to distinguish the data items classified to the different classes, you can include the classification according to the VQ network in the function call. However, notice the difference from the case when you submit the true output. It is only in the second case that you can see misclassified data and get a feeling of the quality of the classifier.

Indicate the data classified to the different classes.

```
In[21]:= NetPlot[vq, x, vq[x], SymbolStyle → {Hue[.0], Hue[.6], Hue[.8]}]
```



The classification result can also be displayed in a table. Each entry in the table represents one class of the VQ network and includes information about the data vectors assigned to that class. For each class, a series of entries of the form $a:b$ are given where $a$ indicates the class label of the data, and $b$ indicates the number of samples from this class. For example, if the second entry is $2:20,\ 3:2$, then this means that 20 samples from class 2 and two samples from class 3 are assigned to class 2 by the network.

First the class label according to the supplied output data is given, followed by ":" , and then the number of data vectors of this kind assigned to the class is given.

Present the classification with a table.

```
In[22]:= NetPlot[vq, x, y, DataFormat → Table]
```

```
        VQ Classification

 1:20      2:20      3:20
```

### 11.2.2 VQ in Three-Dimensional Space

In this three-dimensional example it is harder to illustrate the classification result. However, since many real problems are based on higher-dimensional data vectors, this example illustrates how you can proceed when the data cannot be visualized in two-dimensional plots.

Read in the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << vqthreeclasses3D.dat;
```

The three-dimensional data set consists of three classes, each of which is divided into two clusters. The input vectors are stored in the matrix $x$ and the output vectors in the matrix $y$, following the standard format of the package described in Section 3.2, Package Conventions.

Check the dimensionality.

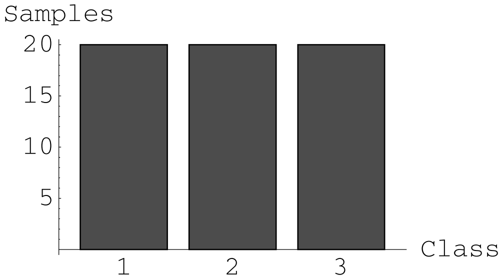
```
In[3]:= Dimensions[x]
        Dimensions[y]

Out[3]= {60, 3}

Out[4]= {60, 3}
```

Obviously, there are 60 three-dimensional data values divided into three classes.

You can check how the data is distributed between the different classes. If the distribution is very skewed, you might want to take some measures (for example, selecting more data from the classes that might not be well represented) prior to the training.

Plot the distribution of data vectors over the classes.

```
In[5]:= NetClassificationPlot[x, y]
```

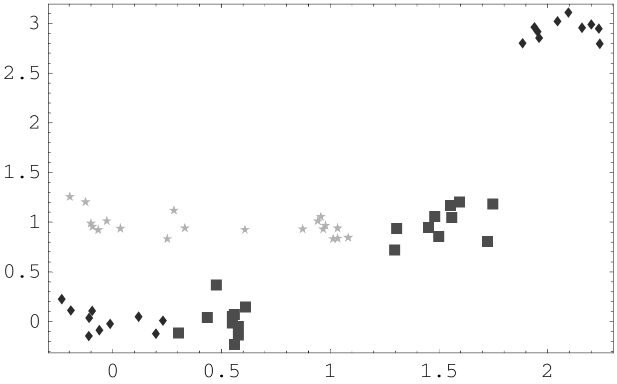


Each bar represents the number of data vectors in that class.

It is possible to project the high-dimensional data vectors to a two-dimensional space and then look at the projection. This is done by multiplying the input data vectors with a projection matrix.

Look at a projection of the data.

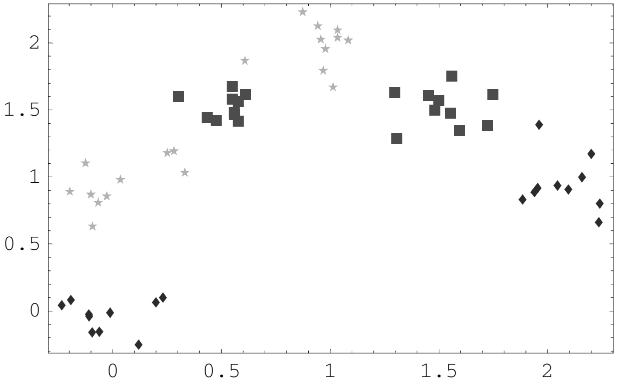
```
In[6]:= NetClassificationPlot[x . {{1, 0}, {0, 0}, {0, 1}},
         y, SymbolStyle → {Hue[.7], Hue[.5], Hue[.0]}]
```



The problem is, of course, to decide how the projection matrix should be chosen. It is hard to give any general recommendations other than that you should try several different projections.

Choose a different projection.

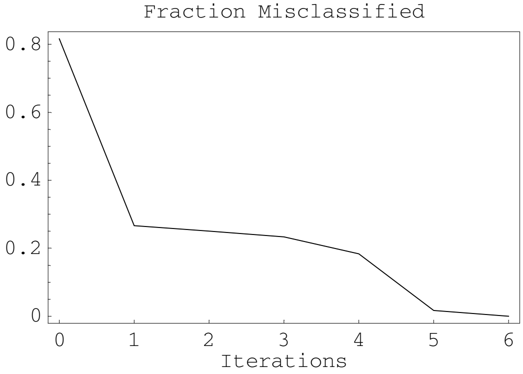
```
In[7]:= NetClassificationPlot[x . {{1, 0}, {0, 1}, {0, 0}},
         y, SymbolStyle → {Hue[.7], Hue[.5], Hue[.0]}]
```



If you find a projection so that the classes are well separated, then in most cases it is better to work on the projected data. Here the original three-dimensional data is used. Train a VQ network to the data using six codebook vectors.

Train a VQ network.

```
In[8]:= {vq, fitrecord} = VQFit[x, y, 6];
```



The obtained VQ network can be evaluated in different ways. You can always use the standard evaluation rule and then use general *Mathematica* commands to illustrate the result in a plot.

Evaluate the VQ network on two data vectors.

```
In[9]:=  vq[{{1, 1, 1}, {3, 3, 3}}]

Out[9]=  {{0, 0, 1}, {1, 0, 0}}
```

The number of the column designated by 1 is the class to which the corresponding data vector is assigned.

You can also use the special commands for illustrating VQ networks and their performance.

Get the fraction of the misclassified data vectors.

```
In[10]:=  VQPerformance[vq, x, y]

Out[10]=  0
```

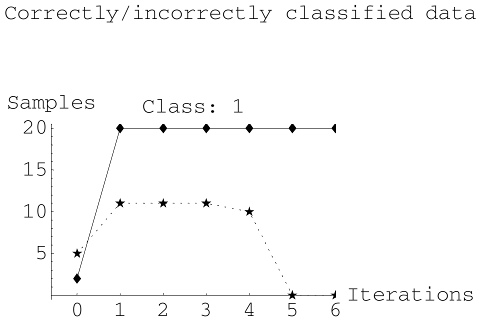
Check the mean distance between the data vectors and the codebook vectors.

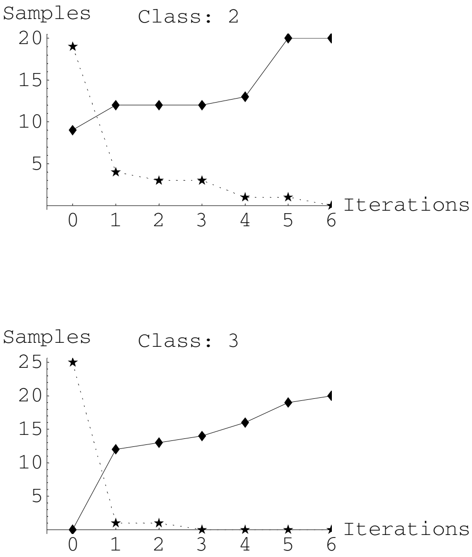
```
In[11]:=  VQDistance[vq, x]

Out[11]=  0.226935
```

By applying `NetPlot` to the training record you obtain one plot for each class showing the improvement of the classifier versus training iterations.

Plot the improvement of the classification performance.

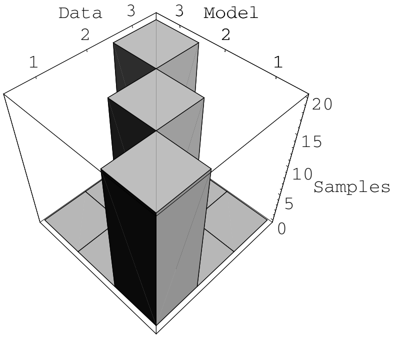
```
In[12]:=  NetPlot[fitrecord, x, y]
```





Plot the classification result with bar charts.

```
In[13]:= NetPlot[vq, x, y]
```



The classification result can also be presented with a table. There is one entry for each class according to the classification by the VQ network. At each entry the number of data vectors and their classes, as indicated by the output data, are listed.

Illustrate the classification with a table.

```
In[14]:= NetPlot[vq, x, y, DataFormat → Table]
```

```
   VQ Classification
┌──────┬──────┬──────┐
│      │      │      │
│ 1:20 │ 2:20 │ 3:20 │
└──────┴──────┴──────┘
```

### 11.2.3 Overlapping Classes

When input data exhibits major overlaps among their various classes, the training algorithm for VQ networks must be able to cope. The LVQ1 algorithm's codebook vectors will not be able to converge to the class cluster centers when data vectors from other classes are in the vicinity. In fact, the code vectors will repel away from vectors of incorrect classes. In contrast, the competitive algorithm is capable of dealing with this situation by creating codebook vectors that are independent of data that are not from the class of interest. This will be illustrated in a small example with two overlapping Gaussian clusters. See Section 10.1.2, UnsupervisedNetFit, and Section 11.1.2, VQFit, for details on the algorithms.

Load the *Neural Networks* package and a data set consisting of two overlapping classes.

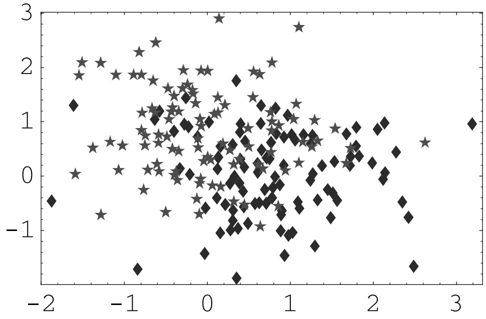
```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << overlapping.dat;
```

The input data vectors are stored in the matrix $x$ and the output in $y$ following the standard format in Section 3.2, Package Conventions.

Look at the data.

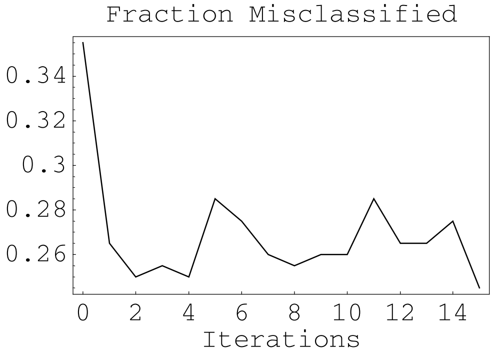
```
In[3]:= NetClassificationPlot[x, y, SymbolStyle → {Hue[.7], Hue[.0]}]
```



First a VQ network is trained using the default LVQ1 algorithm.

Train a VQ network with two codebook vectors using `LVQ1`.

```
In[4]:= {vq, fitrecord} = VQFit[x, y, 2, 15];
```



The mean distance between the data points and the codebook vector is the smallest possible if the codebook vectors are placed in the center of the clusters.

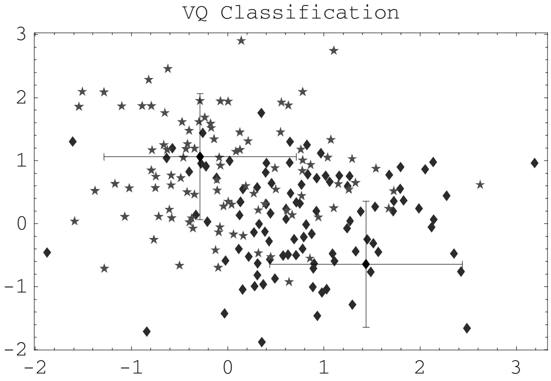
The mean distance between the data vectors and the codebook vector.

```
In[5]:= VQDistance[vq, x]

Out[5]= 0.968518
```

You can easily plot the position of the codebook vectors.

Plot the data and the classifier.

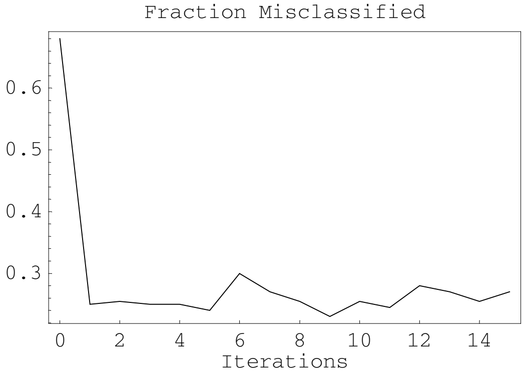
```
In[6]:=  NetPlot[vq, x, y, Voronoi → False,
           CbvSymbol → {1, 1}, SymbolStyle → {Hue[.7], Hue[.0]}]
```



This result will now be compared to the `Competitive` command.

Train a VQ network with competitive training algorithm.

```
In[7]:=  {vq, fitrecord} = VQFit[x, y, 2, 15, Method → Competitive];
```



Check the mean distance between the data points and the codebook vectors. It is a little bit smaller than before when the LVQ1 algorithm was used. The reason for this is that the codebook vectors converge to the cluster centers when the competitive algorithm is used.

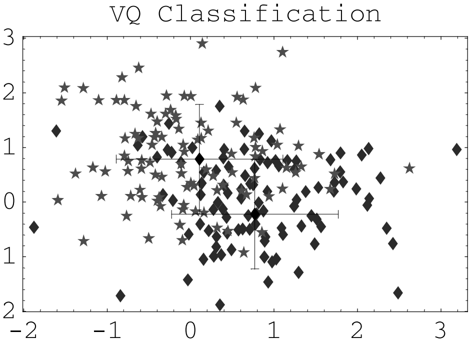
Find the mean distance between the data vectors and the codebook vector.

```
In[8]:= VQDistance[vq, x]

Out[8]= 0.881444
```

Look at the position of the codebook vectors and compare this plot with the earlier one obtained with LVQ1.

Plot the data and the codebook vectors.

```
In[9]:= NetPlot[vq, x, y, Voronoi → False,
         CbvSymbol → {1, 1}, SymbolStyle → {Hue[.7], Hue[.0]}]
```



### 11.2.4 Skewed Data Densities and Badly Scaled Data

Input data vectors are classified to the closest codebook vector using the Euclidian distance. This might not be a good distance measure if the data distributions are very skewed. Sometimes it is possible to scale the data to circumvent this problem, but not always. In this example one possible problem is illustrated.

Load the *Neural Networks* package.

```
In[1]:= << NeuralNetworks`
```

The file vqskewed.dat contains two-dimensional data from two classes with different distributions.

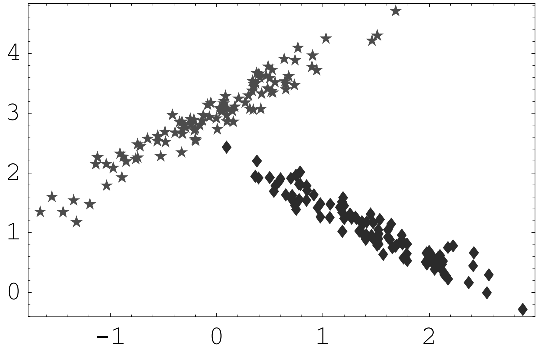
Load the data.

```
In[2]:= << vqskewed.dat;
```

The input vectors are contained in *x* and the output in *y* according to the standard format of the *Neural Networks* package described in Section 3.2, Package Conventions.

Plot the data.

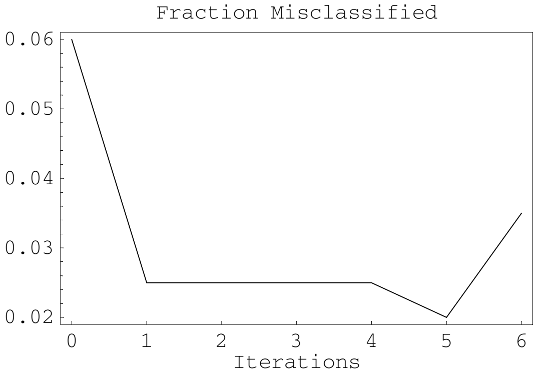
```
In[3]:= NetClassificationPlot[x, y, SymbolStyle → {Hue[.7], Hue[.0]}]
```



As seen in the plot, the data distributions are very skewed. Consider a VQ network to classify the data vectors using two codebook vectors, one for each class.

Train a VQ network.

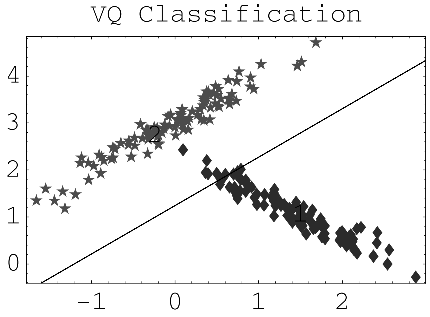
```
In[4]:= {vq, fitrecord} = VQFit[x, y, 2];
```



Plot the data together with the classifier. Some data vectors will be incorrectly classified.

Plot the data vectors and the classification result.

```
In[5]:=  NetPlot[vq, x, y, SymbolStyle → {Hue[.7], Hue[.0]}]
```



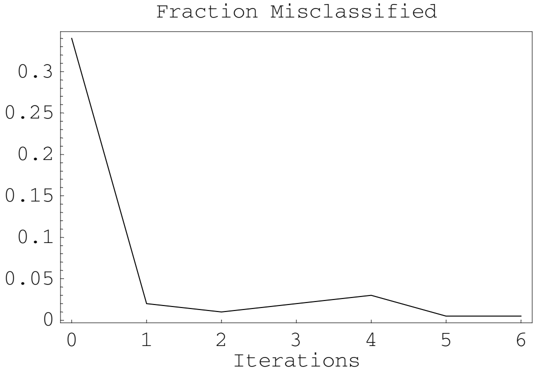
Find the percentage of misclassified data vectors.

```
In[6]:=  VQPerformance[vq, x, y] * 100

Out[6]=  3.5
```

Consider now a VQ network with more codebook vectors for each class. This will make it possible to describe the extension of the data.

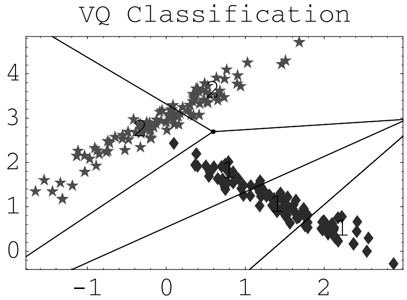
Train a VQ network with three codebook vectors for the first class and two for the second.

```
In[7]:=  {vq, fitrecord} = VQFit[x, y, {3, 2}];
```



Plot the data vectors and the classification result.

```
In[8]:=  NetPlot[vq, x, y, SymbolStyle → {Hue[.7], Hue[.0]}]
```



Now you have three codebook vectors representing the first class and two for the second one. In the plot this is indicated by three codebook vectors labeled 1 and two labeled 2.

Is the classification any better than if only two codebook vectors are used?

Find the percentage of misclassified data vectors.

```
In[9]:=  VQPerformance[vq, x, y] * 100

Out[9]=  0.5
```

### 11.2.5 Too Few Codebook Vectors

In this example you will see what can happen if you have too few codebook vectors for some of the classes. There is no general rule describing how to choose the number of codebook vectors, so they have to be chosen by trial and error in most cases.

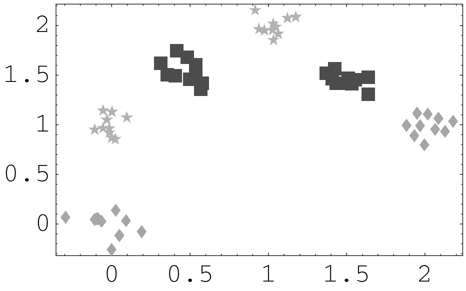
Load the *Neural Networks* package and the data.

```
In[1]:=  << NeuralNetworks`

In[2]:=  << vqthreeclasses.dat;
```

The data vectors are stored in the matrix $x$ with one data vector on each row. The data format follows the general standard of the package as described in Section 3.2, Package Conventions.

Plot the data.

```
In[3]:=  NetClassificationPlot[x, y, SymbolStyle → {Hue[.1], Hue[.5], Hue[.0]}]
```

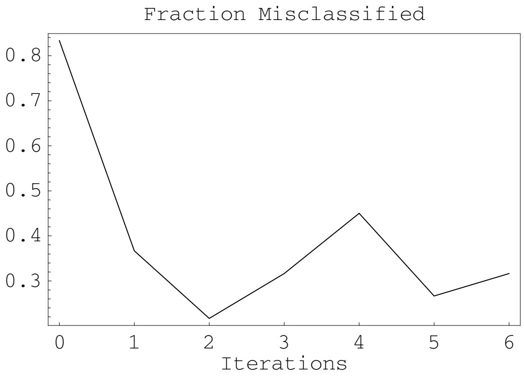


The data is divided into three classes, each consisting of two clusters.

Initialize and train a VQ network with four codebook vectors. They will be distributed among the three classes, giving two for the first class and one each for the other two classes.

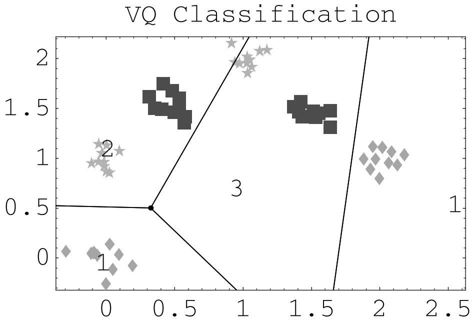
Initialize and train a VQ network.

```
In[4]:=  {vq, fitrecord} = VQFit[x, y, 4];
```



Display the obtained classifier together with the data.

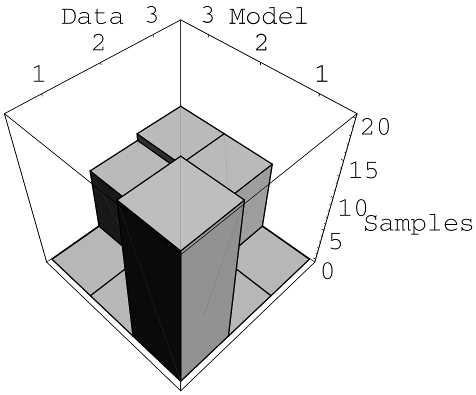
```
In[5]:= NetPlot[vq, x, y, SymbolStyle → {Hue[.1], Hue[.5], Hue[.0]}]
```



The result is not very impressive, as is also evident in the bar chart.

Display the result with bar charts.

```
In[6]:= NetPlot[vq, x, y, DataFormat → BarChart]
```



The off-diagonal bars correspond to misclassified data vectors and, as you can see, there are plenty of them. The reason for this is that there is only one codebook vector to describe two clusters. This happens for two of the classes.

## 11.3 Change Step Length

The option `StepLength` works for VQ networks exactly as it does for the unsupervised networks. If the default does not suffice, you can supply another function that takes the iteration number as input and gives the step length as output. You can also submit a numerical value, which gives a constant step length. These possibilities are illustrated here.

Read in the *Neural Networks* package.

```
In[1]:=  << NeuralNetworks`
```

A set of test data is used to illustrate the `StepLength` option.

Load the data vectors and output indicating class.

```
In[2]:=  << vqthreeclasses.dat;
```

The input data vectors are stored in the matrix *x* and the output in *y*. The data format follows the general standard of the *Neural Networks* package as described in Section 3.2, Package Conventions.

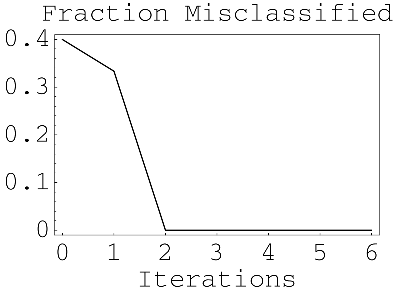
Define a function to give the step length as a function of the iteration number.

```
In[3]:=  step = Function[n, 1 / (5. + n)];
```

The initialized VQ network is then trained with the new step length function.

Initialize and train a VQ network.

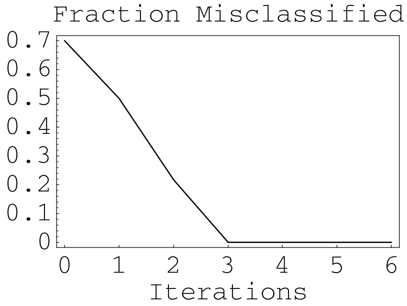
```
In[4]:=  {vq, fitrecord} = VQFit[x, y, 6, StepLength → step];
```



Instead of supplying a function, you can also submit a constant step length.

Train with a constant step length.

```
In[5]:= {vq, fitrecord} = VQFit[x, y, 6, StepLength → 0.1];
```



## 11.4 Further Reading

Most general books on neural networks cover vector quantization algorithms. Here are some suggestions:

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., New York, Macmillan, 1999.

J. Herz, A. Krough, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Reading, MA, Addison-Wesley, 1991.

T. Kohonen, *Self-Organizing Maps*, Berlin, Germany, Springer-Verlag, 1995.

# 12 Application Examples

This chapter contains examples in which several different neural network models are applied and compared.

## 12.1 Classification of Paper Quality

In this example different neural classifiers are compared using data from a hybrid gas array sensor, an *electronic nose*. The odors from five different cardboard papers from commercial manufacturers were recorded with the electronic nose. Different kinds of classifiers will be trained to determine the origin of an unknown sample.

The data was kindly contributed by the Swedish Sensor Centre, S-SENCE. (See their website at www.ifm.liu.se/Applphys/S-SENCE.) More information on the data set can be found in "Identification of paper quality using a hybrid electronic nose" by Martin Holmberg, Fredrik Winquist, Ingemar Lundström, Julian W. Gardner, and Evor L. Hines, *Sensors and Actuators B* **26-27** (1995), pp. 246–249.

Load the *Neural Networks* package and the data.

```
In[1]:= << NeuralNetworks`
```

```
In[2]:= << paper.dat;
```

There is one data set for estimation, *xs* and *ys*, and one for validation of the classifiers, *xv* and *yv*.

Check the dimensions of the data.

```
In[3]:= Dimensions[xs]
        Dimensions[ys]

Out[3]= {48, 15}

Out[4]= {48, 6}
```

There are 48 data samples available for estimation. Each paper sample is characterized by 15 $x$ values from the 15 sensors in the electronic nose sensor array.

Five different types of cardboard paper and plain air were measured, making six possible output classes. The correct class of each data sample is stored in $y$, with a 1 in the appropriate column indicating the class of the sample. This follows the general format of classification data described in Section 3.2.1, Data Format.

Check the class of the 27th sample of validation data.

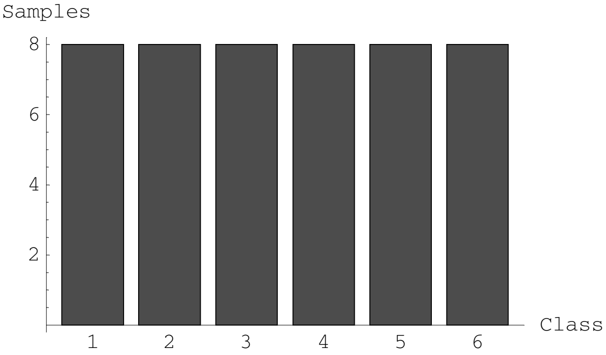
```
In[5]:=  yv[[27]]

Out[5]=  {0, 0, 1, 0, 0, 0}
```

The 27th sample belongs to class 3.

It is always a good idea to check how many data samples there are from each class.

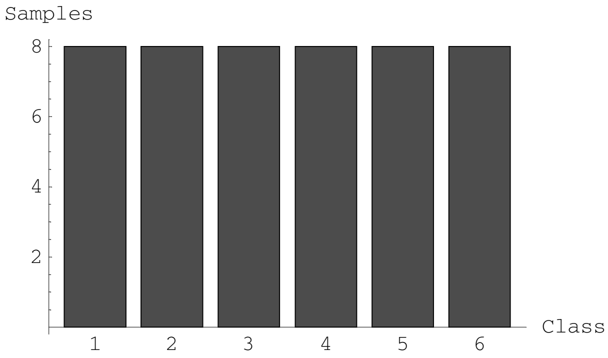
Look at the distribution of estimation data over the classes.

```
In[6]:=  NetClassificationPlot[xs, ys]
```



Look at the distribution of validation data over the classes.

```
In[7]:= NetClassificationPlot[xv, yv]
```



### 12.1.1 VQ Network

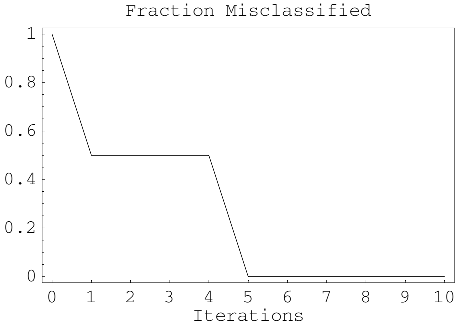
First, try a VQ network on the paper data. You need at least one codebook vector for each class, a minimum of six codebook vectors. More information on the VQ network can be found in Chapter 11, Vector Quantization. The result will vary each time the commands are evaluated, due to the random nature of the initialization and training processes.

Load *Neural Networks*, the additional *Mathematica* standard add-on package, and the data.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`

In[3]:= << paper.dat;
```

Initialize a VQ network with six codebook vectors and train it ten iterations with the `Competitive` algorithm.

```
In[4]:=  {vq, fitrecord} = VQFit[xs, ys, 6, 10, Method → Competitive];
```



Obtain some information about the trained network.

```
In[5]:=  VQInformation[vq]

Out[5]=  VQInformation[VQ[{-Codebook Vectors -},
           {CreationDate → {2002, 4, 3, 14, 39, 23}, AccumulatedIterations → 10}]]
```

The trained VQ network can now be used to classify paper samples by simply applying it to new data vectors.

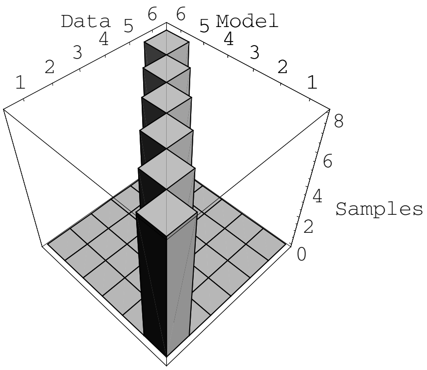
Use the trained VQ network to classify data sample 15 of the validation data.

```
In[6]:=  vq[xs[[15]]]

Out[6]=  {{0, 0, 1, 0, 0, 0}}
```

The classification result can also be illustrated using `NetPlot`. Using `NetPlot` on a data set with more than two dimensions produces a bar chart with the correctly classified samples on the diagonal. It is interesting to compare the classification results of the estimation and validation data.

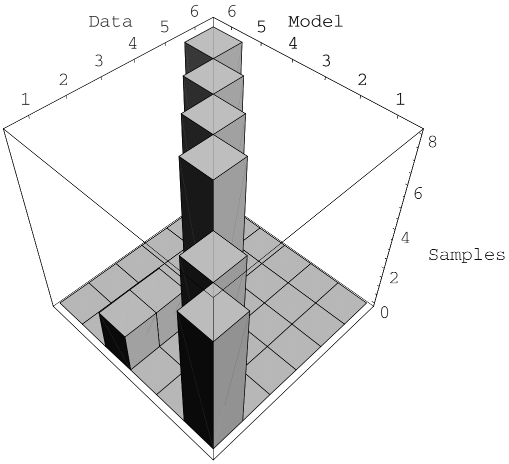
Present the classification evaluated using estimation data.

```
In[7]:= NetPlot[vq, xs, ys]
```



Present the classification evaluated using validation data.

```
In[8]:= NetPlot[vq, xv, yv]
```



From the plots, it is clear that most samples were correctly classified by the VQ network, although no perfect classifications were obtained on the validation data. The off-diagonal bars correspond to incorrectly classified data and the *x* and *y* axes show from which classes they come. Another way to illustrate this is to use the option `Table`.

Illustrate the classification on validation data with a table.

```
In[9]:=  NetPlot[vq, xv, yv, DataFormat → Table]
```

VQ Classification

| | | | | | |
|---|---|---|---|---|---|
| 1:6 | 2:6 | 3:8 | 4:8<br>2:2<br>1:2 | 5:8 | 6:8 |

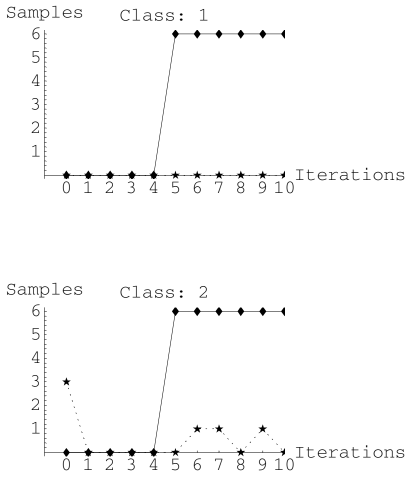
Each box illustrates the data assigned to a class. For example, the second box from the left shows that six data samples from class 2 were assigned to the second class. Note that this may turn out differently if you repeat the example.

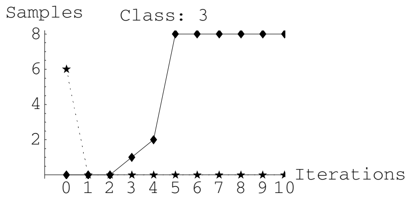
You can also look at how the classification improves for each class as a function of the number of training iterations. In this way you can see if there is a problem with any specific class.

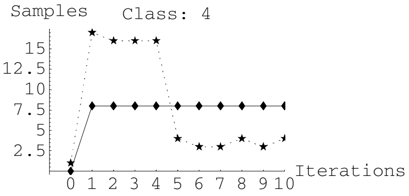
Plot the progress of the classifier on the validation data.

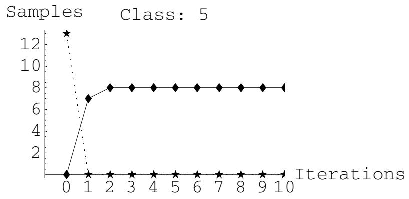
```
In[10]:=  NetPlot[fitrecord, xv, yv]
```

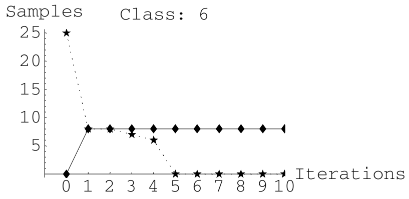
Correctly/incorrectly classified data











The dashed lines indicate incorrectly classified data.

### 12.1.2 RBF Network

An RBF network will be used on the cardboard paper data. RBF networks are often not suitable for use in high-dimensional problems like this one, which has 15 input dimensions. It is, however, possible to choose to use only a subset of the inputs. Choosing a subset of inputs can be seen as a simple form of feature extraction. To simplify the problem, only four inputs will be used and only four of the paper types will be classified. The result will vary each time the commands are evaluated due to the random nature of the initialization and training processes.

Load *Neural Networks*, the additional *Mathematica* standard add-on package, and the data.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

```
In[3]:= << paper.dat;
```

The data is described in the beginning of Section 12.1, Classification of Paper Quality.

Select input data from sensors 5, 6, 7, and 8, and class data from classes 3 through 6.

```
In[4]:= x2s = TakeColumns[xs, {5, 8}];
        y2s = TakeColumns[ys, {3, 6}];
        x2v = TakeColumns[xv, {5, 8}];
        y2v = TakeColumns[yv, {3, 6}];
```

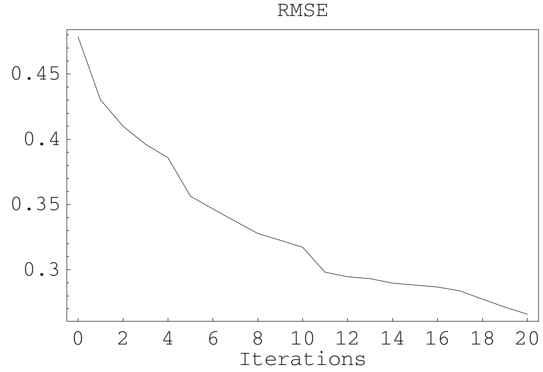
For classification, it is advantageous to add a sigmoidal nonlinearity at the output of the RBF network. This constrains the output to the range 0 to 1. Also, a better-conditioned training problem is often obtained if the linear submodel is excluded.

Initialize the RBF network with six neurons, no linear submodel, and a `Sigmoid` output nonlinearity.

```
In[8]:= rbf = InitializeRBFNet[x2s, y2s, 6, OutputNonlinearity → Sigmoid, LinearPart → False]

Out[8]= RBFNet[{{w1, λ, w2}}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2003, 9, 19, 8, 14, 50.0479376},
          OutputNonlinearity → Sigmoid, NumberOfInputs → 4}]
```

Train the RBF network for 20 iterations.

```
In[9]:=  {rbf2, fitrecord} = NeuralFit[rbf, x2s, y2s, 20];
```



The trained RBF network can now be used to classify input vectors by applying the network to them.

Classify the 24th paper data sample from the validation data.

```
In[10]:=  rbf2[x2v[[24]]]

Out[10]=  {0.163261, 0.164305, 0.106444, 0.562666}
```

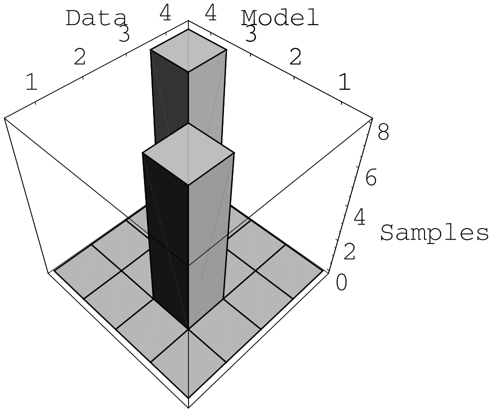
The output of this RBF network is a real-valued function that takes values between 0 and 1. A crisp classification can be obtained in various ways. A simple way is to set values larger than 0.5 to 1 and smaller than 0.5 to 0 in the following manner. (Here, `True` rather than 1 indicates the sample's class. )

```
In[11]:=  Map[# > 0.5 &, rbf2[x2v[[24]]]]

Out[11]=  {False, False, False, True}
```

The classification can also be displayed with a bar chart in which the correctly classified data is on the diagonal and the incorrectly classified samples off the diagonal. A crisp classification can be obtained by changing the output nonlinearity from the smooth step sigmoid to the discrete step by entering the option `OutputNonlinearity → UnitStep`.

Plot the classification result on the validation data.

```
In[12]:= NetPlot[rbf2, x2v, y2v, DataFormat → BarChart, OutputNonlinearity → UnitStep]
```



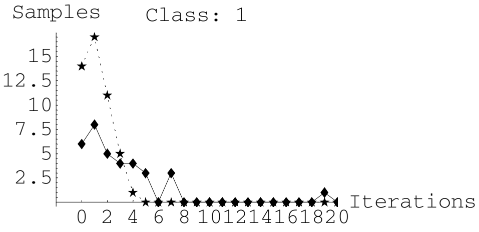
The classification result is not particularly impressive, but due to the randomness in the initialization, you may re-evaluate the example, and the result might become better.

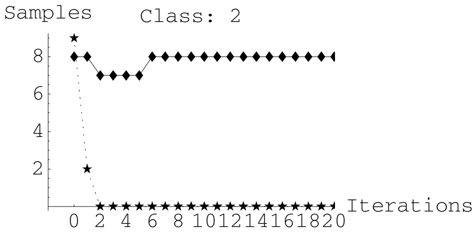
Use `NetPlot` to look at the classification performance improvement during training for each class.

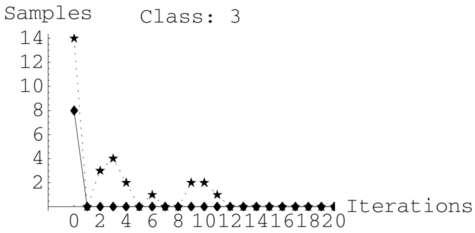
Plot the progress of the classifier on the validation data.

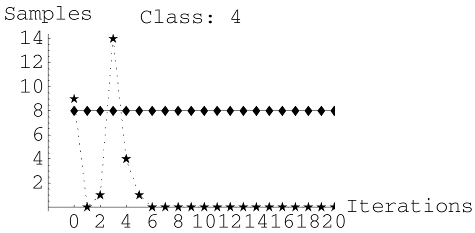
```
In[13]:= NetPlot[fitrecord, x2v, y2v,
          DataFormat → ClassPerformance, OutputNonlinearity → UnitStep]
```

```
Correctly/incorrectly classified data
```









You can repeat the example, selecting different components of the input values and different classes for the output. Because the radial basis functions have local support where they are nonzero, the training often gets trapped in poor local minima. Quite frequently, one of the classes will be totally misclassified. You will see this if you re-evaluate the example with a different initialization of the RBF network. Correct classification is obtained when there is one basis function at each class center. If any of the class centers has not managed to attract a basis function during the training, then this class will not be correctly classified.

### 12.1.3 Feedforward Network

Now try an FF neural network. As with RBF networks, a sigmoidal nonlinearity is added to the output so that the outputs are constrained to the interval 0 to 1. The result will vary each time the commands are evaluated due to the random nature of the initialization and training processes.

Load *Neural Networks*, the additional *Mathematica* standard add-on package, and the data.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

```
In[3]:= << paper.dat;
```

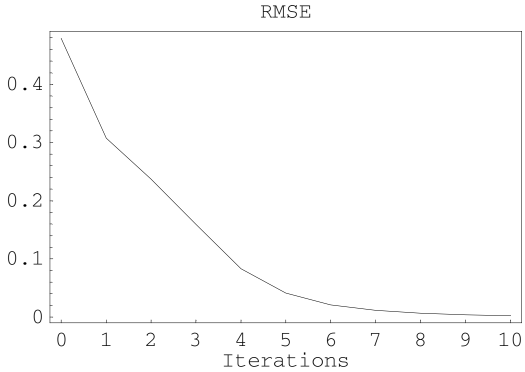
The data is described at the beginning of Section 12.1, Classification of Paper Quality.

Initialize an FF network without any hidden neurons.

```
In[4]:= fdfrwrd = InitializeFeedForwardNet[xs, ys, {}, OutputNonlinearity → Sigmoid]

Out[4]= FeedForwardNet[{{w1}}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 41, 47},
          OutputNonlinearity → Sigmoid, NumberOfInputs → 15}]
```

Train the initialized network for ten iterations.

```
In[5]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, xs, ys, 10];
```



The trained network can now be used to classify input vectors by applying the network to them.

Classify sample 27 of the validation data.

```
In[6]:=  fdfrwrd2[xv[[27]]]
```

```
Out[6]=  {0.112388, 1.58075×10^-11, 1., 1.59305×10^-7, 0.0000494222, 0.000177795}
```

A crisp classification is obtained by setting all output values greater than 0.5 to `True`.

```
In[7]:=  Map[# > 0.5 &, fdfrwrd2[xv[[27]]]]
```

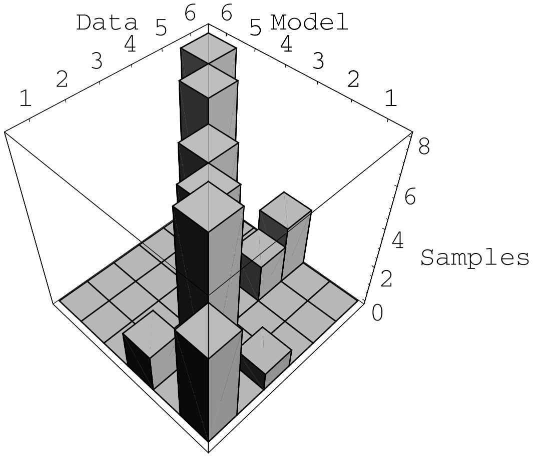
```
Out[7]=  {False, False, True, False, False, False}
```

The 27th sample is correctly classified in class 3.

As with VQ and RBF networks, classification with FF networks can also be illustrated by a bar chart with correctly classified data on the diagonal and incorrectly classified data on off-diagonal bars. By choosing `OutputNonlinearity → UnitStep` the sigmoids at the outputs are changed to a discrete step. This gives crisp classification.

Plot the classification result on the validation data.

```
In[8]:=  NetPlot[fdfrwrd2, xv, yv, DataFormat → BarChart, OutputNonlinearity → UnitStep]
```

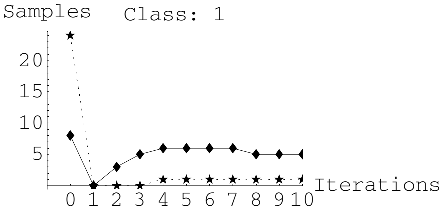


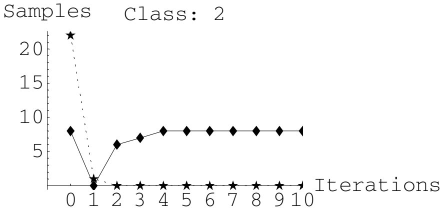
Next, plot the classification performance improvement during training for each class.

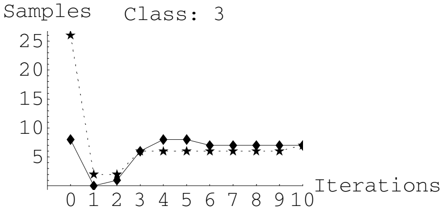
Plot the progress of the classifier on the validation data.

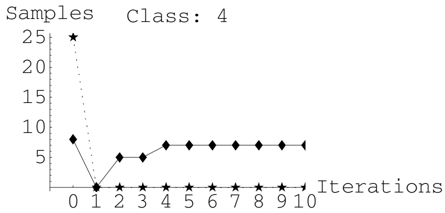
```
In[9]:= NetPlot[fitrecord, xv, yv,
          DataFormat → ClassPerformance, OutputNonlinearity → UnitStep]
```

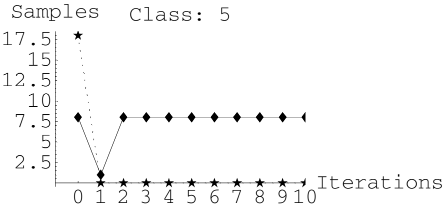
```
Correctly/incorrectly classified data
```

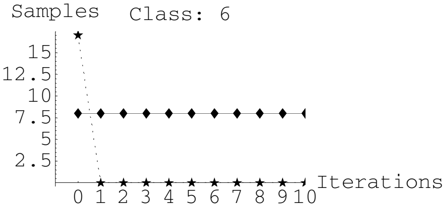












You can repeat the example using different options for the neural network structure. For example, you can introduce a layer of hidden neurons.

Three types of neural networks have been used to classify the paper quality data: VQ, RBF, and FF networks. What conclusion can you draw from the comparison? As mentioned before, RBF networks often have problems with local minima, especially if the dimension of the input space is high. To reduce this problem, only three of the available 15 dimensions were used. Of course, when 12 dimensions are neglected, there is a danger that the remaining three dimensions do not contain enough information to separate the classes. Therefore, RBF nets are not very good for this problem.

An FF network also may have problems with local minima, especially if you change the network and include a hidden layer, but these problems are not as severe as for the RBF network. You can test for problems by re-evaluating the example a few times. Though the VQ does not have any problems with local minima in this example, it may with other data. It is hard to say which classifier is the best, but the VQ network was the easiest one to train.

## 12.2 Prediction of Currency Exchange Rate

In this example, data consist of the daily exchange rates of the British pound and the German mark compared to the U.S. dollar from the beginning of 1987 to the end of 1997.

The data was contributed by Agustin Leon, Quantitative Research, Rotella Capital Management, whose contribution is gratefully acknowledged.

Load *Neural Networks*, the additional *Mathematica* standard add-on package, and the data.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

```
In[3]:= << currency.dat;
```

Two time series have now been loaded. The variable *ybr* contains the exchange rate for the British pound, and *ydeu* contains the rate for the German mark.

Check the time-series dimensions.

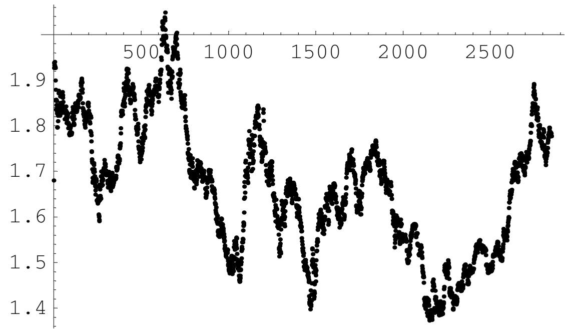
```
In[4]:= Dimensions[ybr]
        Dimensions[ydeu]

Out[4]= {2856, 4}

Out[5]= {2849, 4}
```

Exchange rates from approximately 2850 days are available. There are four rates given for each day, representing the opening, high, low, and closing prices.

First concentrate on the German mark.

Plot the high price of the German mark versus the day number.

```
In[6]:= ListPlot[Flatten[TakeColumns[ydeu, {2}]]]
```



Suppose you want to predict the opening price using the exchange rates from the previous day. Therefore, the opening price is defined as the output of the process, which will be stored in *y*, and the three other rates are defined as inputs and will be stored in *u*.

Divide the columns into input and output.

```
In[7]:= u = TakeColumns[ydeu, {2, 4}];
        y = TakeColumns[ydeu, {1}];
```

The predictor model can be described by the following equation:

$$\hat{y}(t) = g(\theta, x(t)) \qquad (1)$$

Here $\hat{y}(t)$ is the prediction of the output $y(t)$, the function $g$ is the neural network model, $\theta$ represents the parameters of the model, and $x(t)$ is the model's regressor. The regressor is given by the following equation:

$$x(t) = [y(t-1)\ u_1(t-1)\ u_2(t-1)\ u_3(t-1)]^T \qquad (2)$$

To have a fair test of the predictor, the data set is divided into training data and validation data. The second data set will be used to validate and compare the different predictors. The following commands write the training data into matrices *ue* and *ye* and the validation data into *uv* and *yv*.

Divide the data into training and validation data.

```
In[9]:=  ue = u[[Range[1000]]];
         ye = y[[Range[1000]]];
         uv = u[[Range[1001, 2000]]];
         yv = y[[Range[1001, 2000]]];
```

NeuralARX is the appropriate neural model for this problem. Such a network is initialized and estimated by the command NeuralARXFit as described in Chapter 8, Dynamic Neural Networks. First, a linear predictor model will be estimated. To find this prediction, choose FeedForwardNet without hidden neurons. To obtain the regressor in the form Equation 12.2, the regressor indices have to be chosen as follows: $n_a = \{1\}$, $n_b = \{1, 1, 1\}$, and $n_k = \{1, 1, 1\}$.

Estimate a linear model for the currency prediction.

```
In[13]:=  {model1, fitrecord} = NeuralARXFit[ue, ye,
             {{1}, {1, 1, 1}, {1, 1, 1}}, FeedForwardNet, {}, CriterionPlot → False];

          NeuralFit::LS :
           The model is linear in the parameters. Only one training iteration is necessary
             to reach the minimum. If no training iteration was performed, it is
             because the fit was completed during the initialization of the network.
```

A linear model is just a linear combination of the regressor components and a DC-level parameter; that is, a linear model in Equation 12.1 can be expressed as

$$\hat{y}(t) = a_1\, y(t-1) + b_1\, u_1(t-1) + b_2\, u_2(t-1) + b_3\, u_3(t-1) + b_4 \qquad (3)$$

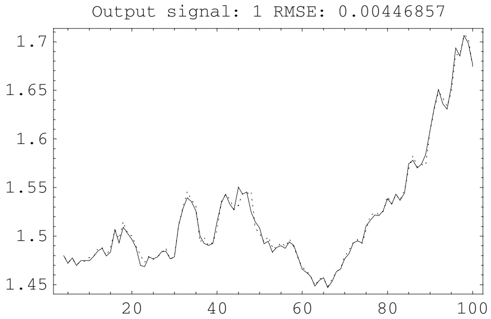
If you look at the parameters of the trained model, you see that $b_3$ is close to 1 and the rest of the parameters are close to 0. This means that the opening exchange rate is most closely correlated with the closing rate of the day before. This seems to be a very natural feature and you can, therefore, have some faith in the model.

```
In[14]:=  model1[[1]][{yy[t - 1], u1[t - 1], u2[t - 1], u3[t - 1]}]

Out[14]=  {0.00367334 - 0.0142261 u1[-1 + t] -
            0.00706428 u2[-1 + t] + 1.02698 u3[-1 + t] - 0.00762763 yy[-1 + t]}
```

Before a nonlinear model is trained on the data, the one-step prediction on the validation data is evaluated. Since the market is changing with time, the prediction is evaluated only on the 100 days following the estimation data.

Evaluate the one-step prediction on validation data.

```
In[15]:= NetComparePlot[uv, yv, model1, PredictHorizon → 1, ShowRange → {4, 100}]
```

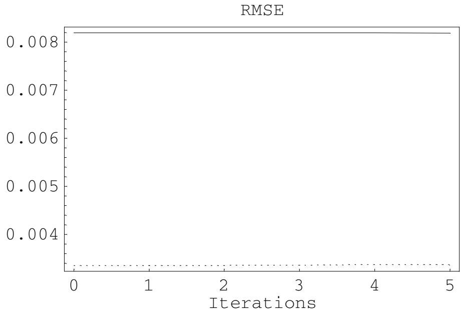


The plot shows the predicted rate together with the true rate. The two curves are so close that you can hardly see any difference between them. The RMSE of the prediction is also given, and you can also use it as a measure of the quality of the predictor.

It is now time to try nonlinear models to see if they can predict better than the linear one. An FF network with two hidden neurons is chosen. The regressor chosen is the same as for the linear model. A linear model is included in parallel with the network. Because the validation data is included in the following training, the estimate is obtained with stopped search.

Train an FF network with two neurons on the exchange rate data.

```
In[16]:= {model2, fitrecord} = NeuralARXFit[ue, ye, {{1}, {1, 1, 1}, {1, 1, 1}},
            FeedForwardNet, {2}, uv, yv, 5, LinearPart → True, Separable → False];
```

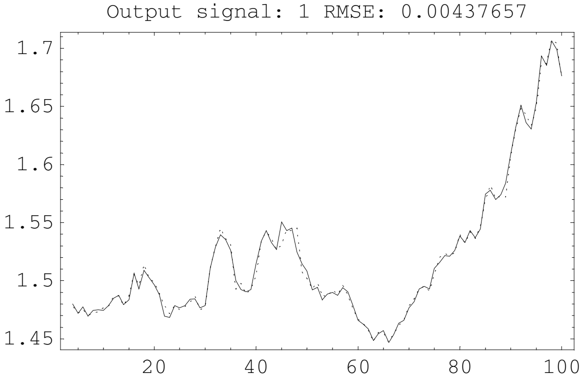


```
NeuralFit::StoppedSearch :
 The net parameters are obtained by stopped search using the supplied
   validation data. The neural net is given at the 0th training iteration.
```

The improvement of the neural network during the training is very small because the initialization of the network uses least-squares to fit the linear parameters.

Compute the one-step prediction on the same data interval used for the linear model.

```
In[17]:= NetComparePlot[uv, yv, model2, PredictHorizon → 1, ShowRange → {4, 100}]
```

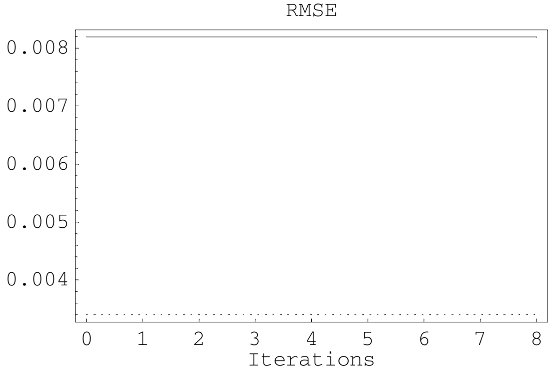


The RMSE is slightly lower for the nonlinear model than for the linear one. Thus, the prediction is slightly better. It is typical for the improvement of economic data to be small. Otherwise, it would be too easy to make money.

Now consider an RBF network. Keep the same arguments used for the FF network, except change to an RBF network with two neurons.

Initialize and train an RBF network on the exchange rate data.

```
In[18]:= {model3, fitrecord} = NeuralARXFit[ue, ye,
           {{1}, {1, 1, 1}, {1, 1, 1}}, RBFNet, 2, uv, yv, 8, Separable → False];
```

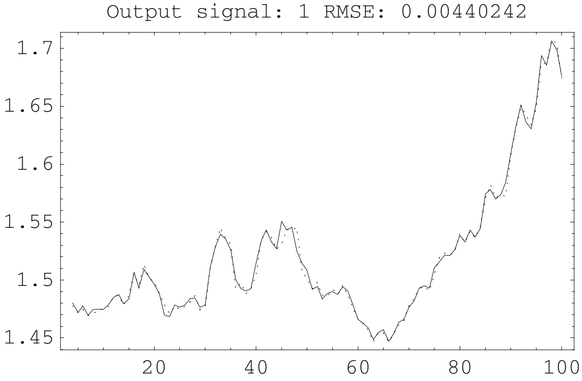


```
NeuralFit::StoppedSearch :
 The net parameters are obtained by stopped search using the supplied
   validation data. The neural net is given at the 6th training iteration.
```

The performance of the RBF network on validation data became worse during the training, causing the initialized RBF network to be returned.

Evaluate the one-step prediction with the RBF network.

```
In[19]:= NetComparePlot[uv, yv, model3, PredictHorizon → 1, ShowRange → {4, 100}]
```



The RBF network is slightly better than the linear network, but not as good as the FF network. If you re-evaluate the example, the result might change slightly due to the randomness in the initialization.

You can repeat the example changing several options, such as the following:

- Change the number of neurons.
- Change the regressor to contain more past values.
- Exclude some of the input signals from the regressor.

You can also change the data interval used in the training and in the validation. Also, try to predict the British pound instead of the German mark.

The example illustrates that it is possible to predict the opening exchange rate using the rates from the previous day. The relationship is obviously nonlinear, since the nonlinear models based on FF and RBF networks performed slightly better than the linear model.

# 13 Changing the Neural Network Structure

The three sections in this chapter describe how you can modify a network structure. The first section describes how to change the parameter values in an existing network or remove a neuron. The remaining two sections apply only to RBF and FF networks. These sections explain how to obtain special model structures by fixing some parameters at predefined values during the training and specifying the neuron activation function.

## 13.1 Change the Parameter Values of an Existing Network

This section describes how you can change a neural network by changing the values of the numerical parameters. You will need to do this if, for example, you want to implement your own training algorithms and then insert the parameters into an existing network.

The following subsections explain how to change each type of network available in *Neural Networks*, except for the perceptron and the Hopfield network. Changing the parameters of the perceptron and Hopfield network is straightforward, considering their storage format described in Section 4.1.1, InitializePerceptron, and Section 9.1.1, HopfieldFit.

### 13.1.1 Feedforward Network

First, consider the way in which network parameters are stored. FF networks are stored in objects with head `FeedForwardNet`. These objects consist of two components as described in Section 3.2, Package Conventions. The first component contains the parameters and the second component is a list of rules.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

Initialize an FF network with two inputs, two outputs, and one hidden layer containing four neurons. This is done by indicating the number of inputs and outputs with matrices of the appropriate size; that is, without using any data. A linear submodel is included in parallel with this example network. The additional parameters for the submodel are placed in the variable $\chi$ in the first component of the network.

```
In[2]:= fdfrwrd = InitializeFeedForwardNet[{Range[2]},
          {Range[2]}, {4}, LinearPart → True, RandomInitialization → True]

Out[2]= FeedForwardNet[{{w1, w2}, χ}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 55, 29},
          OutputNonlinearity → None, NumberOfInputs → 2}]
```

This FF network can be described by the following equation. This matrix description of the FF network is more compact than the description in Section 2.5.1, Feedforward Neural Networks, and makes it easier to explain the parameter storage. The input vector $x$ is augmented with a unit component to include the bias parameters in the same way as the rest of the parameters. Also the output from the hidden layer is described as a vector and augmented with a unit component.

$$\hat{y} = [\sigma([x\ 1]\ w_1)\ 1]\ w_2 + x\chi \qquad (1)$$

The matrices $w_i$ are specified by the following two rules:

- There is one column for each output of a layer.
- There is one row for each input to a layer and an additional row for the bias parameters, indicated by $b$ in Section 2.5.1, Feedforward Neural Networks.

The linear part is described by the matrix $\chi$, containing one row for each input and one column for each output. The last term only exists if the network contains a linear part.

This description holds for FF networks with any number of hidden layers. You just iterate the rules for each layer.

In the example, $w_1$ has three rows and four columns since there are two inputs and one bias in the input layer and four neurons in the hidden layer. The second matrix $w_2$ has two columns and five rows since the model has two outputs and four hidden neurons along with the associated bias.

Using this description of the storage format, it is easy to insert your own parameter values. In general, create the matrices $w_i$ and $\chi$, if you have a linear submodel. Then, put all these matrices in a list $\{\{w_1, w_2, \ldots w_n\}, \chi\}$ and place it in the first position of the FF object.

You can also change values of a subset of the parameters in the network model. This is done by manipulating the corresponding element in the network structure. Here is an example of a safe way to change an element.

Extract the matrices.

```
In[3]:=  {{w1, w2}, χ} = fdfrwrd[[1]];
```

Look at the first matrix.

```
In[4]:=  MatrixForm[w1]

  Out[4]//MatrixForm=
        ( 0.799754    0.722728   -0.0949863  -0.823133 )
        (-0.388832    0.272681    -0.200368  -0.254374 )
        ( 0.41106   -0.0473895    0.114941    0.873398 )
```

Suppose you want to set the parameters in the third column to zero so that the input to the third neuron becomes zero—that is, independent of the input to the network. This is done as follows.

```
In[5]:=  w1 = w1 /. {a_, b_, c_, d_} → {a, b, 0, d};
         MatrixForm[w1]

  Out[6]//MatrixForm=
        ( 0.799754    0.722728   0  -0.823133 )
        (-0.388832    0.272681   0  -0.254374 )
        ( 0.41106   -0.0473895   0   0.873398 )
```

You can now insert the changed matrix $w_1$, along with the rest of the unchanged parameters, into the FF network model.

Insert the changes into the model.

```
In[7]:=  fdfrwrd[[1]] = {{w1, w2}, χ};
```

### 13.1.2 RBF Network

First, investigate the storage format of the network. RBF networks are stored in objects with head `RBFNet`. The first component contains the parameters and the second component is a list of rules, as discussed in Section 3.2, Package Conventions.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

Initialize an RBF network with three inputs, two outputs, and five neurons. This is done by initializing a network with matrices of the appropriate size without any data.

Initialize an RBF network with five neurons.

```
In[2]:= rbf = InitializeRBFNet[{Range[3]}, {Range[2]}, 5, RandomInitialization → True]

Out[2]= RBFNet[{{w1, λ, w2}, χ}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 55, 49},
          OutputNonlinearity → None, NumberOfInputs → 3}]
```

The variables $w1$, $\lambda$, $w2$, and $\chi$ contain the parameters. Compare this to Figure 2.7 in Section 2.5.2, Radial Basis Function Networks.

The RBF network can be described using the following matrix form that is comparable to the description in Section 2.5.2, Radial Basis Function Networks. Here, $G$ is the basis function.

$$\hat{y} = [G\,(\lambda^2 \mid x - w_1 \mid^2)\ 1]\ w_2 + x\chi \tag{2}$$

Parameter storage is explained by the following rules:

- $w_1$ is a matrix with a center of a basis function in each column.
- $\lambda$ is a vector with one component describing the width of the basis function for each neuron.
- $w_2$ is a matrix with one column for each output of the network. The last row contains the bias parameters.
- $\chi$ is a matrix with one column for each output of the network and one row for each network input.

Retrieve some information about the RBF network.

```
In[3]:= NetInformation[rbf]

Out[3]= Radial Basis Function network. Created 2002-4-3 at 14:
          55. The network has 3 inputs and 2 outputs.  It consists of 5
          basis functions of Exp type. The network has a linear submodel.
```

The values of the parameters can be changed by modifying the corresponding element in the network object. To do this safely, follow the idea described in Section 13.1.1, Feedforward Network.

### 13.1.3 Unsupervised Network

Consider the case that you want to change the parameters in unsupervised networks.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

The procedure is demonstrated on an unsupervised network with two inputs and four codebook vectors. Instead of submitting input data when initializing the network, enter a row matrix where the number of components equals the number of inputs.

Initialize an unsupervised network.

```
In[2]:= unsup = InitializeUnsupervisedNet[{Range[2]}, 4]

Out[2]= UnsupervisedNet[{-Codebook Vectors-}, {CreationDate → {2002, 4, 3, 14, 56, 3},
          AccumulatedIterations → 0, SOM → None, Connect → False}]
```

Retrieve information about the network.

```
In[3]:= NetInformation[unsup]

Out[3]= Unsupervised network with 4 codebook vectors. Takes
          data vectors with 2 components. Created 2002-4-3 at 14:56.
```

The first component of the model is a list of four codebook vectors.

Look at the codebook vectors.

```
In[4]:= unsup[[1]] // MatrixForm

  Out[4]//MatrixForm=
        ⎛  3.30475   3.73196 ⎞
        ⎜ -0.163529  3.10258 ⎟
        ⎜  1.19623   3.13703 ⎟
        ⎝  0.357123  1.18737 ⎠
```

There is one codebook vector on each row of the matrix. You may modify the values of the vectors by changing the corresponding elements of the matrix and inserting the modified matrix on the first position in the network.

### 13.1.4 Vector Quantization Network

Because a VQ network's codebook vectors are divided into several classes, it is slightly more complicated to change the parameters in a VQ network than it is to change the parameters in an unsupervised network. This is explained next.

Load the *Neural Networks* package.

```
In[1]:= <<NeuralNetworks`
```

In this example, a VQ network with two inputs, two classes, and six codebook vectors is used.

Initialize a VQ network with six codebook vectors.

```
In[2]:= vq = InitializeVQ[Table[Range[2], {1}], Table[0, {1}, {2}], 6]

Out[2]= VQ[{-Codebook Vectors-},
         {CreationDate → {2002, 4, 3, 14, 56, 15}, AccumulatedIterations → 0}]
```

View some information about the network.

```
In[3]:= NetInformation[vq]

Out[3]= VQ network for 2 classes with {3, 3} codebook vectors per class, altogether
          6. It takes data vectors with 2 components. Created 2002-4-3 at 14:56.
```

The six codebook vectors are divided between the two classes, so that three codebook vectors are used to explain the data in each class. The codebook vectors are contained in the first element of the network.

Look at the codebook vectors.

```
In[4]:= vq[[1]]

Out[4]= {{{0.759012, 0.0104138}, {0.381793, -0.387363}, {-0.042416, 2.56475}},
         {{1.80341, 1.3081}, {0.933643, 2.53618}, {5.04324, 1.13064}}}
```

You may modify any codebook vector by giving new numerical values to the corresponding parameters in this structure. Then you insert it at the first position of the network.

If you are interested in a particular class, you can extract the codebook vectors of that class and display them as a matrix.

Look at the codebook vectors of the second class.

```
In[5]:=  vq[[1, 2]] // MatrixForm

  Out[5]//MatrixForm=
        ⎛ 1.80341   1.3081  ⎞
        ⎜0.933643  2.53618 ⎟
        ⎝ 5.04324   1.13064 ⎠
```

There is one codebook vector on each row of the matrix. You may modify the values of the vectors by changing the corresponding elements of the matrix and inserting the modified matrix on the appropriate position in the network.

## 13.2 Fixed Parameters

For FF and RBF networks, you can modify the network structure by setting some of the parameters to predefined values. Then you can exclude the predefined parameters from the training with the `FixedParameters` option. These parameters will maintain their defined values during the training; thus, fewer parameters are adapted to the data. This might be good from a bias-variance tradeoff perspective, as described in Section 7.5, Regularization and Stopped Search.

In some problems, you might know that the dependence is linear with respect to some inputs but nonlinear with respect to other inputs. The following example demonstrates how this characteristic can be built into the neural network model so that no more parameters than necessary have to be estimated. A more advanced example can be found in Section 8.2.5, Fix Some Parameters—More Advanced Model Structures.

The default of the `FixedParameters` option is `None`, which means that all parameters are trained. You can set `FixedParameters` to a list of integers specifying the parameters to hold fixed during the initialization of the network. You can also fix parameters when training with `NeuralFit`. A specification of `FixedParameters` at the training stage overrides any earlier specification at initialization. The information about fixed parameters is stored in the network model. If you want to train all parameters at a later stage, you must give the option `FixedParameters → None`.

The fixed parameters are indicated by their position in the flattened parameter structure of the network—that is, the position in the list `Flatten[`*net*`[[1]]]`, where *net* is an FF or RBF network model.

Suppose you know, through physical insight or just by assumption, that the unknown function looks something like the following, where *f* is an unknown nonlinear function and *a* is an unknown constant. This problem has two inputs and one output.

$$y = f\,(x_1) + a\,x_2 \qquad (3)$$

Therefore, you know that the function is linear in the second input $x_2$. If the model is specified to be linear in $x_2$ from the beginning, then the training should produce a better model.

Load the *Neural Networks* package.

```
In[1]:=  << NeuralNetworks`
```

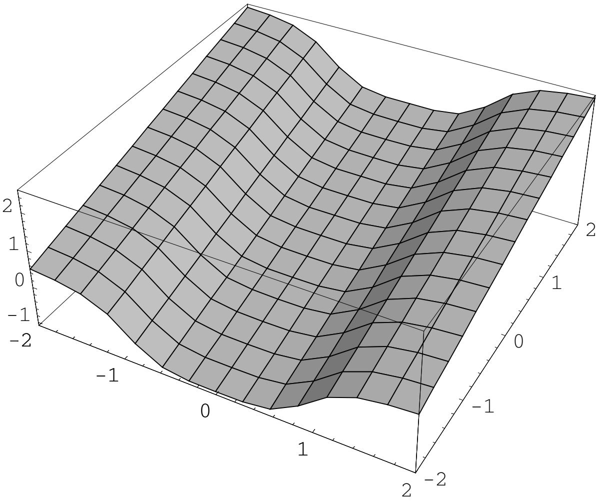
The next two steps produce data from a function of the form shown in Equation 13.3.

Obtain a “true” function.

```
In[2]:=  trueFunction = Function[ArcTan[Abs[#1^4]] + 0.5 #2]

Out[2]=  ArcTan[Abs[#1^4]] + 0.5 #2 &
```

Generate data and plot the output.

```
In[3]:= Ndata = 20;
        x = 4 *
            Flatten[Table[N[{i / Ndata, j / Ndata}], {i, 0, Ndata - 1}, {j, 0, Ndata - 1}], 1] - 2;
        y = Apply[trueFunction, x, 1];
        Plot3D[trueFunction[x1, x2], {x1, -2, 2}, {x2, -2, 2}]
```



From the plot, it is evident that the true function is linear in one of the inputs.

Now, only the data and the knowledge that the true function is linear in the second input will be used to develop an FF network model. A linear part must be included in the model. This is done with the option `LinearPart`.

Initialize an FF network including a linear part.

```
In[7]:= fdfrwrd =
         InitializeFeedForwardNet[x, y, {2}, RandomInitialization → True, LinearPart → True]

Out[7]= FeedForwardNet[{{w1, w2}, χ}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 56, 39},
          OutputNonlinearity → None, NumberOfInputs → 2}]
```

The nonlinear dependence of the second input has to be removed. This is accomplished by setting its components in the `w1` matrix to zero. The position of the `w1` matrix in the network model is evident from the previous output. Also consult the figures and the description of the w1 matrix in Section 2.5.1, Feedforward Neural Networks.

View the matrix w1.

```
In[8]:=  w1 = fdfrwrd[[1, 1, 1]]

Out[8]=  {{-0.641267, 0.170774}, {-0.119565, 0.494298}, {0.651846, 0.201474}}
```

The first row is the dependence on the first input $x_1$, the second row is the dependence on the second input $x_2$, and the third row contains the bias parameters. It is the second row that has to be set to zero. This can be done in two steps. First, the position of the row is identified.

Find the position of the second row of w1.

```
In[9]:=  pos = Position[fdfrwrd[[1]], w1[[2]]]

Out[9]=  {{1, 1, 2}}
```

Next, the elements of the second row are set to zero.

Set the second row of w1 to zero.

```
In[10]:=  fdfrwrd = ReplacePart[fdfrwrd, {0, 0}, Flatten[{1, pos}]]

Out[10]=  FeedForwardNet[{{w1, w2}, χ}, {Neuron → Sigmoid, FixedParameters → None,
            AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 56, 39},
            OutputNonlinearity → None, NumberOfInputs → 2}]
```

Whenever you manipulate the parameters directly, it is wise to check that the structure of the parameter part of the network has been changed correctly.

Check that the manipulation is correct.

```
In[11]:=  fdfrwrd[[1]]

Out[11]=  {{{{-0.641267, 0.170774}, {0, 0}, {0.651846, 0.201474}},
            {{-0.0517033}, {0.792547}, {0.273493}}}, {{-1.04737}, {-0.103775}}}
```

Only the second row of w1, now set to 0, has changed. For the parameters to remain 0, they have to be excluded from the training. This is done by using the option FixedParameters.

To find the parameter numbers to be held fixed, search for the 0 in the parameter structure.

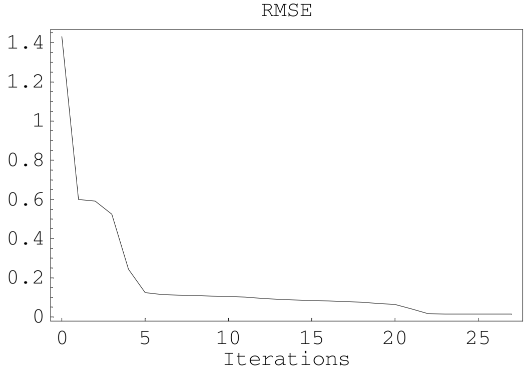
Find the parameters to be fixed and assign them to a variable.

```
In[12]:= fixparameters = Flatten[Position[Flatten[fdfrwrd[[1]]], 0]]

Out[12]= {3, 4}
```

Now the initialized FF network is ready to be trained. The two parameters to be held fixed are indicated in the FixedParameters option.

Train the network with some fixed parameters.

```
In[13]:= {fdfrwrd2, fitrecord} =
           NeuralFit[fdfrwrd, x, y, 40, FixedParameters → fixparameters];
```



Check that the two parameters have not changed.

```
In[14]:= fdfrwrd2[[1]]

Out[14]= {{{{-5.05883, -5.51404}, {0, 0}, {4.93321, -5.41119}},
           {{-1.60494}, {1.43627}, {1.56832}}}, {{-0.0476594}, {0.5}}}
```

The second row of w1 is still {0,0}. The fact that some parameters were fixed in the training is now specified as a rule in the second argument of the network object.

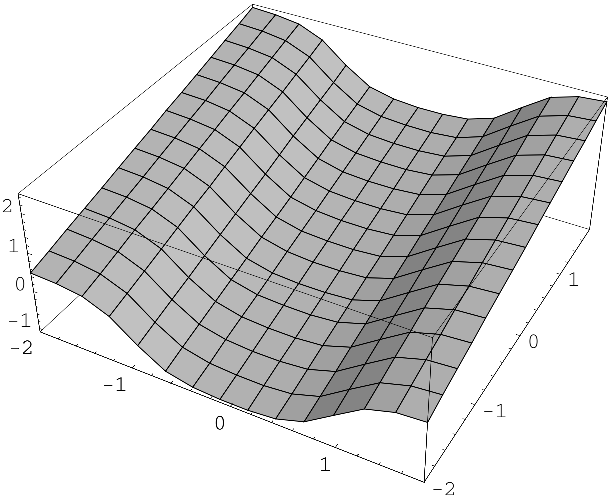
Fixed parameters are indicated as a rule in the network.

```
In[15]:= fdfrwrd2

Out[15]= FeedForwardNet[{{w1, w2}, χ}, {AccumulatedIterations → 27,
           CreationDate → {2002, 4, 3, 14, 56, 46}, Neuron → Sigmoid,
           FixedParameters → {3, 4}, OutputNonlinearity → None, NumberOfInputs → 2}]
```

If you submit the network `fdfrwrd2` to `NeuralFit` again for more training iterations, then the fixed parameters do not have to be specified because they are already indicated in the network. If you want to change the fixed parameters, simply indicate the new set of fixed parameters in the call to `NeuralFit`. This will override any specification in the initial model.

Complete this example with a plot of the estimated function. If you repeat the training without forcing the network to be linear in one direction, then the plot of the estimated function will probably not match the defined function as well as the following plot.

```
In[16]:= NetPlot[fdfrwrd2, x, y]
```



## 13.3 Select Your Own Neuron Function

The symbolic computation capability of *Mathematica* allows you to specify the neuron activation functions for RBF and FF networks. The *Neural Networks* package then uses standard *Mathematica* commands to compute the derivative, and the expression is optimized before the numerical computation begins in the training.

### 13.3.1 The Basis Function in an RBF Network

The Gaussian function is the most commonly used basis function in RBF networks. There is, however, no a priori reason for this choice, and you may use the Neuron option in the initialization to define a different basis function.

Recall that a general RBF network with *nb* basis functions is described by the following equation.

$$\hat{y}(\theta) = \sum_{i=1}^{nb} w_i^2 \, G\left(-\lambda_i^2 \, (x - w_i^1)^2\right) + b \tag{4}$$

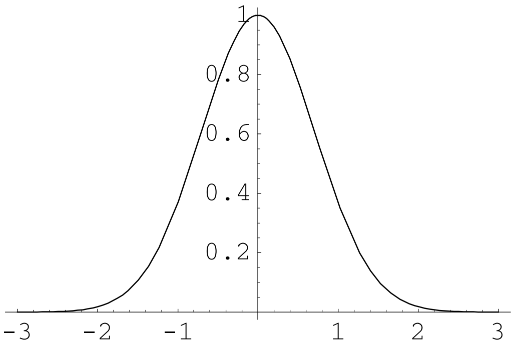
In addition to this expression, you also may have a parallel linear part as described in Section 2.5.2, Radial Basis Function Networks. The default value of G(x) is Exp so that the Gaussian function is obtained.

Load the *Neural Networks* package.

```
In[1]:=  << NeuralNetworks`
```

Look at the default basis function.

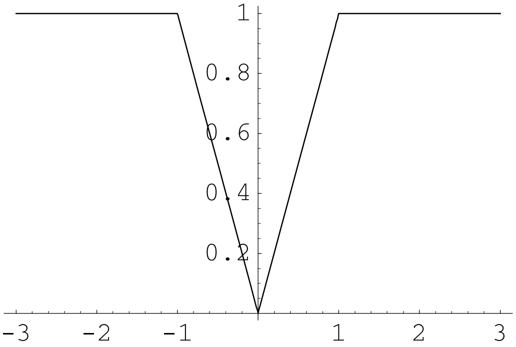
```
In[2]:=  Plot[Exp[-x^2], {x, -3, 3}, PlotRange → All]
```



Another possible choice for the basis function is one that decays linearly with the distance from the basis center. Such a choice can be obtained with the help of the `SaturatedLinear` function. Notice that `G` is a function of the distance squared in Equation 13.4. To make the basis function linear with respect to the distance, you must compensate for this square by introducing a square root. This is done in the following example.

Plot a linear saturated basis function.

```
In[3]:= Plot[SaturatedLinear[Sqrt[x^2]], {x, -3, 3}, PlotRange → All]
```

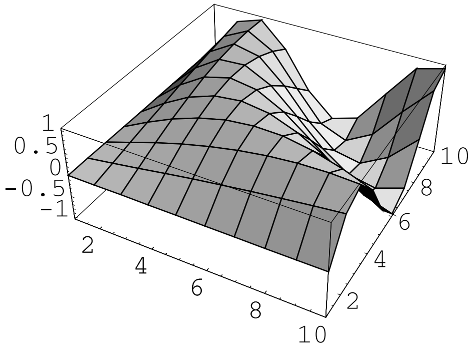


Since a layer of linear parameters is included in the network, in $w_2$, it does not matter that the basis function is inverted.

Now use this basis function in a small example.

Generate data and plot the function.

```
In[4]:= Ndata = 10;
        x1 = Table[N[{i / Ndata, j / Ndata}], {i, 0, Ndata - 1}, {j, 0, Ndata - 1}];
        y1 = Map[Sin[10. #[[1]] #[[2]]] &, x1, {2}];
        ListPlot3D[y1];
        x = Flatten[x1, 1];
        y = Transpose[{Flatten[y1]}];
```



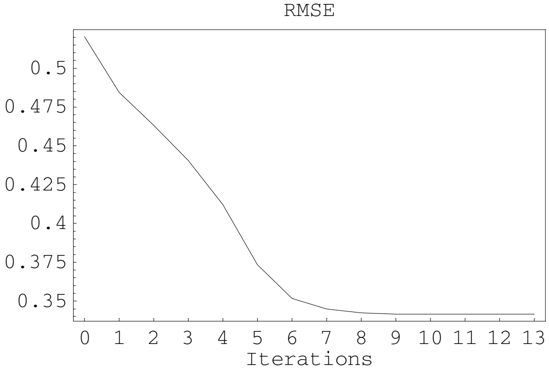
Initialize an RBF network with the proposed basis function.

```
In[10]:= rbf = InitializeRBFNet[x, y, 3, Neuron → Function[x, SaturatedLinear[Sqrt[-x]]]]

Out[10]= RBFNet[{{w1, λ, w2}, χ},
           {Neuron → Function[x, SaturatedLinear[√-x]], FixedParameters → None,
            AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 57, 17},
            OutputNonlinearity → None, NumberOfInputs → 2}]
```

Notice that a negative sign was used inside the square root to compensate for the negative sign in the function defined by Equation 13.4.

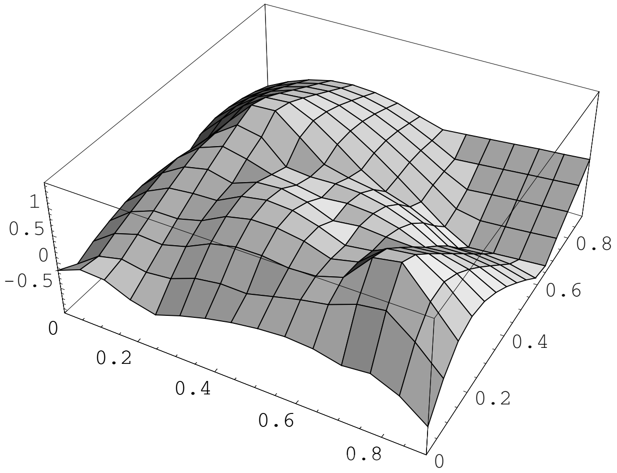
Train the RBF network using the data.

```
In[11]:= {rbf2, fitrecord} = NeuralFit[rbf, x, y, 20];
```



Plot the result.

```
In[12]:= NetPlot[rbf2, x, y, DataFormat → FunctionPlot]
```



You can vary the number of basis functions and repeat the example. You can also try different types of basis functions.

### 13.3.2 The Neuron Function in a Feedforward Network

You can set the neuron activation function to any smooth function. Some of the commonly used functions are demonstrated here.

Load the *Neural Networks* package.

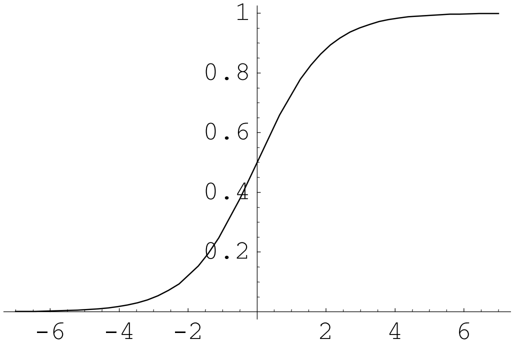
```
In[1]:= << NeuralNetworks`
```

`Sigmoid` is the default activation function.

$$\text{Sigmoid}[x] = \frac{1}{1 + e^{-x}} \qquad (5)$$

Plot the sigmoid.

```
In[2]:= Plot[Sigmoid[x], {x, -7, 7}]
```



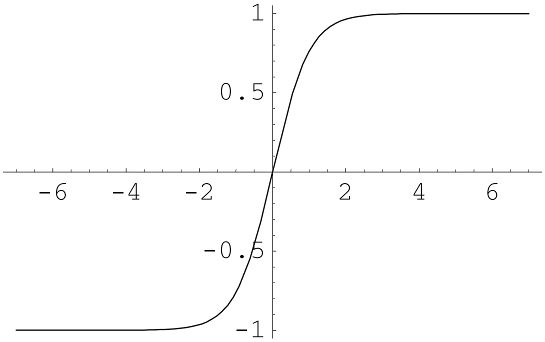
The sigmoid function is also recommended as the output of an FF network when you work on classification problems. Setting `OutputNonlinearity → Sigmoid` during initialization of an FF network will set the output of the network to `Sigmoid`. `Sigmoid` is used because it saturates to zero or one, which are the values used to indicate membership in a class. See Section 3.2 Package Conventions.

Another common choice of activation function is the hyperbolic tangent function.

$$\text{Tanh}[x] = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6}$$

Plot the hyperbolic tangent.

```
In[3]:= Plot[Tanh[x], {x, -7, 7}]
```

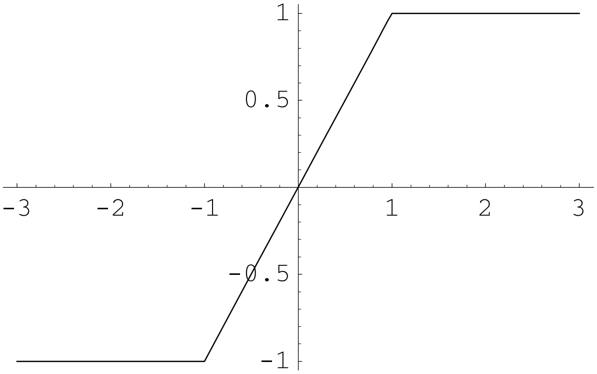


The hyperbolic tangent and sigmoid functions are equivalent when used in the hidden neurons because there is a similarity transformation for the parameters that takes an FF network with one of the activation functions to the other. It does, however, make a difference which function you apply to the output with the `OutputNonlinearity` option.

An interesting alternative neuron function is the saturated linear activation function. It is linear between −1 and 1 but saturates at these values for large positive or negative numbers.

Plot the `SaturatedLinear` function.

```
In[4]:= Plot[SaturatedLinear[x], {x, -3, 3}]
```

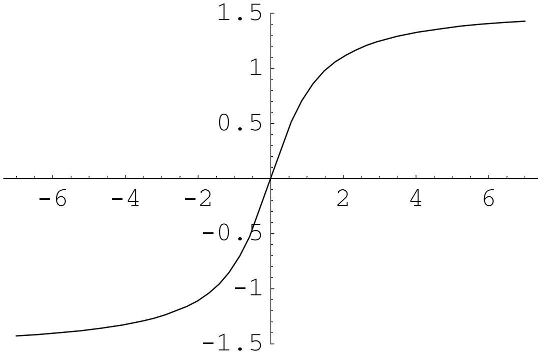


This activation function gives a local linear model that might be of interest in many situations.

Another possibility is the inverse tangent function.

Plot the inverse tangent.

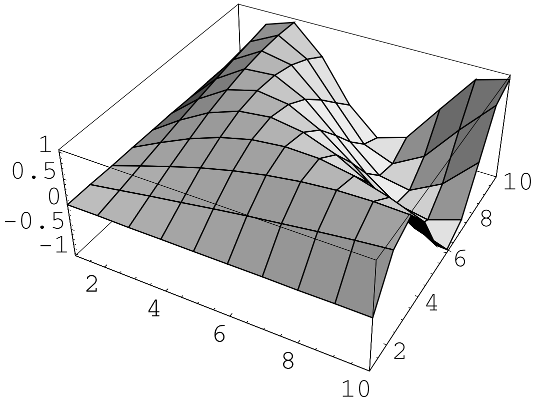
```
In[5]:= Plot[ArcTan[x], {x, -7, 7}]
```



The following demonstrates the use of the `SaturatedLinear` function in an FF network model.

Generate data and plot the actual function to be modeled.

```
In[6]:= Ndata = 10;
        x1 = Table[N[{i / Ndata, j / Ndata}], {i, 0, Ndata - 1}, {j, 0, Ndata - 1}];
        y1 = Map[Sin[10. #[[1]] #[[2]]] &, x1, {2}];
        ListPlot3D[y1];
        x = Flatten[x1, 1];
        y = Transpose[{Flatten[y1]}];
```

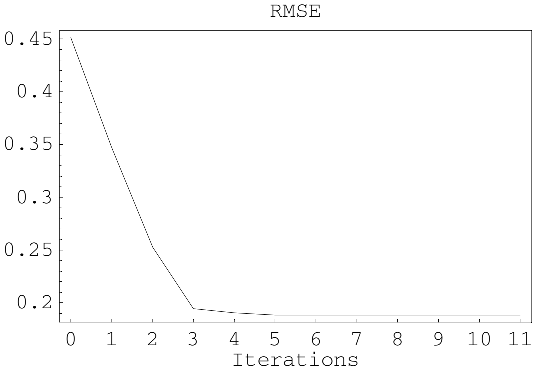


Initialize an FF network with the `SaturatedLinear` activation function.

```
In[12]:= fdfrwrd = InitializeFeedForwardNet[x, y, {4}, Neuron → SaturatedLinear]

Out[12]= FeedForwardNet[{{w1, w2}}, {Neuron → SaturatedLinear, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 57, 50},
           OutputNonlinearity → None, NumberOfInputs → 2}]
```

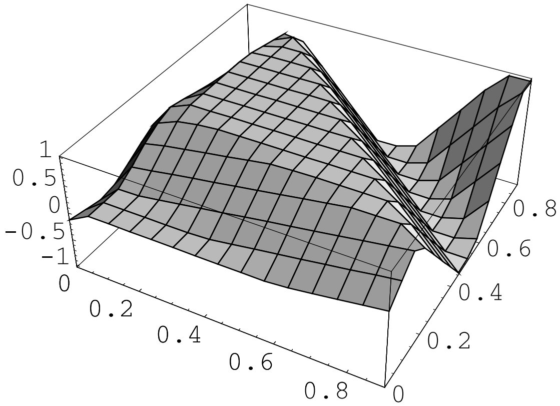
Fit the FF network to the data.

```
In[13]:= {fdfrwrd2, fitrecord} = NeuralFit[fdfrwrd, x, y, 30];
```



Plot the result.

```
In[14]:= NetPlot[fdfrwrd2, x, y, DataFormat → FunctionPlot]
```



The approximation obtained is locally linear, due to the special choice of activation function.

Try repeating this example with the other common functions. Edit the notebook and repeat the evaluations.

## 13.4 Accessing the Values of the Neurons

You might want to access the output values of the neurons from an FF or RBF network. If you just want a plot of the values, use `NetPlot` with the option `DataFormat → HiddenNeurons`. More direct access to the neuron values can be achieved by creating a second network with a slightly changed network structure, as shown in the following two examples.

### 13.4.1 The Neurons of a Feedforward Network

Load the *Neural Networks* package and some test data.

```
In[1]:= << NeuralNetworks`
        << one2twodimfunc.dat;
```

Initialize an FF network with four hidden neurons.

```
In[3]:= fdfrwrd = InitializeFeedForwardNet[x, y, {4}]

Out[3]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 58, 33},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

View the network information.

```
In[4]:= NetInformation[fdfrwrd]

Out[4]= FeedForward network created 2002-4-3 at 14:58. The
          network has 1 input and 2 outputs. It consists of 1 hidden
          layer with 4 neurons with activation function of Sigmoid type.
```

This network has two outputs.

In general, an FF network has the following structure:

$$\hat{y} = [\sigma([x\ 1]\ w_1)\ 1]\ w_2 \qquad (7)$$

To gain access to the output of the hidden neurons, you must turn them into the outputs of the network. This can be done using `NeuronDelete` to remove all the outputs of the network. The outputs of the new network will be the neurons of the hidden layer of the original network. This means that the matrix `w2` is removed. Consult Figures 2.5 and 2.6 in Section 2.5.1, Feedforward Neural Networks, to understand the equivalence of `w2` and the output layer. After the output layer is removed, the network is described by the following equation:

$$\hat{y} = \text{Sigmoid}\left([x\ 1]\ w_1\right) \tag{8}$$

The following command removes the output layer from the example network.

Remove the last layer of the network.

```
In[5]:=  newfdfrwrd = NeuronDelete[fdfrwrd, {{2, 1}, {2, 2}}]

          NeuronDelete::NewOutputLayer :
           All outputs have been deleted. The second-to-last layer becomes the new output.

Out[5]=  FeedForwardNet[{{w1}}, {Neuron → Sigmoid, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 58, 33},
           OutputNonlinearity → Sigmoid, NumberOfInputs → 1}]
```

```
In[6]:=  NetInformation[newfdfrwrd]

Out[6]=  FeedForward network created 2002-4-3 at 14:
           58. The network has 1 input and 4 outputs. It has no hidden
           layer. There is a nonlinearity at the output of type Sigmoid.
```

As predicted, the new network has four outputs and no hidden layer. The four hidden neurons outputs have become the outputs of the modified network.

The new FF network can now be applied to the data.

Values of the neurons when the network is applied to input data `{0.5}`.

```
In[7]:=  newfdfrwrd[{0.5}]

Out[7]=  {9.25426×10^-6, 0.978406, 0.99999, 0.999998}
```

The technique just described can also be used when you work with several hidden layers. To access the values of the hidden neurons in a specific layer, you must first turn this layer into an output layer. This is done by removing all layers of neurons after the one of interest to you.

### 13.4.2 The Basis Functions of an RBF Network

To access the values of the basis functions of an RBF network is slightly more complicated than for an FF network since the output layer cannot be removed. Instead, you can change the output layer to an identity mapping. This is described here.

Load the *Neural Networks* package and a *Mathematica* standard add-on package.

```
In[1]:= << NeuralNetworks`
        << LinearAlgebra`MatrixManipulation`
```

Load some test data.

```
In[3]:= << one2twodimfunc.dat;
```

Initialize an RBF network with four hidden neurons.

```
In[4]:= rbf = InitializeRBFNet[x, y, 4]

Out[4]= RBFNet[{{w1, λ, w2}, χ}, {Neuron → Exp, FixedParameters → None,
          AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 58, 52},
          OutputNonlinearity → None, NumberOfInputs → 1}]
```

It is convenient to express the RBF network using a less formal matrix notation:

$$\hat{y} = [G\,(\lambda^2\ |\ x^T - w_1\ |^2)\ 1]\cdot w_2 + x\cdot\chi \tag{9}$$

First the distances between the input vector $x$ and each column of $w_1$ are computed. These distances are multiplied with the square of their corresponding width in the vector $\lambda$ before the basis function $G$ is mapped over the vector. This forms the output of the hidden layer. A unit DC-level component is appended to the output of the hidden layer, and an inner product with $w_2$ gives the first term of the output. The second term is the linear submodel formed as an inner product between the inputs and the parameter matrix $\chi$.

You can access the values of the basis functions by changing the matrix $w_2$ to an identity matrix with dimension equal to the number of neurons plus a row of zeros for the DC-level parameters. You will then obtain a new RBF network with one output for each neuron. Also, if the original RBF network has a linear part, the linear part must be removed, and this can be done with `NeuronDelete`.

Check the number of neurons.

```
In[5]:=  NetInformation[rbf]

Out[5]=  Radial Basis Function network. Created 2002-4-3 at 14:
           58. The network has 1 input and 2 outputs.  It consists of 4
           basis functions of Exp type. The network has a linear submodel.
```

There are four neurons and this is the dimension of the identity mapping that is inserted.

Add a new matrix for `w2`.

```
In[6]:=  rbf[[1, 1, 3]] = AppendColumns[IdentityMatrix[4], {{0, 0, 0, 0}}]

Out[6]=  {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}, {0, 0, 0, 0}}
```

Delete the linear part.

```
In[7]:=  rbf = NeuronDelete[rbf, {0, 0}]

Out[7]=  RBFNet[{{w1, λ, w2}}, {Neuron → Exp, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 4, 3, 14, 58, 52},
           OutputNonlinearity → None, NumberOfInputs → 1}]
```

The newly obtained RBF network can be evaluated on input data. The output will be the values of the neurons of the original RBF network. Therefore, there will be one output for each neuron.

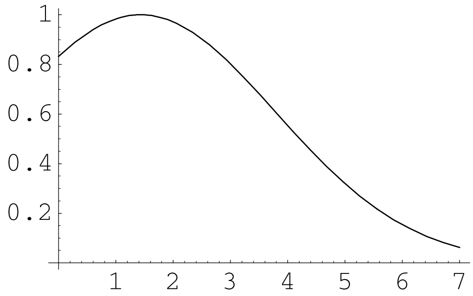
Check the values of the neurons for a numerical input value.

```
In[8]:=  rbf[{4.1}]

Out[8]=  {0.216709, 0.976699, 0.608883, 0.527926}
```

You can also plot the value of a single neuron.

Plot the fourth neuron.

```
In[9]:= Plot[rbf[{x}][[4]], {x, 0, 7}]
```



# Index