MATHEMATICA®

NEURAL NETWORKS

WOLFRAM
RESEARCH

August 2002
First edition
Intended for use with Mathematica 4

Software and manual written by: Jonas Sjoberg

Product managers: Yezabel Dooley and Kristin Kummer
Project managers: Julienne Davison and Julia Guelfi
Editors: Richard Martin, Jan Progen, Kristin Schar
Proofreaders: Rebecca Bigelow, Sam Daniel, Emilie Finn, Mary Jane Harshbarger, Marian Peden, Lynda Sherman
Package design by: Kara Wunderlich

Special thanks to the many alpha and beta testers and 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

Copyright © 1998–2002 Wolfram Research, Inc.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Wolfram Research, Inc.

Wolfram Research, Inc. is the holder of the copyright to the Neural Networks software and documentation ("Product") described in this document, including without limitation such aspects of the Product as its code, structure, sequence, organization, "look and feel", programming language, and compilation of command names. Use of the Product, unless pursuant to the terms of a license granted by Wolfram Research, Inc. or as otherwise authorized by law, is an infringement of the copyright.

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.

Mathematica is a registered trademark of Wolfram Research, Inc. All other trademarks used herein are the property of their respective owners. Mathematica is not associated with Mathematica Policy Research, Inc. or MathTech, Inc.

10 9 8 7 6 5 4 3 2 1

T4055 RCM 090602

Contents

Introduction

About *Neural Networks*

The *Neural Networks* package is a new *Mathematica* 4 application package designed to train, visualize, and validate neural network models. A neural network model is a structure that can be adjusted to map a set of the data's features or relationships. 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 can classify, estimate, predict, or simulate new data from the same or similar sources.

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 package builds on state-of-the-art algorithms with a user-friendly command structure. This makes it easy to get started with the package. Professionals and students with little or no background in neural networks will benefit from the comprehensive online tutorial and the large number of examples illustrating problems and possibilities in connection with neural network training. The tutorial will allow most new users to quickly acquire the knowledge and skills necessary to understand and solve many data-fitting problems. On the other hand, experienced neural network users will benefit from the package's extensive flexibility. The provided algorithms can be modified in many ways using the function options, and it is easy to extend the package with user-written training algorithms.

System Requirements

The *Neural Networks* application package requires *Mathematica* 4 or later. The package makes extensive use of many new *Mathematica* functions introduced in Version 4. The computational speed has been increased by introducing the following two functionalities: (1) the symbolic 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.*

About the Documentation

The *Neural Networks* package comes with extensive online documentation, easily accessible by means of the *Mathematica* Help Browser. It contains a tutorial chapter giving an introduction to neural networks and neural network algorithms. Each type of neural network is presented in an individual chapter that first defines the commands and then illustrates them with some simple examples. More advanced examples are placed in a special application chapter.

The Function Index serves as a reference tool for the package. For each type of neural network, the functions are listed alphabetically and described.

The Neural Networks palette gives a good overview of the supported neural network types, their functions, and their options. The palettes contain direct links to the documentation of each command and its options.

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. “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 “Loading the Package and Data”.

Loading the Package and Data

Neural Networks is one of many available *Mathematica* application packages, and it should be installed in the top-level *Mathematica* directory `AddOns/Applications`. 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, and 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 tell *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 *Mathematica* session.

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*. All illustration data in the documentation 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.

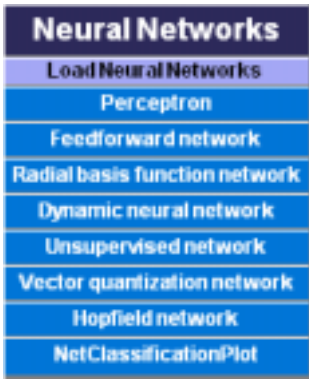
■ Load a data file.

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

Palettes

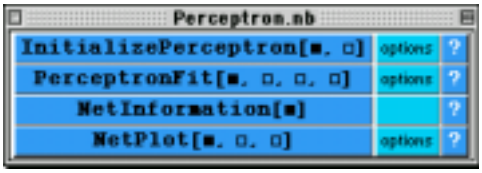
The *Neural Networks* package contains several palettes. As with the standard palettes that come with *Mathematica*, these palettes are available from the front end menu via the File ▹ Palettes command. These palettes provide you with an overview of the available commands plus their options, and give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

The main palette is called the Neural Networks palette. It 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. Following is the Perceptron palette.



The Perceptron palette.

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

Package Conventions

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

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[3]:= << NeuralNetworks`

In[4]:= << one2twodimfunc.dat;
```

In this data file, the input and output matrices are assigned to the variables `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[5]:= x

Out[5]= {{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[6]:= y

Out[6]= {{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[7]:= Dimensions[x]
        Dimensions[y]

Out[7]= {20, 1}

Out[8]= {20, 2}
```

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

■ Look at the input and output of data item 14.

```
In[9]:= x[[14]]
        y[[14]]

Out[9]= {6.5}

Out[10]= {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 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 referred to as *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, and the rest of the positions contain a 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 is 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 the *Neural Networks* package and a data set.

```
In[11]:= << NeuralNetworks`

In[12]:= << threeclasses.dat;
```

■ Look at the 25th input data sample.

```
In[13]:= x[[25]]

Out[13]= {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.

■ Determine the class in which child 25 belongs.

```
In[14]:= y[[25]]

Out[14]= {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 the following chapters and sections in the online documentation: Chapter 4, The Perceptron; Chapter 11, Vector Quantization; Section 12.1, Classification of Paper Quality; and Chapter 9, Hopfield Networks.

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 the indicated type
**Fit*	trains a neural network of the indicated type
NetPlot	illustrates a neural network 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 "Classification Problem Example" and "Function Approximation Example." More examples can also be found in the sections describing the different neural network types in the online documentation. In addition to these four commands, special commands for each neural network type are discussed in the chapter that focuses on each particular network.

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 rules indicating different network properties.

Consider this structure in a simple example.

■ Load the *Neural Networks* package.

```
In[15]:= << NeuralNetworks`
```

■ Create a perceptron network.

```
In[16]:= net = InitializePerceptron[{{1., -1.}, {-1., 1.}}, {1, 0}]

Out[16]= Perceptron[{w, b},
          {CreationDate → {2002, 3, 5, 10, 30, 50}, 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 rule containing `CreationDate` indicates when the network was created, and the other with `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[17]:= net[[1]]

Out[17]= {{{0.373725}, {0.940561}}, {1.78993}}
```

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

Out[18]= Perceptron[{w, b}, {NetworkName → JimsFavoriteModel,
           CreationDate → {2002, 3, 5, 10, 30, 50}, AccumulatedIterations → 0}]
```

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 “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.

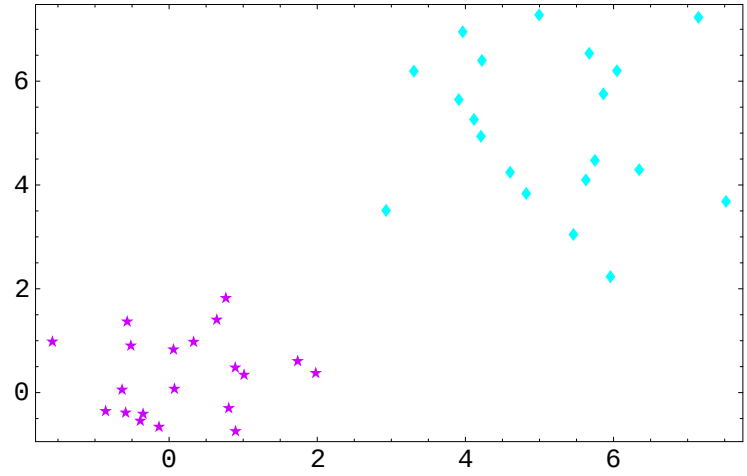
```
In[19]:= << NeuralNetworks`
```

```
In[20]:= << twoclasses.dat;
```

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

■ Make a two-dimensional plot of the data.

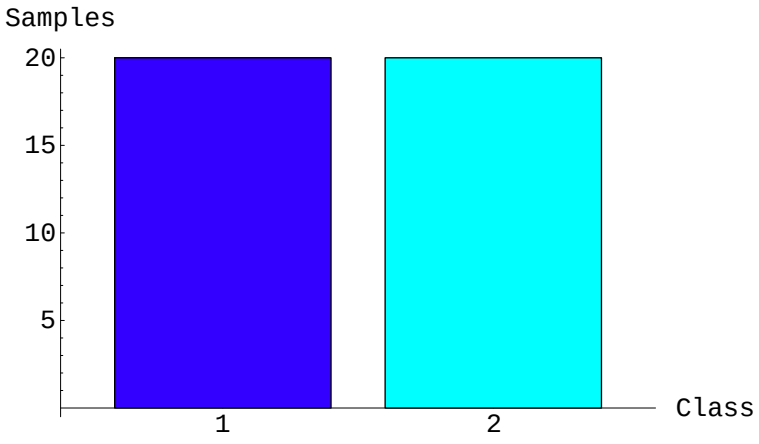
```
In[21]:= NetClassificationPlot[x, y, SymbolStyle → {Hue[0.5], Hue[0.8]}]
```



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

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

```
In[22]:= 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[23]:= << vqthreeclasses3D.dat;
```

■ Check the dimensionality of the input space.

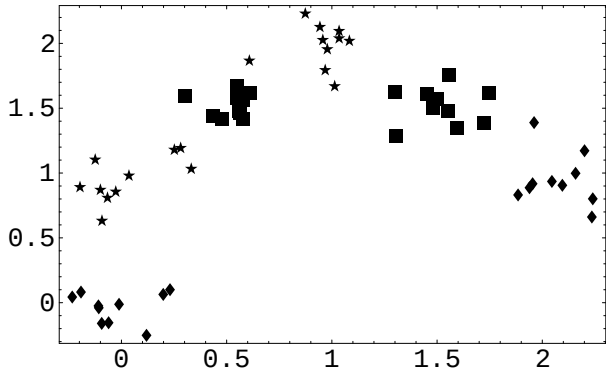
```
In[24]:= Dimensions[x]

Out[24]= {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 classes in `y`.

■ Project and plot the data.

```
In[25]:= NetClassificationPlot[x . {{1, 0}, {0, 1}, {0, 0}}, y]
```



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 the online documentation.

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[26]:= << NeuralNetworks`

In[27]:= << 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 “Data Format”.

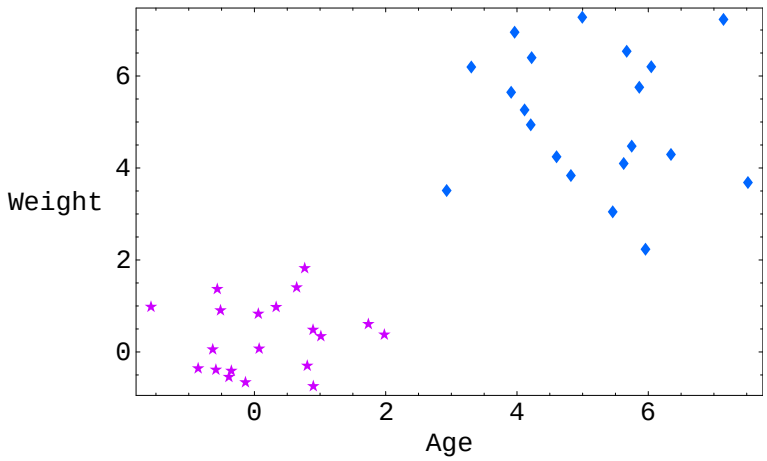
■ View the output data.

```
In[28]:= y

Out[28]= {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[29]:= NetClassificationPlot[x,y,FrameLabel→{"Age","Weight"},SymbolStyle→
         {Hue[0.6],Hue[0.8]}, RotateLabel→False]
```



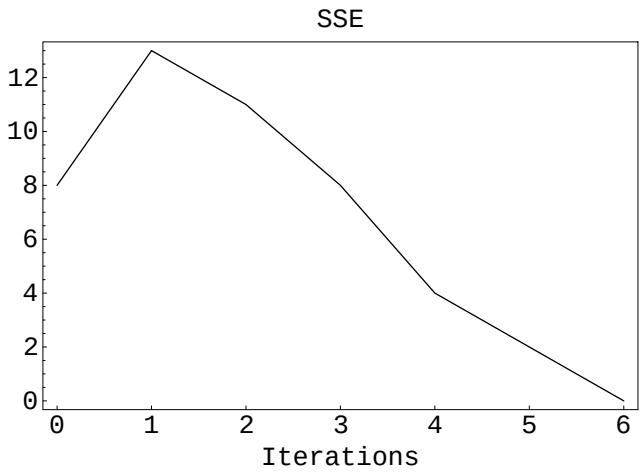
The plot clearly shows that the data is divided into two classes and that it should be possible to divide the groups with a curved line. This curve is found during the training process, and the successfully trained neural network classifier will assign each child to the correct group, given the child's 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[30]:= {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 the online documentation, Section 4.1.1, InitializePerceptron. As a result, 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. For a description of how the training record can be used to analyze the quality of training, see Section 7.8, The Training Record, in the online documentation. Also, for options that change the training parameters and plot, refer to Chapter 4, The Perceptron.

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[31]:= per[{6., 7.}]

Out[31]= {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.

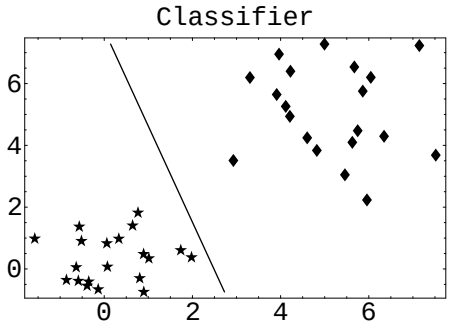
```
In[32]:= Clear[a, b];
         per[{a, b}]

Out[33]= {UnitStep[-69.4437 + 27.9143 a + 8.98818 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 straight 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.

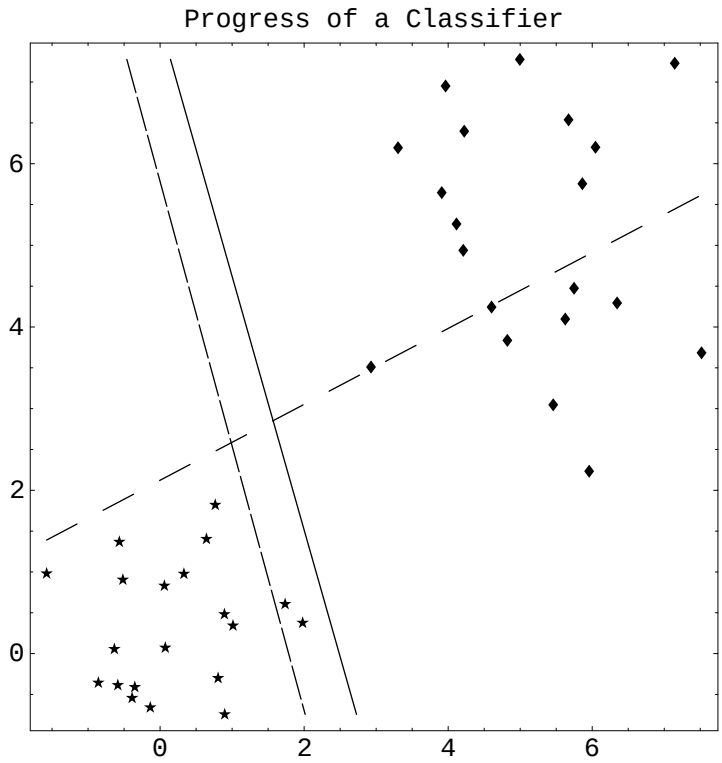
```
In[34]:= NetPlot[per, x, y]
```



`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[35]:= 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, in the online documentation.

Function Approximation Example

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

■ Load the *Neural Networks* package.

```
In[36]:= << NeuralNetworks`
```

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

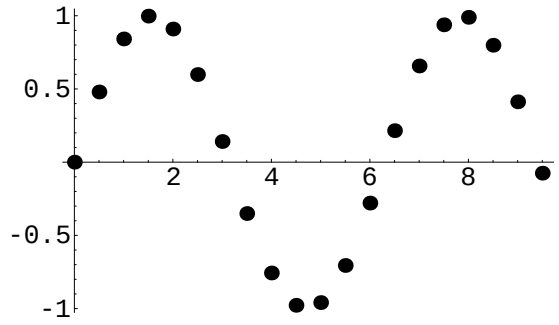
```
In[37]:= << onedimfunc.dat;
         << LinearAlgebra`MatrixManipulation`
```

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 of 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 that 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[39]:= 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.

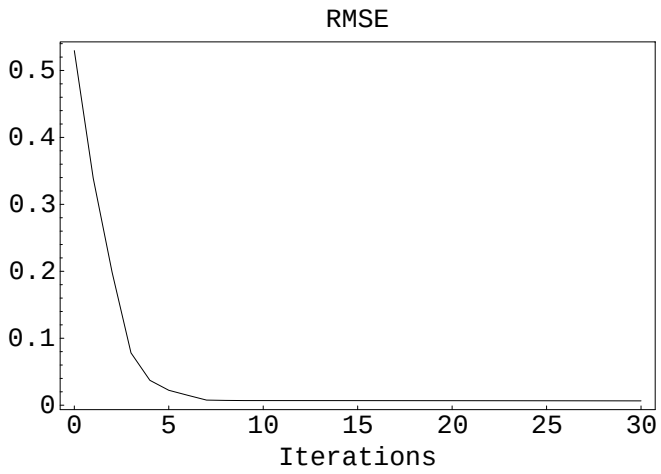
■ Initialize a feedforward network with three neurons.

```
In[40]:= fdfrwrd = InitializeFeedForwardNet[x, y, {3}]

Out[40]= FeedForwardNet[{{w1, w2}}, {Neuron → Sigmoid, FixedParameters → None,
           AccumulatedIterations → 0, CreationDate → {2002, 3, 5, 10, 31, 6},
           OutputNonlinearity → None, NumberOfInputs → 1}]
```

■ Train the initialized network.

```
In[41]:= {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.

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 root-mean-square error (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 in the online documentation. Here we will assume that the network has been successfully trained, though later we will 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. For a discussion of how to use the training record to analyze the quality of training, refer to Section 7.8, The Training Record, in the online documentation.

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[42]:= fdfrwrd2[{3}]

Out[42]= {0.13935}
```

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.

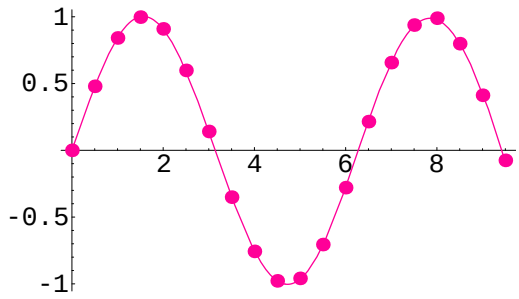
```
In[43]:= Clear[a];
         fdfrwrd2[{a}]
```

$$\text{Out[44]}= \left\{22.3873 + \frac{21.7706}{1 + e^{4.72221 - 0.770598\, a}} - \frac{53.2102}{1 + e^{1.10561 - 0.264303\, a}} - \frac{16.7422}{1 + e^{-0.233718 + 0.835793\, a}}\right\}$$

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 `NetPlot` to plot the neural network function.

■ Plot the function estimate together with the data.

```
In[45]:= 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.