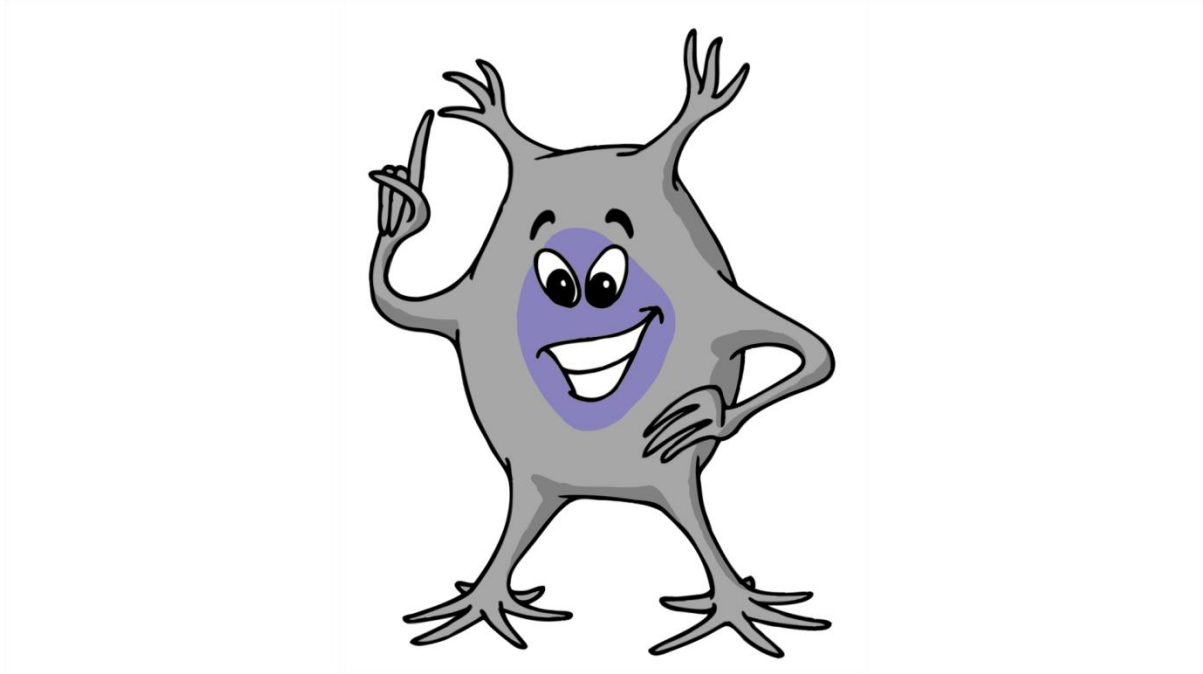
Introduction to Encog 2.3 for C#

Revision 1



http://www.heatonresearch.com/encog

A shortened version of:

"Programming Neural Networks with Encog 2 in C#". ISBN 1604390107. Your purchase of the entire book supports the Encog project! For more information visit.

http://www.heatonresearch.com/book/programming-neural-networks-encog-cs.html

Introduction to Encog 2.3 for C#

http://www.heatonresearch.com

This document serves as the primary documentation for the Encog Artificial Intelligence framework for C#. This document shows how to install, configure and use the basics of Encog.

For a more advanced, and lengthy, manual on Encog programming you may be interested in my book "Programming Neural Networks with Encog 2 in C#". ISBN 1604390107. This book is available in both paperback and ebook formats. This book adds many chapters and is nearly 500 pages long. Your purchase of our book supports the Encog project! For more information visit.

http://www.heatonresearch.com/book/programming-neural-networks-encog-cs.html

Contents

1 What is Encog?

Encog is an Artificial Intelligence (AI) Framework for Java and .Net. Though Encog supports several areas of AI outside of neural networks, the primary focus for the Encog 2.x versions is neural network programming. This book was published as Encog 2.3 was being released. It should stay very compatible with later editions of Encog 2. Future versions in the 2.x series will attempt to add functionality with minimal disruption to existing code.

1.1 The History of Encog

The first version of Encog, version 0.5 was released on July 10, 2008. However, the code for Encog originates from the first edition of "Introduction to Neural Networks with Java", which I published in 2005. This book was largely based on the Java Object Oriented Neural Engine (JOONE). Basing my book on JOONE proved to be problematic. The early versions of JOONE were quite promising, but JOONE quickly became buggy, with future versions introducing erratic changes that would frequently break examples in my book. As of 2010, with the writing of this book, the JOONE project seems mostly dead. The last release of JOONE was a "release candidate", that occurred in 2006. As of the writing of this document, in 2010, there have been no further JOONE releases.

The second edition of my book used 100% original code and was not based on any neural network API. This was a better environment for my "Introduction to Neural Networks for Java/C#" books, as I could give exact examples of how to implement the neural networks, rather than how to use an API. This book was released in 2008.

I found that many people were using the code presented in the book as a neural network API. As a result, I decided to package it as such. Version 0.5 of Encog is basically all of the book code combined into a package structure. Versions 1.0 through 2.0 greatly enhanced the neural network code well beyond what I would cover in an introduction book.

The goal of my "Introduction to Neural Networks with Java/C#" is to teach someone how to implement basic neural networks of their own. The goal of this book is to teach someone to use Encog to create more complex neural network structures without the need to know how the underlying neural network code actually works.

These two books are very much meant to be read in sequence, as I try not to repeat too much information in this book. However, you should be able to start with Encog if you have a basic understanding of what neural networks are used for. You must also understand the C# programming language. Particularly, you should be familiar with the following:

- C# Generics
- Collections
- Object Oriented Programming

Before we begin examining how to use Encog, let's first take a look at what sorts of problems Encog might be adept at solving. Neural networks are a programming technique. They are not a silver bullet solution for every programming problem you will encounter. There are some programming problems that neural networks are extremely adept at solving. There are other problems for which neural networks will fail miserably.

1.2 Problem Solving with Neural Networks

A significant goal of this book is to show you how to construct Encog neural networks and to teach you when to use them. As a programmer of neural networks, you must understand which problems are well suited for neural network solutions and which are not. An effective neural network programmer also knows which neural network structure, if any, is most applicable to a given problem. This section begins by first focusing on those problems that are not conducive to a neural network solution.

1.2.1 Problems Not Suited to a Neural Network Solution

Programs that are easily written out as flowcharts are examples of problems for which neural networks are not appropriate. If your program consists of well-defined steps, normal programming techniques will suffice.

Another criterion to consider is whether the logic of your program is likely to change. One of the primary features of neural networks is their ability to learn. If the algorithm used to solve your problem is an unchanging business rule, there is no reason to use a neural network. In fact, it might be detrimental to your application if the neural network attempts to find a better solution, and begins to diverge from the desired process and produces unexpected results.

Finally, neural networks are often not suitable for problems in which you must know exactly how the solution was derived. A neural network can be very useful for solving the problem for which it was trained, but the neural network cannot explain its reasoning. The neural network knows something because it was trained to know it. The neural network cannot explain how it followed a series of steps to derive the answer.

1.2.2 Problems Suited to a Neural Network

Although there are many problems for which neural networks are not well suited, there are also many problems for which a neural network solution is quite useful. In addition, neural networks can often solve problems with fewer lines of code than a traditional programming algorithm. It is important to understand which problems call for a neural network approach.

Neural networks are particularly useful for solving problems that cannot be expressed as a series of steps, such as recognizing patterns, classification, series prediction, and data mining.

Pattern recognition is perhaps the most common use for neural networks. For this type of problem, the neural network is presented a pattern. This could be an image, a

sound, or any other data. The neural network then attempts to determine if the input data matches a pattern that it has been trained to recognize. There will be many examples in this book of using neural networks to recognize patterns.

Classification is a process that is closely related to pattern recognition. A neural network trained for classification is designed to take input samples and classify them into groups. These groups may be fuzzy, lacking clearly defined boundaries. Alternatively, these groups may have quite rigid boundaries.

As you read though this document you will undoubtedly have questions about the Encog Framework. One of the best places to go for answers is the Encog forums at Heaton Research. You can find the Heaton Research forums at the following URL:

http://www.heatonresearch.com/forum

2 Installing and Using Encog

- Downloading Encog
- Running Examples
- Running the Workbench

This chapter shows how to install and use Encog. This consists of downloading Encog from the Encog Web site, installing and then running the examples. You will also be shown how to run the Encog Workbench. Encog makes use of .Net. This chapter assumes that you have already downloaded and installed the .Net runtime, as well as Visual Studio, or some other C# IDE.

Encog currently requires .Net Runtime 3.5 or higher. It is assumed that you are using Visual Studio 2008 or higher. Though Encog will likely work with other C# IDEs, such as SharpDevelop, this guide assumes you are using Visual Studio 2008. You can download a free copy of Visual Studio, if you meet certain requirements, from the following URL.

http://www.microsoft.com/exPress

We will begin with installing Encog.

Installing Encog

You can always download the latest version of Encog from the following URL:

http://www.encog.org

On this page, you will find a link to download the latest version of Encog and find the following files at the Encog download site:

- The Encog Core
- The Encog Examples
- The Encog Workbench
- The Encog Workbench Source Code

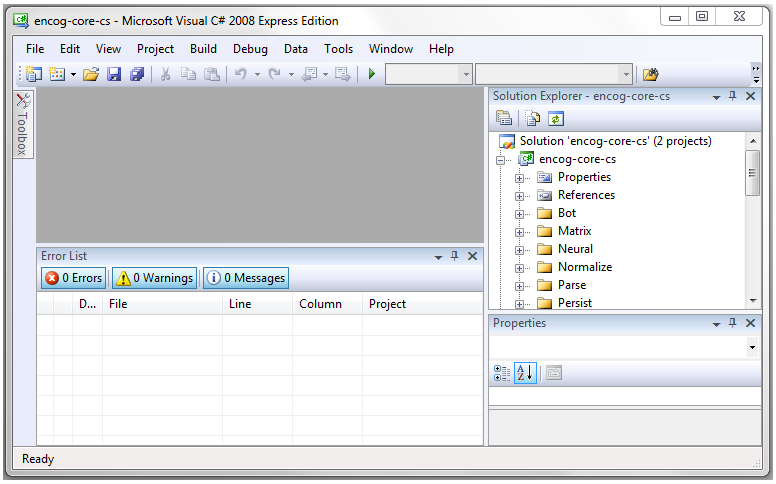
For this document, you will need to download the first three files (Encog Core, and Encog Examples and Encog Workbench). Make sure you select the .Net/C# version of Encog, as there is also a Java version of Encog.

There will be several versions of the workbench available. You can download the workbench as a Windows executable, a universal script file, or a Macintosh application. Choose the flavor of the workbench that is most suited to your computer. You do not need the workbench source code to use this document.

Compiling the Encog Core

Unless you would like to modify Encog itself, it is unlikely that you would need to compile the Encog core. Compiling the Encog core will recompile and rebuild the Encog core DLL file. It is very easy to recompile the Encog core using Visual Studio. Look inside of the the Encog core archive and you will find a Microsoft Solution (.sln) file named **encog-core-cs.sln**. Open this file and you will see Encog in Visual studio, as seen in Figure 2.1.

Figure 2.1: Encog Core in Visual Studio



To recompile the Encog DLL, simply choose "Build" and then "Rebuild Solution", as you would rebuild any project. Once the project has been rebuilt, you will find the new Encog core DLL file in the **bin\release** directory of your project.

Compiling and Executing Encog Examples

The Encog examples are packaged as one Microsoft Solution (.sln) file with one Microsoft Visual Studio Project (.csproj) file for each of the examples. You can execute these individual projects to run each example.

Most of the Encog examples are "console mode". This allows you to see how to create neural networks without a great deal of "GUI overhead." To run a console example you must execute the **ConsoleExamples** provided in the examples project. If you execute it with no arguments, you will see the following output.

```
Must specify the example to run as the first argument
The following commands are available:
adalinedigits
tsp-anneal
art1-classify
bam
benchmark
tsp-boltzmann
cpn
```

```
xor-elman
tsp-genetic
hopfield-simple
hopfield-associate
xor-jordan
market
multibench
xor-anneal
xor-backprop
xor-gaussian
xor-genetic
xor-manhattan
xor-radial
xor-rprop
xor-scg
xor-thresholdless
forest
lunar
image
persist-encog
persist-serial
sunspots
```

This gives you the commands you can use to run the individual examples. For example, to run the RPROP example, use the following command.

```
ConsoleExamples -pause xor-rprop
```

You can also tell the example to pause the output. This can be very useful when you execute from Visual Studio. Otherwise you will not see the example, because the window closes to fast.

```
ConsoleExamples -pause xor-rprop
```

Appendix B shows the locations of each of these examples.

Third-Party Libraries Used by Encog

There are only two external DLL files needed for the examples to run. They are contained in a directory named `dll`. These DLLs are summarized here.

- encog-core-cs.dll
- log4net.dll
- nunit.core.dll
- nunit.core.interfaces.dll
- nunit.uikit.dll

The first DLL is the Encog core. All projects that make use of Encog will use the Encog core. The second DLL is log4net. LOG4NET is a logging package used by many applications. LOG4NET is based on a similar package, named LOG4J that is used by

Encog Java. The last three DLLs are used by NUNIT, which provides unit testing for Encog.

Running an Example from the Command Line

Any of the Encog examples can be run from the command line. Simply navigate to the directory that contains the example's EXE file. Once inside of that directory execute the example as you would any application by typing its name and pressing the enter key.

Running Encog Examples in Visual Studio

The examples can also be run from an IDE, such as Visual Studio. The examples are all individual projects contained in one single solution file. Inside of the examples archive you will find the examples solution file, named **`EncogExamplesCS.sln`**. Once you open this file you will see individual project folders for all of the Encog example projects. Choose the project that you would like to run and right-click its folder. From the right-click popup menu choose "Set as Startup Project". This will cause the name of that project to appear bold.

Now that you have selected the project, and made it the startup project, you are ready to run an example. Click the "green arrow" on the tool bar and run the example as you would run an application under Visual Studio.

3 Introduction to Encog

- The Encog Framework
- What is a Neural Network?
- Using a Neural Network
- Training a Neural Network

Artificial neural networks are programming techniques that attempt to emulate the human brain's biological neural networks. Artificial neural networks (ANNs) are just one branch of artificial intelligence (AI). This document focuses primarily on artificial neural networks, frequently called simply neural networks, and the use of the Encog Artificial Intelligence Framework, usually just referred to as Encog. Encog is an open source project that provides neural network and HTTP bot functionality.

This document explains how to use neural networks with Encog and the C# programming language. Though this document focuses on C#, it could be used as a guide for other .Net language, such as VB.Net. Obviously, code contained in this document would need to be translated from C# to the .Net language of your choice.

The emphasis of this document is on how to use the neural networks, rather than how to actually create the software necessary to implement a neural network. Encog provides all of the low-level code necessary to construct many different kinds of neural networks. If you are interested in learning to actually program the internals of a neural network, using C#, you may be interested in the book "Introduction to Neural Networks with C#" (ISBN: 978-1604390094).

Encog provides the tools to create many different neural network types. Encog supports feedforward, recurrent, self-organizing maps, radial basis function and Hopfield neural networks. The low-level types provided by Encog can be recombined and extended to support additional neural network architectures as well. The Encog Framework can be obtained from the following URL:

http://www.encog.org/

Encog is released under the Lesser GNU Public License (LGPL). All of the source code for Encog is provided in a Subversion (SVN) source code repository provided by the Google Code project. Encog is also available for the Java and Silverlight platforms.

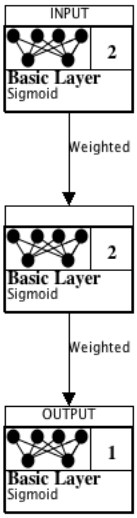
Encog neural networks, and related data, can be stored in .EG files. These files can be edited by a GUI editor provided with Encog. The Encog Workbench allows you to edit, train and visualize neural networks. The Encog Workbench can generate code in Java, Visual Basic or C#. The Encog Workbench can be downloaded from the above URL.

3.1 What is a Neural Network?

We will begin by examining what exactly a neural network is. A simple feedforward neural network can be seen in Figure 3.1. This diagram was created with the Encog Workbench. It is not just a diagram; this is an actual functioning neural network from

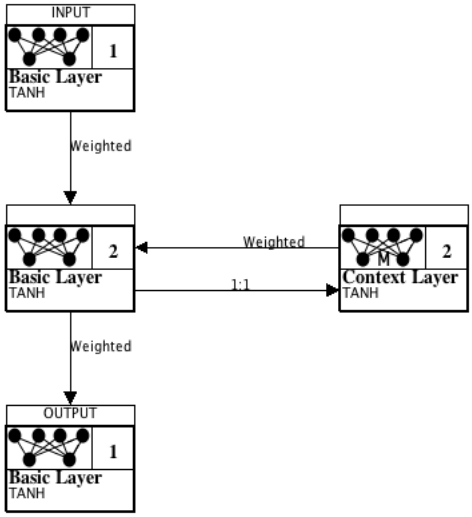
Encog as you would actually edit it.

Figure 3.1: Simple Feedforward Neural Network



Networks can also become more complex than the simple network above. Figure 3.2 shows a recurrent neural network.

Figure 3.2: Simple Recurrent Neural Network



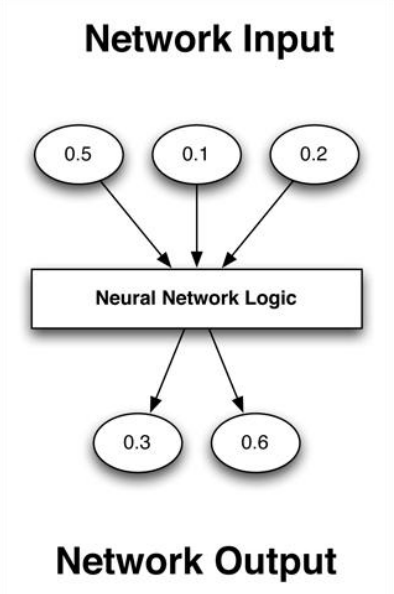
Looking at the above two neural networks you will notice that they are composed of layers, represented by the boxes. These layers are connected by lines, which represent synapses. Synapses and layers are the primary building blocks for neural networks created by Encog. The next chapter focuses solely on layers and synapses.

Before we learn to build neural networks with layers and synapses, let's first look at

what exactly a neural network is. Look at Figures 1.1 and 1.2. They are quite a bit different, but they share one very important characteristic. They both contain a single input layer and a single output layer. What happens between these two layers is very different, between the two networks. In this chapter, we will focus on what comes into the input layer and goes out of the output layer. The rest of the document will focus on what happens between these two layers.

Almost every neural network seen in this document will have, at a minimum, an input and output layer. In some cases, the same layer will function as both input and output layer. You can think of the general format of any neural network found in this document as shown in Figure 3.3.

Figure 3.3: Generic Form of a Neural Network



To adapt a problem to a neural network, you must determine how to feed the problem into the input layer of a neural network, and receive the solution through the output layer of a neural network. We will look at the input and output layers in this chapter. We will then determine how to structure the input and interpret the output. The input layer is where we will start.

3.1.1 Understanding the Input Layer

The input layer is the first layer in a neural network. This layer, like all layers, has a specific number of neurons in it. The neurons in a layer all contain similar properties. The number of neurons determines how the input to that layer is structured. For each input neuron, one **`double`** value is stored. For example, the following array could be used as input to a layer that contained five neurons.

```
double[] input = new double[5];
```

The input to a neural network is always an array of doubles. The size of this array

directly corresponds to the number of neurons on this layer. Encog uses the class **INeuralData** to hold these arrays. You could easily convert the above array into an **INeuralData** object with the following line of code.

```
INeuralData data = new BasicNeuralData(input);
```

The interface **INeuralData** defines any "array like" data that may be presented to Encog. You must always present the input to the neural network inside of an **INeuralData** object. The class **BasicNeuralData** implements the **INeuralData** interface. The class **BasicNeuralData** is not the only way to provide Encog with data. There are other implementations of **INeuralData**, as well. We will see other implementations later in the document.

The **BasicNeuralData** class simply provides a memory-based data holder for the neural network. Once the neural network processes the input, an **INeuralData** based class will be returned from the neural network's output layer. The output layer is discussed in the next section.

3.1.2 Understanding the Output Layer

The output layer is the final layer in a neural network. The output layer provides the output after all of the previous layers have had a chance to process the input. The output from the output layer is very similar in format to the data that was provided to the input layer. The neural network outputs an array of doubles.

The neural network wraps the output in a class based on the **INeuralData** interface. Most of the built in neural network types will return a **BasicNeuralData** class as the output. However, future, and third party, neural network classes may return other classes based other implementations of the **INeuralData** interface.

Neural networks are designed to accept input, which is an array of doubles, and then produce output, which is also an array of doubles. Determining how to structure the input data, and attaching meaning to the output, are two of the main challenges to adapting a problem to a neural network. The real power of a neural network comes from its pattern recognition capabilities. The neural network should be able to produce the desired output even if the input has been slightly distorted.

3.1.3 Hidden Layers

As previously discussed, neural networks contain and input layer and an output layer. Sometimes the input layer and output layer are the same. Often the input and output layer are two separate layers. Additionally, other layers may exist between the input and output layers. These layers are called hidden layers. These hidden layers can be simply inserted between the input and output layers. The hidden layers can also take on more complex structures.

The only purpose of the hidden layers is to allow the neural network to better produce the expected output for the given input. Neural network programming involves first

defining the input and output layer neuron counts. Once you have defined how to translate the programming problem into the input and output neuron counts, it is time to define the hidden layers.

The hidden layers are very much a "black box". You define the problem in terms of the neuron counts for the hidden and output layers. How the neural network produces the correct output is performed, in part, by the hidden layers. Once you have defined the structure of the input and output layers you must define a hidden layer structure that optimally learns the problem. If the structure of the hidden layer is too simple it may not learn the problem. If the structure is too complex, it will learn the problem but will be very slow to train and execute.

Later chapters in this document will discuss many different hidden layer structures. You will learn how to pick a good structure, based on the problem that you are trying to solve. Encog also contains some functionality to automatically determine a potentially optimal hidden layer structure. Additionally, Encog also contains functions to prune back an overly complex structure.

Some neural networks have no hidden layers. The input layer may be directly connected to the output layer. Further, some neural networks have only a single layer. A single layer neural network has the single layer self-connected. These connections permit the network to learn. Contained in these connections, called synapses, are individual weight matrixes. These values are changed as the neural network learns. We will learn more about weight matrixes in the next chapter.

3.2 Using a Neural Network

We will now look at how to structure a neural network for a very simple problem. We will consider creating a neural network that can function as an XOR operator. Learning the XOR operator is a frequent "first example" when demonstrating the architecture of a new neural network. Just as most new programming languages are first demonstrated with a program that simply displays "Hello World", neural networks are frequently demonstrated with the XOR operator. Learning the XOR operator is sort of the "Hello World" application for neural networks.

3.2.1 The XOR Operator and Neural Networks

The XOR operator is one of three commonly used Boolean logical operators. The other two are the AND and OR operators. For each of these logical operators, there are four different combinations. For example, all possible combinations for the AND operator are shown below.

```
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
```

This should be consistent with how you learned the AND operator for computer

programming. As its name implies, the AND operator will only return true, or one, when both inputs are true.

The OR operator behaves as follows.

```
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
```

This also should be consistent with how you learned the OR operator for computer programming. For the OR operator to be true, either of the inputs must be true.

The "exclusive or" (XOR) operator is less frequently used in computer programming, so you may not be familiar with it. XOR has the same output as the OR operator, except for the case where both inputs are true. The possible combinations for the XOR operator are shown here.

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

As you can see the XOR operator only returns true when both inputs differ. In the next section we will see how to structure the input, output and hidden layers for the XOR operator.

3.2.2 Structuring a Neural Network for XOR

There are two inputs to the XOR operator and one output. The input and output layers will be structured accordingly. We will feed the input neurons the following **double** values:

```
0.0,0.0
1.0,0.0
0.0,1.0
1.0,1.0
```

These values correspond to the inputs to the XOR operator, shown above. We will expect the one output neuron to produce the following **double** values:

```
0.0
1.0
1.0
0.0
```

This is one way that the neural network can be structured. This method allows a simple feedforward neural network to learn the XOR operator. The feedforward neural network, also called a perceptron, is one of the first neural network architectures that we will learn.

There are other ways that the XOR data could be presented to the neural network.

Later in this document we will see two examples of recurrent neural networks. We will examine the Elman and Jordan styles of neural networks. These methods would treat the XOR data as one long sequence. Basically concatenate the truth table for XOR together and you get one long XOR sequence, such as:

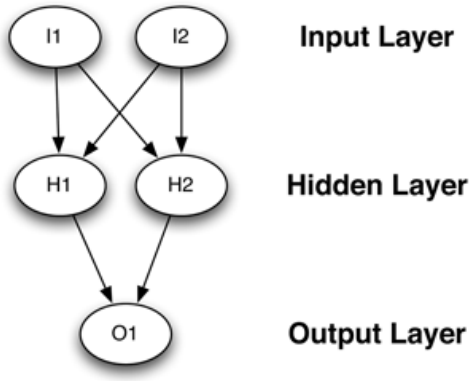
```
0.0,0.0,0.0,
0.0,1.0,1.0,
1.0,0.0,1.0,
1.0,1.0,0.0
```

The line breaks are only for readability. This is just treating XOR as a long sequence. By using the data above, the network would have a single input neuron and a single output neuron. The input neuron would be fed one value from the list above, and the output neuron would be expected to return the next value.

This shows that there is often more than one way to model the data for a neural network. How you model the data will greatly influence the success of your neural network. If one particular model is not working, you may need to consider another. For the examples in this document we will consider the first model we looked at for the XOR data.

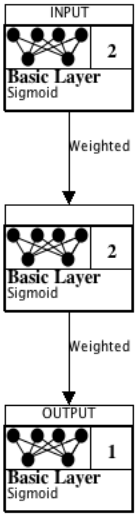
Because the XOR operator has two inputs and one output, the neural network will follow suit. Additionally, the neural network will have a single hidden layer, with two neurons to help process the data. The choice for 2 neurons in the hidden layer is arbitrary, and often comes down to trial and error. The XOR problem is simple, and two hidden neurons are sufficient to solve it. A diagram for this network can be seen in Figure 3.4.

Figure 3.4: Neuron Diagram for the XOR Network



Usually, the individual neurons are not drawn on neural network diagrams. There are often too many. Similar neurons are grouped into layers. The Encog workbench displays neural networks on a layer-by-layer basis. Figure 3.5 shows how the above network is represented in Encog.

Figure 3.5: Encog Layer Diagram for the XOR Network



The code needed to create this network is relatively simple.

```
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(2));
network.AddLayer(new BasicLayer(2));
network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
network.Reset();
```

In the above code you can see a **`BasicNetwork`** being created. Three layers are added to this network. The first layer, which becomes the input layer, has two neurons. The hidden layer is added second, and it has two neurons also. Lastly, the output layer is added, which has a single neuron. Finally, the **`FinalizeStructure`** method must be called to inform the network that no more layers are to be added. The call to **`Reset`** randomizes the weights in the connections between these layers.

Neural networks frequently start with a random weight matrix. This provides a starting point for the training methods. These random values will be tested and refined into an acceptable solution. However, sometimes the initial random values are too far off. Sometimes it may be necessary to reset the weights again, if training is ineffective.

These weights make up the long-term memory of the neural network. Additionally, some layers have threshold values that also contribute to the long-term memory of the neural network. Some neural networks also contain context layers, which give the neural network a short-term memory as well. The neural network learns by modifying these weight and threshold values.

Now that the neural network has been created, it must be trained. Training is discussed in the next section.

3.2.3 Training a Neural Network

To train the neural network, we must construct a **INeuralDataSet** object. This object contains the inputs and the expected outputs. To construct this object, we must create two arrays. The first array will hold the input values for the XOR operator. The second array will hold the ideal outputs for each of 115 corresponding input values. These will correspond to the possible values for XOR. To review, the four possible values are as follows:

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

First we will construct an array to hold the four input values to the XOR operator. This is done using a two dimensional **double** array. This array is as follows:

```
public static double[][] XOR_INPUT ={
  new double[2] { 0.0, 0.0 },
  new double[2] { 1.0, 0.0 },
  new double[2] { 0.0, 1.0 },
  new double[2] { 1.0, 1.0 } };
```

Likewise, an array must be created for the expected outputs for each of the input values. This array is as follows:

```
public static double[][] XOR_IDEAL = {
  new double[1] { 0.0 },
  new double[1] { 1.0 },
  new double[1] { 1.0 },
  new double[1] { 0.0 } };
```

Even though there is only one output value, we must still use a two-dimensional array to represent the output. If there had been more than one output neuron, there would have been additional columns in the above array.

Now that the two input arrays have been constructed an **INeuralDataSet** object must be created to hold the training set. This object is created as follows.

```
INeuralDataSet trainingSet = new BasicNeuralDataSet(XOR_INPUT,
XOR_IDEAL);
```

Now that the training set has been created, the neural network can be trained. Training is the process where the neural network's weights are adjusted to better produce the expected output. Training will continue for many iterations, until the error rate of the network is below an acceptable level. First, a training object must be created. Encog supports many different types of training.

For this example we are going to use Resilient Propagation (RPROP). RPROP is perhaps the best general-purpose training algorithm supported by Encog. Other training techniques are provided as well, as certain problems are solved better with certain training techniques. The following code constructs a RPROP trainer.

```
ITrain train = new ResilientPropagation(network, trainingSet);
```

All training classes implement the **ITrain** interface. The RPROP algorithm is implemented by the **ResilientPropagation** class, which is constructed above.

Once the trainer has been constructed the neural network should be trained. Training the neural network involves calling the **Iteration** method on the **ITrain** class until the error is below a specific value.

```
int epoch = 1;

do
{
  train.Iteration();
  Console.WriteLine("Epoch #" + epoch + " Error:" + train.Error);
  epoch++;
} while ((epoch < 5000) && (train.Error > 0.01));
```

The above code loops through as many iterations, or epochs, as it takes to get the error rate for the neural network to be below 1%. Once the neural network has been trained, it is ready for use. The next section will explain how to use a neural network.

3.2.4 Executing a Neural Network

Making use of the neural network involves calling the **Compute** method on the **BasicNetwork** class. Here we loop through every training set value and display the output from the neural network.

```
Console.WriteLine("Neural Network Results:");

foreach (INeuralDataPair pair in trainingSet)
{
  INeuralData output = network.Compute(pair.Input);
  Console.WriteLine(pair.Input[0] + "," + pair.Input[1]
  + ", actual=" + output[0] + ",ideal=" + pair.Ideal[0]);
}
```

The **Compute** method accepts an **INeuralData** class and also returns a **INeuralData** object. This contains the output from the neural network. This output is displayed to the user. With the program is run the training results are first displayed. For each Epoch, the current error rate is displayed.

```
Epoch #1 Error:0.5604437512295236
Epoch #2 Error:0.5056375155784316
Epoch #3 Error:0.5026960720526166
Epoch #4 Error:0.4907299498390594
...
Epoch #104 Error:0.01017278345766472
Epoch #105 Error:0.010557202078697751
Epoch #106 Error:0.011034965164672806
Epoch #107 Error:0.009682102808616387
```

The error starts at 56% at epoch 1. By epoch 107 the training has dropped below 1% and training stops. Because neural network was initialized with random weights, it may take different numbers of iterations to train each time the program is run. Additionally, though the final error rate may be different, it should always end below 1%.

Finally, the program displays the results from each of the training items as follows:

```
Neural Network Results:
0.0,0.0, actual=0.002782538818034049,ideal=0.0
1.0,0.0, actual=0.9903741937121177,ideal=1.0
0.0,1.0, actual=0.9836807956566187,ideal=1.0
1.0,1.0, actual=0.0011646072586172778,ideal=0.0
```

As you can see, the network has not been trained to give the exact results. This is normal. Because the network was trained to 1% error, each of the results will also be within generally 1% of the expected value.

Because the neural network is initialized to random values, the final output will be different on second run of the program.

```
Neural Network Results:
0.0,0.0, actual=0.005489822214926685,ideal=0.0
1.0,0.0, actual=0.985425090860287,ideal=1.0
0.0,1.0, actual=0.9888064742994463,ideal=1.0
1.0,1.0, actual=0.005923146369557053,ideal=0.0
```

Above, you see a second run of the program. The output is slightly different. This is normal.

This is the first Encog example. You can see the complete program in Listing 1.1. All of the examples contained in this document are also included with the examples downloaded with Encog. For more information on how to download these examples and where this particular example is located, refer to Appendix A, "Installing Encog".

Listing 1.1: Solve XOR with RPROP

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Layers;
using Encog.Neural.Activation;
using Encog.Neural.Data.Basic;
using Encog.Neural.NeuralData;
using Encog.Neural.Networks.Training;
using Encog.Neural.Data;
using Encog.Neural.Networks.Training.Propagation.Resilient;
using ConsoleExamples.Examples;

namespace Encog.Examples.XOR.Resilient
{
```

```
public class XORResilient : IExample
{
  public static ExampleInfo Info
  {
    get
    {
      ExampleInfo info = new ExampleInfo(
        typeof(XORResilient),
          "xor-rprop",
          "XOR Operator with Resilient Propagation",
          "Use RPROP to learn the XOR operator.");
         return info;
    }
  }

  /// <summary>
  /// Input for the XOR function.
  /// </summary>
  public static double[][] XOR_INPUT ={
    new double[2] { 0.0, 0.0 },
    new double[2] { 1.0, 0.0 },
    new double[2] { 0.0, 1.0 },
    new double[2] { 1.0, 1.0 } };

  /// <summary>
  /// Ideal output for the XOR function.
  /// </summary>
  public static double[][] XOR_IDEAL = {
    new double[1] { 0.0 },
    new double[1] { 1.0 },
    new double[1] { 1.0 },
    new double[1] { 0.0 } };

  /// <summary>
  /// Program entry point.
  /// </summary>
  /// <param name="args">Not used.</param>
  public void Execute(IExampleInterface app)
  {
    BasicNetwork network = new BasicNetwork();
    network.AddLayer(new BasicLayer(
      new ActivationSigmoid(), true, 2));
    network.AddLayer(new BasicLayer(
      new ActivationSigmoid(), true, 6));
    network.AddLayer(new BasicLayer(
      new ActivationSigmoid(), true, 1));
    network.Structure.FinalizeStructure();
    network.Reset();

    INeuralDataSet trainingSet =
      new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

    // train the neural network
    // train the neural network
    ITrain train =
```

```
        new ResilientPropagation(network, trainingSet);

      int epoch = 1;

      do
      {
        train.Iteration();
        Console.WriteLine("Epoch #" + epoch
          + " Error:" + train.Error);
        epoch++;
      } while ((epoch < 5000) && (train.Error > 0.001));

      // test the neural network
      Console.WriteLine("Neural Network Results:");
      foreach (INeuralDataPair pair in trainingSet)
      {
        INeuralData output = network.Compute(pair.Input);
        Console.WriteLine(pair.Input[0] + "," + pair.Input[1]
          + ", actual=" + output[0] + ",ideal=" + pair.Ideal[0]);
      }
    }
  }
}
```

3.3 Chapter Summary

Encog is a framework that allows you to create neural networks or bot applications. This document focuses on using Encog to create neural network applications. This chapter focused on the overall layout of a neural network. In this chapter, you saw how to create an Encog application that could learn the XOR operator.

Neural networks are made up of layers. These layers are connected by synapses. The synapses contain weights that make up the memory of the neural network. Some layers also contain threshold values that also contribute to the memory of the neural network. Together, thresholds and weights make up the long-term memory of the neural network. Networks can also contain context layers. Context layers are used to form a short-term memory.

There are several different layer types supported by Encog. However, these layers fall into three groups, depending on where they are placed in the neural network. The input layer accepts input from the outside. Hidden layers accept data from the input layer for further processing. The output layer takes data, either from the input or final hidden layer, and presents it on to the outside world.

The XOR operator was used as an example for this chapter. The XOR operator is frequently used as a simple “Hello World” application for neural networks. The XOR operator provides a very simple pattern that most neural networks can easily learn. It is important to know how to structure data for a neural network. Neural networks both accept and return an array of floating point numbers.

This chapter introduced layers and synapses. You saw how they are used to construct

a simple neural network. The next chapter will greatly expand on layers and synapses. You will see how to use the various layer and synapse types offered by Encog to construct neural networks.

4 Using Activation Functions

- What are Layers and Synapses?
- Encog Layer Types
- Encog Synapse Types
- Neural Network Properties
- Neural Network Logic
- Building with Layers and Synapses

Encog neural networks are made up of layers, synapses, properties and a logic definition. In this chapter we will examine the various types of layers and synapses supported by Encog. You will see how the layer and synapse types can be combined to create a variety of neural network types.

4.1 What are Layers and Synapses?

A layer is a collection of similar neurons. All neurons in the layer must share exactly the same characteristic as other neurons in this layer. A layer accepts a parameter that specifies how many neurons that layer is to have. Layers hold an array of threshold values. There is one threshold value for each of the neurons in the layer. The threshold values, along with the weight matrix form the long-term memory of the neural network. Some layers also hold context values that make up the short-term memory of the neural network.

A synapse is used to connect one layer to another. The synapses contain the weight matrixes used by the neural network. The weight matrixes hold the connection values between each of the neurons in the two layers that are connected by this synapse.

Every Encog neural network contains a neural logic class. This class defines how a neural network processes its layers and synapses. A neural logic class must implement the `INeuralLogic` interface. Every Encog neural network must have an `INeuralLogic` based logic class. Without such a class the network would not be able to process incoming data. `INeuralLogic` classes allow Encog to be compatible with a wide array of neural network types.

Some `INeuralLogic` classes require specific layer types. For the `INeuralLogic` classes to find these layers, the layers must be tagged. Tagging allows a type to be assigned to any layer in the neural network. Not all layers need to be tagged.

Neural network properties are stored in a collection of name-value pairs. They are stored in a simple `IDictionary` structure. Some `INeuralLogic` classes require specific parameters to be set for them to operate. These parameters are stored in the neural network properties.

Neural networks are constructed of layers and synapses. There are several different types of layers and synapses, provided by Encog. This chapter will introduce all of the Encog layer types and synapse types. We will begin by examining the Encog layer types.

4.2 Understanding Encog Layers

There are a total of three different layer types used by Encog. In this section we will examine each of these layer types. All three of these layer types implement the **ILayer** interface. As additional layer types are added to Encog, they will support the **ILayer** interface as well. We will begin by examining the **ILayer** interface.

4.2.1 Using the Layer Interface

The **ILayer** interface defines many important methods that all layers must support. Additionally, most Encog layers implement a constructor that initializes that unique type of layer. Listing 4.1 shows the **ILayer** interface.

Listing 4.1: The Layer Interface

```
public interface ILayer : ICloneable, IEncogPersistedObject
{
  void AddNext(ILayer next);
  void AddNext(ILayer next, SynapseType type);
  void AddSynapse(ISynapse synapse);
  INeuralData Compute(INeuralData pattern);

  IActivationFunction ActivationFunction
  {
    get;
    set;
  }

  int NeuronCount
  {
    get;
    set;
  }

  IList<ISynapse> Next
  {
    get;
  }

  ICollection<ILayer> NextLayers
  {
    get;
  }

  double[] Threshold
  {
    get;
    set;
  }

  int X
  {
    get;
    set;
```

```
  }


  int Y
  {
    get;
    set;
  }

  int ID
  {
    get;
    set;
  }

  bool HasThreshold
  {
    get;
  }

  bool IsConnectedTo(ILayer layer);
  void Process(INeuralData pattern);
  INeuralData Recur();
}
```

As you can see there are a number of methods that must be implemented to create a layer. We will now review some of the more important methods.

The **AddNext** method is used to connect another layer to this one. The next layer is connected with an **ISynapse**. There are two overloads to the **AddNext** method. The first allows you to simply specify the next layer and a **WeightedSynapse** is automatically created to connect the new layer. The second allows you to specify the next layer and use the **SynapseType** enumeration to specify what type of synapse you would like to connect the two layers. Additionally, the **AddSynapse** method allows you to simply pass in an already created **ISynapse**.

The **Next** property can be called to get an **IList** of the **ISynapse** objects used to connect to the next layers. Additionally the **NextLayers** property can be used to determine which layers this **ILayer** is connected to. To see if this **Layer** is connected to another specific **ILayer** call the **IsConnectedTo** method.

The **Threshold** property allows access to the threshold values for this layer. The threshold values are numeric values that change as the neural network is trained, together with the weight matrix values; they form the long-term memory of the neural network. Not all layers have threshold values; the **HasThreshold** property can be used to determine if a layer has threshold values.

The **Activation** property allows access to the activation function. Activation functions are mathematical functions that scale the output from a neuron layer. Encog supports many different activation functions. Activation functions will be covered in

much greater detail in the next chapter.

Finally, the **Compute** method is provided that applies the activation function and does any other internal processing necessary to **Compute** the output from this layer. You will not usually call **Compute** directly, rather you will call the **Compute** method on the **INetwork** that this layer is attached to, and it will call the appropriate **Compute** functions for its various layers.

4.2.2 Using the Basic Layer

The **BasicLayer** implements the **ILayer** interface. The **BasicLayer** class has two primary purposes. First, many types of neural networks can be built completely from **BasicLayer** objects, as it is a very useful layer in its own right. Second, the **BasicLayer** provides the basic functionality that some other layers require. As a result, some of the other layers in Encog subclass are base on the **BasicLayer** class.

The most basic form of the **BasicLayer** constructor accepts a single integer parameter that specifies how many neurons this layer will have. This constructor creates a layer that uses threshold values and the hyperbolic tangent function as the activation function. For example, the following code creates three layers, with varying numbers of neurons.

```
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(2));
network.AddLayer(new BasicLayer(6));
network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
network.Reset();
```

If you would like more control over the layer, you can use a more advanced constructor. The following constructor allows you to specify the activation function, as well as if threshold values should be used.

```
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(new ActivationSigmoid(),true,2));
network.AddLayer(new BasicLayer(new ActivationSigmoid(),true,6));
network.AddLayer(new BasicLayer(new ActivationSigmoid(),true,1));
network.Structure.FinalizeStructure();
network.Reset();
```

The above code creates the same sort of network as the previous code segment; however, a sigmoid activation function is used. The **true** parameter means that threshold values should be used. Some neural network architectures do not use threshold values, while others do. As you progress through this document you will see both networks that use, as well as those that do not use threshold values.

The **BasicLayer** class is used for many neural network types in this document.

4.2.3 Using the Context Layer

The **ContextLayer** class implements a contextual layer. This layer allows the neural network to have a short-term memory. The context layer always remembers the last input values that were fed to it. This causes the context layer to always output what it originally received on the previous run of the neural network. This allows the neural network to remember the last data that was fed to it on the previous run. The context layer is always one iteration behind.

Context layers are usually used with a recurrent neural network. Recurrent neural networks do not feed the layers just forward. Layers will be connected back into the flow of the neural network.

```
ILayer context = new ContextLayer(2);
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(1));
network.AddLayer(hidden = new BasicLayer(2));
hidden.AddNext(context, SynapseType.OneToOne);
context.AddNext(hidden);

network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
```

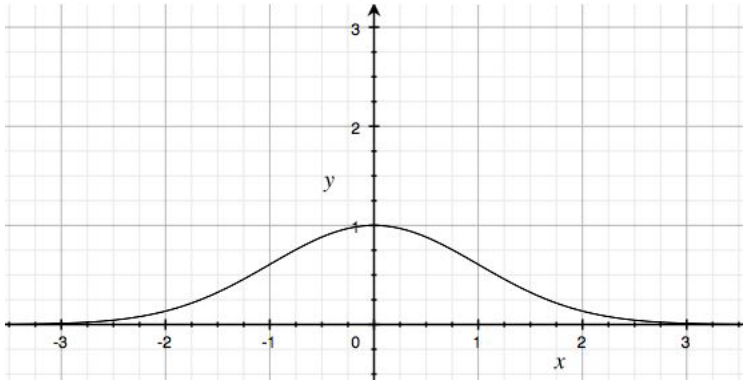
The above code shows a **ContextLayer** used with regular **BasicLayer** objects. The output from the hidden layer in the above neural network not only goes to the output layer. The output from the hidden layer also is fed into the **ContextLayer**. A **OneToOneSynapse** is used to feed the **ContextLayer**. We simply want the context layer to remember the output from the hidden layer; we do not want any processing. A **WeightedSynapse** is fed out of the **ContextLayer** because we do want additional processing. We want the neural network to learn from the output of the **ContextLayer**.

These features allow the **ContextLayer** to be very useful for recognizing sequences of input data. The patterns are no longer mutually exclusive when you use a **ContextLayer**. If "Pattern A" is presented to the neural network, followed by "Pattern B", it is much different than "Pattern B" being presented first. Without a context layer, the order would not matter.

4.2.4 Using the Radial Basis Function Layer

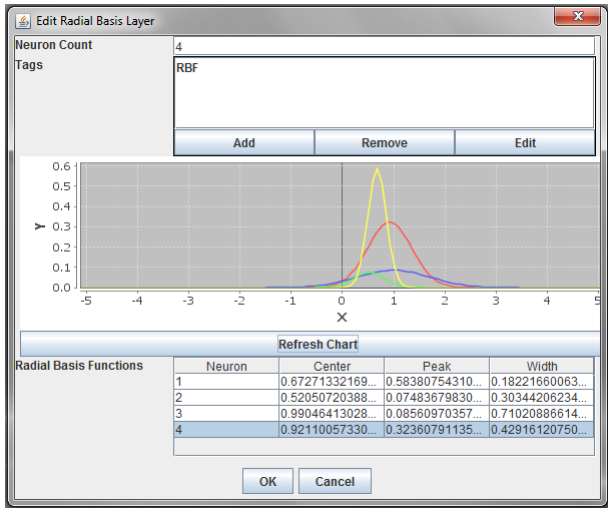
The **RadialBasisFunctionLayer** object implements a radial basis function (RBF) layer. This layer type is based on one or more RBF functions. A radial basis function reaches a peak and decreases quickly on both sides of the graph. One of the most common radial basis functions is the Gaussian function. The Gaussian function is the default option for the **RadialBasisFunctionLayer** class. You can see the Gaussian function in Figure 4.1.

Figure 4.1: The Gaussian Function



The above figure shows a graph of the Gaussian function. Usually several Gaussian functions are combined to create a **RadialBasisFunctionLayer**. Figure 4.2 shows a **RadialBasisFunctionLayer** being edited in the Encog Workbench. Here you can see that this layer is made up of multiple Gaussian functions.

Figure 4.2: An RBF Layer in Encog Workbench



The following code segment shows the **RadialBasisFunctionLayer** as part of a RBF neural network.

```
RadialBasisFunctionLayer rbfLayer;
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(new ActivationLinear(),false,2));

network.AddLayer(rbfLayer = new RadialBasisFunctionLayer(4),
SynapseType.Direct);
network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
network.Reset();
rbfLayer.RandomizeGaussianCentersAndWidths(0, 1);
```

As you can see from the above code, the RBF layer is used as a hidden layer between two **BasicLayer** objects.

4.3 Understanding Encog Synapses

In the previous section you saw how neural networks are made up of layers. Synapses are used to connect these layers together. Encog supports a variety of different synapse types. Synapses are used to carry information between the levels of a neural network. The synapses differ primarily in how the neurons are connected and what processing should be done on the information as it flows between levels.

Some of the synapse types supported by Encog make use of weight matrixes. A weight matrix allows the connections between each of the source layer neurons to have its connection weighted to the target neuron layer. By adjusting each of these weights, the neural network can learn.

In the next section you will learn about the synapse types that Encog supports. Any synapse that Encog makes use of must support the **ISynapse** interface. This interface is discussed in the next section.

4.3.1 The Synapse Interface

The **ISynapse** interface defines all of the essential methods that a class must support to function as a synapse. The **ISynapse** interface is shown in Listing 4.2.

Listing 4.2: The Synapse Interface

```
public interface ISynapse : IEncogPersistedObject
{
  INeuralData Compute(INeuralData input);

  ILayer FromLayer
  {
    get;
    set;
  }

  int FromNeuronCount
  {
    get;
  }

  Matrix.Matrix WeightMatrix
  {
    get;
    set;
  }

  int MatrixSize
  {
    get;
  }
```

```
  ILayer ToLayer
  {
    get;
    set;
  }

  int ToNeuronCount
  {
    get;
  }

  SynapseType SynapseType
  {
    get;
  }

  bool IsSelfConnected
  {
    get;
  }

  bool IsTeachable
  {
    get;
  }
}
```

As you can see there are a number of methods that must be implemented to create a synapse. We will now review some of the more important methods.

The **FromLayer** and **ToLayer** properties can be used to find the source and target layers for the neural synapse. The **IsSelfConnected** can also be used to determine if the synapse creates a self-connected layer. Encog also supports self-connected layers. A layer is self-connected if it has a self-connected synapse. A self-connected synapse is a synapse where the "from layer" and "to layer" are the same layer.

The **WeightMatrix** property allows access to the weight matrix for the neural network. A neural network that has a weight matrix is "teachable", and the **IsTeachable** method will return **true**. The **MatrixSize** property can also be called to determine the size of the weight matrix.

Finally, the **Compute** method is provided that applies any synapse specific transformation, such as weight matrixes. You will not usually call **Compute** directly, rather you will call the **Compute** method on the **INetwork** that this layer is attached to, and it will call the appropriate **Compute** functions for its various synapses.

4.3.2 Constructing Synapses

Often the synapses are simply created in the background and the programmer is not really aware of what type of synapse is even being created. The **AddLayer** method of

the **BasicNetwork** class automatically creates a new **WeightedSynapse** every time a new layer is added to the neural network.

The **AddLayer** method of the **BasicNetwork** class hides quite a bit of complexity. However, it is useful to see what is actually going on, and how the synapses are created. The following lines of code will show how to create a neural network "from scratch", where every object that is needed to create the neural network is created by hand. The first step is to create a **BasicNetwork** object to hold the layers.

```
 network = new BasicNetwork();
```

Next, we create three layers. Hidden, input and output layers are all created.

```
ILayer inputLayer = new BasicLayer(new ActivationSigmoid(),
  true,2);
ILayer hiddenLayer = new BasicLayer(
  new ActivationSigmoid(),true,2);
ILayer outputLayer = new BasicLayer(
  new ActivationSigmoid(),true,1);
```

Two synapses are needed to connect these three layers together. One synapse holds the input to the hidden layer. The second synapse holds the hidden to the output layer. The following lines of code create these synapses.

```
ISynapse synapseInputToHidden = new
WeightedSynapse(inputLayer,hiddenLayer);
ISynapse synapseHiddenToOutput = new
WeightedSynapse(hiddenLayer,outputLayer);
```

These synapses can then be added to the two layers they originate from.

```
inputLayer.Next.Add(synapseInputToHidden);
hiddenLayer.Next.Add(synapseHiddenToOutput);
```

The **BasicNetwork** object should be informed what the input and output layer. Finally, the network structure should be finalized and the weight matrix and threshold values reset.

```
network.TagLayer(BasicNetwork.TAG_INPUT, inputLayer);
network.TagLayer(BasicNetwork.TAG_OUTPUT, outputLayer);
network.Structure.FinalizeStructure();
network.Reset();
```

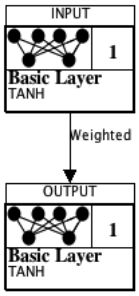
This section will discuss the different types of synapses supported by Encog. We will begin with the weighted synapse.

4.3.3 Using the WeightedSynapse Class

The weighted synapse is perhaps the most commonly used synapse type in Encog. The **WeightedSynapse** class is used by many different neural network architectures. Any place that a learning synapse is needed, the **WeightedSynapse** class is a good candidate. The **WeightedSynapse** connects every neuron in the source layer with

every neuron in the target layer. Figure 4.3 shows a diagram of the weighted synapse.

Figure 4.3: The Weighted Synapse



This is the default synapse type for Encog. To create a weighted synapse object usually you will simply add a layer to the network. The default synapse type is the weighted synapse. You can also construct a weighted synapse object with the following line of code.

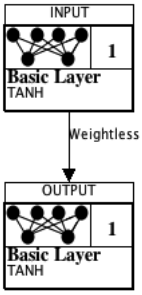
```
ISynapse synapse = new WeightedSynapse(from,to);
```

Once the weighted synapse has been created, it can be added to the next collection on the source layer's target.

4.3.4 Using the Weightless Synapse

The weightless synapse works very similar to the weighted synapse. The primary difference is that there are no weights in the weightless synapse. It provides a connection between each of the neurons in the source layer to every other neuron in the target layer. Figure 4.4 shows the weightless synapse.

Figure 4.4: The Weightless Synapse



The weightless synapse is implemented inside of the **`WeightlessSynapse`** class. The following line of code will construct a weightless synapse.

```
ISynapse synapse = new WeightlessSynapse(from,to);
```

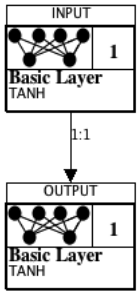
The weightless synapse is used when you would like to fully connect two layers, but want the information to pass through to the target layer untouched. The weightless

synapse is unteachable.

4.3.5 Using the OneToOne Synapse

The one to one synapse works very similar to the weightless synapse. Like the weightless synapse, the one to one synapse does not include any weight values. The primary difference is that every neuron in the source layer is connected to the corresponding neuron in the target layer. Each neuron is connected to only one other neuron. Because of this, the one to one synapse requires that the source and target layers have the same number of neurons. Figure 4.5 shows the one to one synapse.

Figure 4.5: The One to One Synapse



The following code segment shows how to construct a neural network that makes use of a one to one layer. The one to one layer is used in conjunction with a context layer.

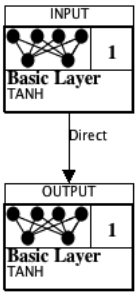
```
ILayer context = new ContextLayer(2);
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(1));
network.AddLayer(hidden = new BasicLayer(2));
hidden.AddNext(context, SynapseType.OneToOne);
context.AddNext(hidden);
network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
```

The one to one synapse is generally used to directly feed the values from the output of a layer to a context layer. However, it can serve any purpose where you would like to send a copy of the output of one layer to another similarly sized layer.

4.3.6 Using the Direct Synapse

The direct synapse is useful when you want to send a complete copy of the output from the source to every neuron in the target. Most layers are not designed to accept an array from every source neuron, so the number of layers that the direct synapse can be used with is limited. Currently, the only Encog layer type that supports the `DirectSynapse` is the `RadialBasisFunctionLayer` class. Figure 4.6 shows how the direct synapse works.

Figure 4.6: The Direct Synapse



The following code segment shows how to use the **DirectSynapse**.

```
RadialBasisFunctionLayer rbfLayer;
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(
  new ActivationLinear(), false, 2));
network.AddLayer(rbfLayer = new RadialBasisFunctionLayer(4),
  SynapseType.Direct);
network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
network.Reset();
rbfLayer.RandomizeGaussianCentersAndWidths(0, 1);
```

As you can see, the **DirectSynapse** is being used to feed a **RadialBasisFunctionLayer**.

4.4 Understanding Neural Logic

Every Encog neural network must contain a neural logic class. The **INeuralLogic** classes define how a neural network will process its layers and synapses. All neural logic classes must implement the **INeuralLogic** interface. By default a **BasicNetwork** class will make use of the **SimpleRecurrentLogic** logic class. This class can be used for both feedforward and simple recurrent networks. Because these are some of the most common neural network types in use, the **SimpleRecurrentLogic** class was chosen as the default.

The next few sections summarize the network logic classes provided by Encog.

4.4.1 The ART1Logic Class

The **ART1Logic** class is used to implement an adaptive resonance theory neural network. Adaptive Resonance Theory (ART) is a form of neural network developed by Stephen Grossberg and Gail Carpenter. There are several versions of the ART neural network, which are numbered ART-1, ART-2 and ART-3. The ART neural network is trained using either a supervised or unsupervised learning algorithm, depending on the version of ART being used. ART neural networks are used for pattern recognition and prediction. Encog presently supports ART1.

To create an ART1 neural network with Encog you should make use of the `ART1Logic` class.

4.4.2 The BAMLogic Class

The `BAMLogic` class is used to implement a Bidirectional Associative Memory (BAM) network. The BAM network is a type of neural network developed by Bart Kosko in 1988. The BAM is a recurrent neural network that works similarly to and allows patterns of different lengths to be mapped bidirectionally to other patterns. This allows it to act as almost a two-way hash map. During its training, the BAM network is fed pattern pairs. The two halves of each pattern do not have to be of the same length. However, all patterns must be of the same overall structure. The BAM network can be fed a distorted pattern on either side and will attempt to map to the correct value.

4.4.3 The BoltzmannLogic Class

The `BoltzmannLogic` class is used to implement a Boltzmann machine neural network. A Boltzmann machine is a type of neural network developed by Geoffrey Hinton and Terry Sejnowski. It appears identical to a Hopfield neural network except it contains a random nature to its output. A temperature value is present that influences the output from the neural network. As this temperature decreases so does the randomness. This is called simulated annealing. Boltzmann networks are usually trained in an unsupervised mode. However, supervised training can be used to refine what the Boltzmann machine recognizes.

To create a Boltzmann machine neural network with Encog you should make use of the `BoltzmannLogic` class.

4.4.4 The FeedforwardLogic Class

To create a feedforward with Encog the `FeedforwardLogic` class should be used. It is also possible to use the `SimpleRecurrentLogic` class as in place of the `FeedforwardLogic` class; however, the network will run slower. If there are no recurrent loops, the more simple `FeedforwardLogic` class should be used.

The feedforward neural network, or perceptron, is a type of neural network first described by Warren McCulloch and Walter Pitts in the 1940s. The feedforward neural network, and its variants, is the most widely used form of neural network. The feedforward neural network is often trained with the backpropagation training technique, though there are other more advanced training techniques, such as resilient propagation. The feedforward neural network uses weighted connections from an input layer to zero or more hidden layers, and finally to an output layer. It is suitable for many types of problems. Feedforward neural networks are used frequently in this document.

4.4.5 The HopfieldLogic Class

To create a Hopfield neural network with Encog, you should use the

HopfieldLogic class. Dr. John Hopfield developed the Hopfield neural network in 1979. The Hopfield network is a single layer recurrent neural network. The Hopfield network always maintains a "current state" which is the current output of the neural network. The Hopfield neural network also has an energy property, which is calculated exactly the same as the temperature property of the Boltzmann machine. The Hopfield network is trained for several patterns. The state of the Hopfield network will move towards the closest pattern, thus "recognizing" that pattern. As the Hopfield network moves towards one of these patterns, the energy lowers.

To create a Hopfield neural network with Encog you should make use of the **HopfieldLogic** class.

4.4.6 The SimpleRecurrentLogic Class

To create a neural network where some layers are connected to context layers that connect back to previous layers, you should use the **SimpleRecurrentLogic** class. The Elman and Jordan neural networks are examples of the sort of networks where the **SimpleRecurrentLogic** class can be used. The **SimpleRecurrentLogic** class can also be used to implement a simple feedforward neural network, however, the **FeedforwardLogic** class will execute faster.

To create either an Elman or Jordan type of neural network with Encog you should make use of the **SimpleRecurrentLogic** class.

4.4.7 The SOMLogic Class

To create a Self Organizing Map with Encog the **SOMLogic** class should be used. The Self Organizing Map (SOM) is a neural network type introduced by Teuvo Kohonen. SOM's are used to classify data into groups.

To create a SOM neural network with Encog you should make use of the **SOMLogic** class.

4.5 Understanding Properties and Tags

The **BasicNetwork** class also provides properties and tags to address the unique needs of different neural network logic types. Properties provide a set of name-value pairs that the neural logic can access. This is how you set properties about how the neural network should function. Tags allow individual layers to be identified. Some of the neural network logic types will affect layers differently. The layer tags allow the neural network logic to know which layer is which.

The following code shows several properties being set for an ART1 network.

```
BasicNetwork network = new BasicNetwork();

network.SetProperty(ARTLogic.PROPERTY_A1, 1);
network.SetProperty(ARTLogic.PROPERTY_B1, 2);
```

```
network.SetProperty(ARTLogic.PROPERTY_C1, 3);
network.SetProperty(ARTLogic.PROPERTY_D1, 4);
```

The first parameter specifies the name of the property. The neural network logic classes will define constants for properties that they require. The name of the property is a string.

The following code shows two network layers being tagged.

```
network.TagLayer(BasicNetwork.TAG_INPUT, layerF1);
network.TagLayer(BasicNetwork.TAG_OUTPUT, layerF2);
network.TagLayer(ART1Pattern.TAG_F1, layerF1);
network.TagLayer(ART1Pattern.TAG_F2, layerF2);
```

Here multiple tags are being applied to the **layerF1** and **layerF2** layers. One layer can have multiple tags; however, a single tag can only be applied to one layer.

The **BasicNetwork** class does not keep a list of layers. The only way that layers actually "join" the neural network is either by being tagged, or linked through a synapse connection to a layer that is already tagged.

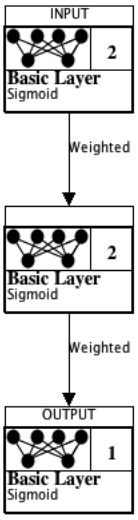
4.6 Building with Layers and Synapses

You are now familiar with all of the layer and synapse types supported by Encog. You will now be given a brief introduction to building ANNs with these neural network types. You will see how to construct several neural network types. They will be used to solve problems related to the XOR operator. For now, the XOR operator is a good enough introduction to several neural network architectures. We will see more interesting examples, as the document progresses. We will begin with the feedforward neural network.

4.6.1 Creating Feedforward Neural Networks

The feedforward neural network is one of the oldest types of neural networks still in common use. The feedforward neural network is also known as the perceptron. The feedforward neural network works by having one or more hidden layers sandwiched between an input and output layer. Figure 4.7 shows an Encog Workbench diagram of a feedforward neural network.

Figure 4.7: The Feedforward Neural Network



Listing 4.3 shows a simple example of a feedforward neural network learning to recognize the XOR operator.

Listing 4.3: Simple XOR Feedforward Neural Network

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Layers;
using Encog.Neural.Activation;
using Encog.Neural.Data.Basic;
using Encog.Neural.NeuralData;
using Encog.Neural.Networks.Training;
using Encog.Neural.Data;
using Encog.Neural.Networks.Training.Propagation.Resilient;
using ConsoleExamples.Examples;

namespace Encog.Examples.XOR.Resilient
{
  public class XORResilient : IExample
  {
    public static ExampleInfo Info
    {
      get
      {
        ExampleInfo info = new ExampleInfo(
          typeof(XORResilient),
          "xor-rprop",
          "XOR Operator with Resilient Propagation",
          "Use RPROP to learn the XOR operator.");
        return info;
        }
      }
```

```
    /// <summary>
    /// Input for the XOR function.
    /// </summary>
    public static double[][] XOR_INPUT ={
      new double[2] { 0.0, 0.0 },
      new double[2] { 1.0, 0.0 },
      new double[2] { 0.0, 1.0 },
      new double[2] { 1.0, 1.0 } };

    /// <summary>
    /// Ideal output for the XOR function.
    /// </summary>
    public static double[][] XOR_IDEAL = {
      new double[1] { 0.0 },
      new double[1] { 1.0 },
      new double[1] { 1.0 },
      new double[1] { 0.0 } };

    /// <summary>
    /// Program entry point.
    /// </summary>
    /// <param name="args">Not used.</param>
    public void Execute(IExampleInterface app)
    {
      BasicNetwork network = new BasicNetwork();
      network.AddLayer(new BasicLayer(
        new ActivationSigmoid(), true, 2));
      network.AddLayer(new BasicLayer(
        new ActivationSigmoid(), true, 6));
      network.AddLayer(new BasicLayer(
        new ActivationSigmoid(), true, 1));
      network.Structure.FinalizeStructure();
      network.Reset();

      INeuralDataSet trainingSet =
        new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

      // train the neural network
      ITrain train = new ResilientPropagation(
        network, trainingSet);

      int epoch = 1;

      do
      {
        train.Iteration();
        Console.WriteLine("Epoch #" + epoch
          + " Error:" + train.Error);
        epoch++;
      } while ((epoch < 5000) && (train.Error > 0.001));

      // test the neural network
      Console.WriteLine("Neural Network Results:");
      foreach (INeuralDataPair pair in trainingSet)
      {
```

```
            INeuralData output = network.Compute(pair.Input);
            Console.WriteLine(pair.Input[0] + "," + pair.Input[1]
            + ", actual=" + output[0] + ",ideal=" + pair.Ideal[0]);
          }
        }
    }
}
```

As you can see from the above listing, it is very easy to construct a three-layer, feedforward neural network. Essentially, three new **BasicLayer** objects are created and added to the neural network with calls to the **AddLayer** method. Because no synapse type is specified, the three layers are connected together using the **WeightedSynapse**.

You will notice that after the neural network is constructed, it is trained. There are quite a few ways to train a neural network in Encog. Training is the process where the weights and thresholds are adjusted to values that will produce the desired output from the neural network. This example uses resilient propagation (RPROP) training. RPROP is the best choice for most neural networks to be trained with Encog. For certain special cases, some of the other training types may be more efficient.

```
// train the neural network
ITrain train = new ResilientPropagation(
          network, trainingSet);
```

With the trainer setup we must now cycle through a bunch of iterations, or epochs. Each of these training iterations should decrease the "error" of the neural network. The error is the difference between the current actual output of the neural network and the desired output.

```
int epoch = 1;
do
{
  train.Iteration();
  Console.WriteLine("Epoch #" + epoch
    + " Error:" + train.Error);
  epoch++
```

Continue training the neural network so long as the error rate is greater than one percent.

```
} while ((epoch < 5000) && (train.Error > 0.001));
```

Now that the neural network has been trained, we should test it. To do this, the same data that the neural network was trained with is presented to the neural network. The following code does this.

```
Console.WriteLine("Neural Network Results:");
foreach (INeuralDataPair pair in trainingSet)
{
```

```
  INeuralData output = network.Compute(pair.Input);
  Console.WriteLine(pair.Input[0] + "," + pair.Input[1]
  + ", actual=" + output[0] + ",ideal=" + pair.Ideal[0]);
}
```

This will produce the following output:

```
Epoch #1 Error:0.9902997764512583
Epoch #2 Error:0.6762359214192293
Epoch #3 Error:0.49572129129302844
Epoch #4 Error:0.49279160045197135
Epoch #5 Error:0.5063357328001542
Epoch #6 Error:0.502484567412553
Epoch #7 Error:0.4919515177527043
Epoch #8 Error:0.49157058621332506
Epoch #9 Error:0.48883664423510526
Epoch #10 Error:0.48977067420698456
Epoch #11 Error:0.4895238942630234
Epoch #12 Error:0.4870271073515729
Epoch #13 Error:0.48534672846811844
Epoch #14 Error:0.4837776485977757
Epoch #15 Error:0.48184530627656685
Epoch #16 Error:0.47980242878514856
Epoch #17 Error:0.47746641141708474
Epoch #18 Error:0.4748474362926616
Epoch #19 Error:0.47162728117571795
Epoch #20 Error:0.46807640808835427
...
Epoch #495 Error:0.010583637636670955
Epoch #496 Error:0.010748859630158925
Epoch #497 Error:0.010342203029249158
Epoch #498 Error:0.00997945501479827
Neural Network Results: 0.0,0.0, actual=0.005418223644461675,ideal=0.0
1.0,0.0, actual=0.9873413174817033,ideal=1.0 0.0,1.0,
actual=0.9863636878918781,ideal=1.0 1.0,1.0,
actual=0.007650291171204077,ideal=0.0
```

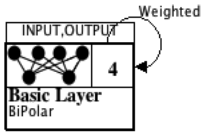
As you can see the error rate starts off high and steadily decreases. Finally, the patterns are presented to the neural network. As you can see, the neural network can handle the XOR operator. It does not produce the exact output it was trained with, but it is very close. The values 0.0054 and 0.0076 are very close to zero, just as 0.987 and 0.986 are very close to one.

For this network, we are testing the neural network with exactly the same data that the neural network was trained with. Generally, this is a very bad practice. You want to test the neural network on data that it was not trained with. This lets you see how the neural network is performing with new data that it has never processed before. However, the XOR function only has four possible combinations, and they all represent unique patterns that the network must be trained for. Neural networks presented later in this document will not use all of their data for training. Rather, they will be tested on data it has never been presented with before.

4.6.2 Creating Self-Connected Neural Networks

We will now look at self-connected neural networks. The Hopfield neural network is a good example of a self-connected, neural network. The Hopfield neural network contains a single layer of neurons. This layer is connected to itself. Every neuron on the layer is connected to every other neuron on the same layer. However, no two neurons are connected to themselves. Figure 4.8 shows a Hopfield neural network diagramed in the Encog Workbench.

Figure 4.8: The Hopfield Neural Network



Listing 4.4 shows a simple example of a Hopfield neural network learning to recognize various patterns.

Listing 4.4: Hopfield Neural Network

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ConsoleExamples.Examples;
using Encog.Neural.NeuralData.Bipolar;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Logic;
using Encog.Neural.Networks.Pattern;

namespace Encog.Examples.Hopfield.Associate
{
  public class HopfieldAssociate: IExample
  {
    public static ExampleInfo Info
    {
      get
      {
        ExampleInfo info = new ExampleInfo(
          typeof(HopfieldAssociate),
          "hopfield-associate",
          "Hopfield Associates Patterns",
"Simple Hopfield neural network that learns to associate patterns.");
          return info;
        }
      }


    public const int HEIGHT = 10;
    public const int WIDTH = 10;

    private IExampleInterface app;
```

```
    public String[][] PATTERN = {
      new String[WIDTH] {
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O"  },

      new String[WIDTH] {
        "OO  OO  OO",
        "OO  OO  OO",
        "  OO  OO  ",
        "  OO  OO  ",
        "OO  OO  OO",
        "OO  OO  OO",
        "  OO  OO  ",
        "  OO  OO  ",
        "OO  OO  OO",
        "OO  OO  OO"  },

      new String[WIDTH]  {
        "OOOOO     ",
        "OOOOO     ",
        "OOOOO     ",
        "OOOOO     ",
        "OOOOO     ",
        "     OOOOO",
        "     OOOOO",
        "     OOOOO",
        "     OOOOO",
        "     OOOOO"  },

      new String[WIDTH] {
        "O  O  O  O",
        " O  O  O  ",
        "  O  O  O ",
        "O  O  O  O",
        " O  O  O  ",
        "  O  O  O ",
        "O  O  O  O",
        " O  O  O  ",
        "  O  O  O ",
        "O  O  O  O"  },

      new String[WIDTH]  {
        "OOOOOOOOOO",
        "O        O",
        "O OOOOOO O",
        "O O    O O",
        "O O OO O O",
```

```
      "O O OO O O",
      "O O    O O",
      "O OOOOOO O",
      "O        O",
      "OOOOOOOOOO"  } };

  public String[][] PATTERN2 = {
    new String[WIDTH] {
      "          ",
      "          ",
      "          ",
      "          ",
      "          ",
      " O O O O O",
      "O O O O O ",
      " O O O O O",
      "O O O O O ",
      " O O O O O"  },

      new String[WIDTH] {
      "OOO O    O",
      " O  OOO OO",
      "  O O OO O",
      " OOO   O  ",
      "OO  O  OOO",
      " O OOO   O",
      "O OO  O  O",
      "   O OOO  ",
      "OO OOO  O ",
      " O  O  OOO"  },

      new String[WIDTH] {
      "OOOOO     ",
      "O   O OOO ",
      "O   O OOO ",
      "O   O OOO ",
      "OOOOO     ",
      "     OOOOO",
      " OOO O   O",
      " OOO O   O",
      " OOO O   O",
      "     OOOOO"  },

      new String[WIDTH] {
      "O  OOOO  O",
      "OO  OOOO  ",
      "OOO  OOOO ",
      "OOOO  OOOO",
      " OOOO  OOO",
      "  OOOO  OO",
      "O  OOOO  O",
      "OO  OOOO  ",
      "OOO  OOOO ",
      "OOOO  OOOO"  },
```

```
      new String[WIDTH] {
      "OOOOOOOOOO",
      "O        O",
      "O        O",
      "O        O",
      "O   OO   O",
      "O   OO   O",
      "O        O",
      "O        O",
      "O        O",
      "OOOOOOOOOO"  } };

  public BiPolarNeuralData ConvertPattern(
    String[][] data, int index)
  {
    int resultIndex = 0;
    BiPolarNeuralData result =
      new BiPolarNeuralData(WIDTH * HEIGHT);
    for (int row = 0; row < HEIGHT; row++)
    {
      for (int col = 0; col < WIDTH; col++)
      {
        char ch = data[index][row][col];
        result.SetBoolean(resultIndex++, ch == 'O');
      }
    }
    return result;
  }

  public void Display(
    BiPolarNeuralData pattern1, BiPolarNeuralData pattern2)
  {
    int index1 = 0;
    int index2 = 0;

    for (int row = 0; row < HEIGHT; row++)
    {
      StringBuilder line = new StringBuilder();

      for (int col = 0; col < WIDTH; col++)
      {
        if (pattern1.GetBoolean(index1++))
          line.Append('O');
        else
          line.Append(' ');
      }

      line.Append("   ->   ");

      for (int col = 0; col < WIDTH; col++)
      {
        if (pattern2.GetBoolean(index2++))
          line.Append('O');
        else
          line.Append(' ');
```

```
        }

        Console.WriteLine(line.ToString());
      }
    }


    public void Evaluate(BasicNetwork hopfield,
      String[][] pattern)
    {
      HopfieldLogic hopfieldLogic =
        (HopfieldLogic)hopfield.Logic;
      for (int i = 0; i < pattern.Length; i++)
      {
        BiPolarNeuralData pattern1 = ConvertPattern(pattern, i);
        hopfieldLogic.CurrentState = pattern1;
        int cycles = hopfieldLogic.RunUntilStable(100);
        BiPolarNeuralData pattern2 =
          (BiPolarNeuralData)hopfieldLogic.CurrentState;
        Console.WriteLine("Cycles until stable(max 100): "
           + cycles + ", result=");
        Display(pattern1, pattern2);
        Console.WriteLine("----------------------");
      }
    }

    public void Execute(IExampleInterface app)
    {
      this.app = app;
      HopfieldPattern pattern = new HopfieldPattern();
      pattern.InputNeurons = WIDTH * HEIGHT;
      BasicNetwork hopfield = pattern.Generate();
      HopfieldLogic hopfieldLogic =
        (HopfieldLogic)hopfield.Logic;

      for (int i = 0; i < PATTERN.Length; i++)
      {
        hopfieldLogic.AddPattern(ConvertPattern(PATTERN, i));
      }

      Evaluate(hopfield, PATTERN);
      Evaluate(hopfield, PATTERN2);
    }
  }
}
```

The Hopfield example begins by creating a **HopfieldPattern** class. The pattern classes allow for common types of neural networks to be constructed automatically. You simply provide the parameters about the type of neural network you wish to create, and the pattern takes care of setting up layers, synapses, parameters and tags.

```
HopfieldPattern pattern = new HopfieldPattern();
```

This Hopfield neural network is going to recognize graphic patterns. These graphic

patterns are mapped on to grids. The number of input neurons will be the total number of cells in the grid. This is the width times the height.

```
pattern.InputNeurons = WIDTH * HEIGHT;
```

The Hopfield pattern requires very little input, just the number of input neurons. Other patterns will require more parameters. Now that the **HopfieldPattern** has been provided with all that it needs, the **Generate** method can be called to create the neural network.

```
BasicNetwork hopfield = pattern.Generate();
```

The logic object is obtained for the Hopfield network.

```
HopfieldLogic hopfieldLogic =
        (HopfieldLogic)hopfield.Logic;
```

The logic class is used to add the patterns that the neural network is to be trained on. This is similar to the training seen in the last section, except it happens much faster for the simple Hopfield neural network.

```
for(int i=0;i<PATTERN.length;i++){
  hopfieldLogic.AddPattern(ConvertPattern(PATTERN,i));
}
```

Now that the network has been "trained" we will test it. Just like in the last section, we will evaluate the neural network with the same data with which it was trained.

```
Evaluate(hopfield,PATTERN);
```

However, in addition to the data that the network has already been presented with, we will also present new data. This new data are distorted images of the data that the network was trained on. The network should be able to still recognize the patterns, even though they were distorted.

```
Evaluate(hopfield,PATTERN2);
```

The following shows the output of the Hopfield neural network. As you can see the Hopfield neural network is first presented with the patterns that it was trained on. The Hopfield network simply echoes these patterns. Next, the Hopfield neural network is presented with distorted versions of the patterns with which it was trained. As you can see from the code snippet below, the Hopfield neural network still recognizes the values.

```
Cycles until stable(max 100): 1, result=
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
```

```
 O O O O O   ->    O O O O O
----------------------
Cycles until stable(max 100): 1, result=
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
  OO  OO     ->     OO  OO
  OO  OO     ->     OO  OO
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
  OO  OO     ->     OO  OO
  OO  OO     ->     OO  OO
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
----------------------
Cycles until stable(max 100): 1, result=
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
----------------------
Cycles until stable(max 100): 1, result=
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
----------------------
Cycles until stable(max 100): 1, result=
OOOOOOOOOO   ->   OOOOOOOOOO
O        O   ->   O        O
O OOOOOO O   ->   O OOOOOO O
O O    O O   ->   O O    O O
O O OO O O   ->   O O OO O O
O O OO O O   ->   O O OO O O
O O    O O   ->   O O    O O
O OOOOOO O   ->   O OOOOOO O
O        O   ->   O        O
OOOOOOOOOO   ->   OOOOOOOOOO
----------------------
Cycles until stable(max 100): 2, result=
             ->   O O O O O
             ->    O O O O O
             ->   O O O O O
             ->    O O O O O
```

```
             ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
----------------------
Cycles until stable(max 100): 2, result=
OOO O    O   ->   OO  OO  OO
 O  OOO OO   ->   OO  OO  OO
  O O OO O   ->     OO  OO
 OOO   O     ->     OO  OO
OO  O  OOO   ->   OO  OO  OO
 O OOO   O   ->   OO  OO  OO
O OO  O  O   ->     OO  OO
   O OOO     ->     OO  OO
OO OOO  O    ->   OO  OO  OO
 O  O  OOO   ->   OO  OO  OO
----------------------
Cycles until stable(max 100): 2, result=
OOOOO        ->   OOOOO
O   O OOO    ->   OOOOO
O   O OOO    ->   OOOOO
O   O OOO    ->   OOOOO
OOOOO        ->   OOOOO
     OOOOO   ->        OOOOO
 OOO O   O   ->        OOOOO
 OOO O   O   ->        OOOOO
 OOO O   O   ->        OOOOO
     OOOOO   ->        OOOOO
----------------------
Cycles until stable(max 100): 2, result=
O  OOOO  O   ->   O  O  O  O
OO  OOOO     ->    O  O  O
OOO  OOOO    ->     O  O  O
OOOO  OOOO   ->   O  O  O  O
 OOOO  OOO   ->    O  O  O
  OOOO  OO   ->     O  O  O
O  OOOO  O   ->   O  O  O  O
OO  OOOO     ->    O  O  O
OOO  OOOO    ->     O  O  O
OOOO  OOOO   ->   O  O  O  O
----------------------
Cycles until stable(max 100): 2, result=
OOOOOOOOOO   ->   OOOOOOOOOO
O        O   ->   O        O
O        O   ->   O OOOOOO O
O        O   ->   O O    O O
O   OO   O   ->   O O OO O O
O   OO   O   ->   O O OO O O
O        O   ->   O O    O O
O        O   ->   O OOOOOO O
O        O   ->   O        O
OOOOOOOOOO   ->   OOOOOOOOOO
----------------------
```

As you can see, the neural network can recognize the distorted values as well as those values with which it was trained. This is a much more comprehensive test than was performed in the previous section. This is because the network is evaluated with data that it has never seen before.

When the Hopfield neural network recognizes a pattern, it returns the pattern that it was trained with. This is called autoassociation.

The program code for the **Evaluate** method will now be examined. This shows how to present a pattern to the neural network.

```
public void Evaluate(BasicNetwork hopfield,
      String[][] pattern)
{
```

First the logic object is obtained.

```
HopfieldLogic hopfieldLogic =
  (HopfieldLogic)hopfield.Logic;
```

Loop over all of the patterns and present each to the neural network.

```
for (int i = 0; i < pattern.Length; i++)
{
  BiPolarNeuralData pattern1 = ConvertPattern(pattern, i);
```

The pattern is obtained from the array and converted to a form that can be presented to the neural network. The graphic patterns are binary, either the pixel is on or it is off. To convert the image all displayed pixels are converted to the numbers. We are using bipolar numbers, so a display pixel is 1, a hidden pixel is -1.

The Hopfield neural network has a current state. The neurons will be at either 1 or -1 level. The current state of the Hopfield network is set to the pattern that we want to recognize.

```
    hopfieldLogic.CurrentState = pattern1;
```

The Hopfield network will be run until it stabilizes. A Hopfield network will adjust its pattern until it no longer changes. At this point it has stabilized. The Hopfield neural network will stabilize on one of the patterns that it was trained on. The following code will run the Hopfield network until it stabilizes, up to 100 iterations.

```
int cycles = hopfieldLogic.RunUntilStable(100);
BiPolarNeuralData pattern2 =
  (BiPolarNeuralData)hopfieldLogic.CurrentState;
```

Once the network's state has stabilized it is displayed.

```
Console.WriteLine("Cycles until stable(max 100): "
  + cycles + ", result=");
Display(pattern1, pattern2);
Console.WriteLine("----------------------");
```

These are just a few of the neural network types that can be constructed with Encog. As the document progresses, you will learn many more.

4.7 Chapter Summary

Encog neural networks are made up of layers, synapses, properties and a neural logic class. This chapter reviewed each of these. A layer is a collection of similar neurons. A synapse connects one layer to another. Properties define unique qualities that one neural network type might have. The neural logic class defines how the output of the neural network should be calculated.

Activation functions are very important to neural networks. Activation functions scale the output from one layer before it reaches the next layer. The next chapter will discuss how Encog makes use of activation functions.

5 Building Encog Neural Networks

- Activation Functions
- Derivatives and Propagation Training
- Choosing an Activation Function

Activation functions are used by many neural network architectures to scale the output from layers. Encog provides many different activation functions that can be used to construct neural networks. In this chapter you will be introduced to these activation functions.

5.1 The Role of Activation Functions

Activation functions are attached to layers. They are used to scale data output from a layer. Encog applies a layer's activation function to the data that the layer is about to output. If you do not specify an activation function for **BasicLayer**, the hyperbolic tangent activation will be the defaulted. The following code creates several **BasicLayer** objects with a default hyperbolic tangent activation function.

```
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(2));
network.AddLayer(new BasicLayer(3));
network.AddLayer(new BasicLayer(1));
network.Structure.FinalizeStructure();
network.Reset();
```

If you would like to use an activation function other than the hyperbolic tangent function, use code similar to the following:

```
ActivationSigmoid a = new ActivationSigmoid();
BasicNetwork network = new BasicNetwork();
network.AddLayer(new BasicLayer(a,true,2));
network.AddLayer(new BasicLayer(a,true,3));
network.AddLayer(new BasicLayer(a,true,1));
network.Structure.FinalizeStructure();
network.Reset();
```

The sigmoid tangent activation function is assigned to the variable **a** and passed to each of the **AddLayer** calls. The **true** value, that was also introduced, specifies that the **BasicLayer** should also have threshold values.

5.1.1 The ActivationFunction Interface

All classes that are to serve as activation functions must implement the **IActivationFunction** interface. This interface is shown in Listing 3.1.

Listing 3.1: The IActivationFunction Interface

```
public interface IActivationFunction : IEncogPersistedObject
{
  void ActivationFunction(double[] d);
```

```
  void DerivativeFunction(double[] d);
  bool HasDerivative
  {
    get;
  }
}
```

The actual activation function is implemented inside of the **ActivationFunction** method. The **ActivationSIN** class is a very simple activation function that implements the sine wave. You can see the **ActivationFunction** implementation below.

```
public override void ActivationFunction(double[] d)
{
  for (int i = 0; i < d.Length; i++)
  {
    d[i] = BoundMath.Sin(d[i]);
  }
}
```

As you can see, the activation simply applies the sine function to the array of provided values. This array represents the output neuron values that the activation function is to scale. It is important that the function be given the entire array at once. Some of the activation functions perform operations, such as averaging, that require seeing the entire output array.

You will also notice from the above code that a special class, named **BoundMath**, is used to calculate the sine. This causes "not a number" and "infinity" values to be removed. Sometimes, during training, unusually large or small numbers may be generated. The **BoundMath** class is used to eliminate these values by binding them to either a very large or a very small number. The sine function will not create an out-of-bounds number, and **BoundMath** is used primarily for completeness.

However, we will soon see other functions that could produce out of bound numbers. Exponent and radical functions can be particularly prone to this. Once a "not a number" (NaN) is introduced into the neural network, the neural network will no longer produce useful results. As a result, bounds checking must be performed.

5.1.2 Derivatives of Activation Functions

If you would like to use propagation training with your activation function, then the activation function must have a derivative. The derivative is calculated by a function named **DerivativeFunction.**

```
public override void DerivativeFunction(double[] d)
{
  for (int i = 0; i < d.Length; i++)
  {
    d[i] = BoundMath.Cos(d[i]);
  }
}
```

The **DerivativeFunction** works very similar to the **ActivationFunction**, an array of values is passed in to calculate.

5.2 Encog Activation Functions

The next sections will explain each of the activation functions supported by Encog. There are several factors to consider when choosing an activation function. Firstly, the type of neural network you are using may dictate the activation function you must use. Secondly, you should consider if you would like to train the neural network using propagation. Propagation training requires an activation function that provides a derivative. You must also consider the range of numbers you will be dealing with. This is because some activation functions deal with only positive numbers or numbers in a particular range.

5.2.1 ActivationBiPolar

The **ActivationBiPolar** activation function is used with neural networks that require bipolar numbers. Bipolar numbers are either **true** or **false**. A **true** value is represented by a bipolar value of 1; a **false** value is represented by a bipolar value of -1. The bipolar activation function ensures that any numbers passed to it are either -1 or 1. The **ActivationBiPolar** function does this with the following code:

```
if (d[i] > 0)
{
  d[i] = 1;
}
else
{
  d[i] = -1;
}
```

As you can see the output from this activation is limited to either -1 or 1. This sort of activation function is used with neural networks that require bipolar output from one layer to the next. There is no derivative function for bipolar, so this activation function cannot be used with propagation training.

5.2.2 Activation Competitive

The **ActivationCompetitive** function is used to force only a select group of neurons to win. The winner is the group of neurons that has the highest output. The outputs of each of these neurons are held in the array passed to this function. The size of the winning group of neurons is definable. The function will first determine the winners. All non-winning neurons will be set to zero. The winners will all have the same value, which is an even division of the sum of the winning outputs.

This function begins by creating an array that will track whether each neuron has already been selected as one of the winners. We also count the number of winners so far.

```
bool[] winners = new bool[d.Length];
```

```
double sumWinners = 0;
```

First, we loop **maxWinners** a number of times to find that number of winners.

```
for (int i = 0; i < this.maxWinners; i++)
{
  double maxFound = Double.MinValue;
  int winner = -1;
```

Now, we must find one winner. We will loop over all of the neuron outputs and find the one with the highest output.

```
for (int j = 0; j < d.Length; j++)
{
```

If this neuron has not already won, and it has the maximum output then it might potentially be a winner, if no other neuron has a higher activation.

```
  if (!winners[j] && d[j] > maxFound)
  {
     winner = j;
     maxFound = d[j];
  }
}
```

Keep the sum of the winners that were found, and mark this neuron as a winner. Marking it a winner will prevent it from being chosen again. The sum of the winning outputs will ultimately be divided among the winners.

```
sumWinners += maxFound;
winners[winner] = true;
```

Now that we have the correct number of winners, we must adjust the values for winners and non-winners. The non-winners will all be set to zero. The winners will share the sum of the values held by all winners.

```
for (int i = 0; i < d.Length; i++)
{
  if (winners[i])
  {
    d[i] = d[i] / sumWinners;
  }
  else
  {
    d[i] = 0.0;
  }
```

This sort of an activation function can be used with competitive, learning neural networks, such as the Self Organizing Map. This activation function has no derivative, so it cannot be used with propagation training.

5.2.3 ActivationGaussian

The **ActivationGaussian** function is based on the Gaussian function. The

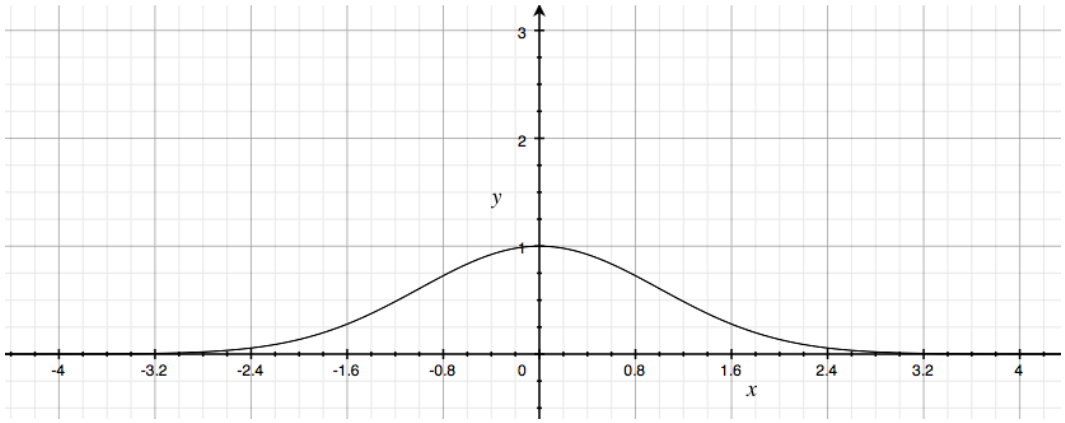
Gaussian function produces the familiar bell-shaped curve. The equation for the Gaussian function is shown in Equation 5.1.

Equation 5.1: The Gaussian Function

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

There are three different constants that are fed into the Gaussian function. The constant **a** represents the curve's peak. The constant **b** represents the position of the curve. The constant **c** represents the width of the curve.

Figure 5.1: The Graph of the Gaussian Function



The Gaussian function is implemented in C# as follows.

```
return this.peak
      * BoundMath.Exp(-Math.Pow(x - this.center, 2)
      / (2.0 * this.width * this.width));
```

The Gaussian activation function is not a commonly used activation function. However, it can be used when finer control is needed over the activation range. The curve can be aligned to somewhat approximate certain functions.

The radial basis function layer provides an even finer degree of control, as it can be used with multiple Gaussian functions. There is a valid derivative of the Gaussian function; therefore, the Gaussian function can be used with propagation training.

5.2.4 ActivationLinear

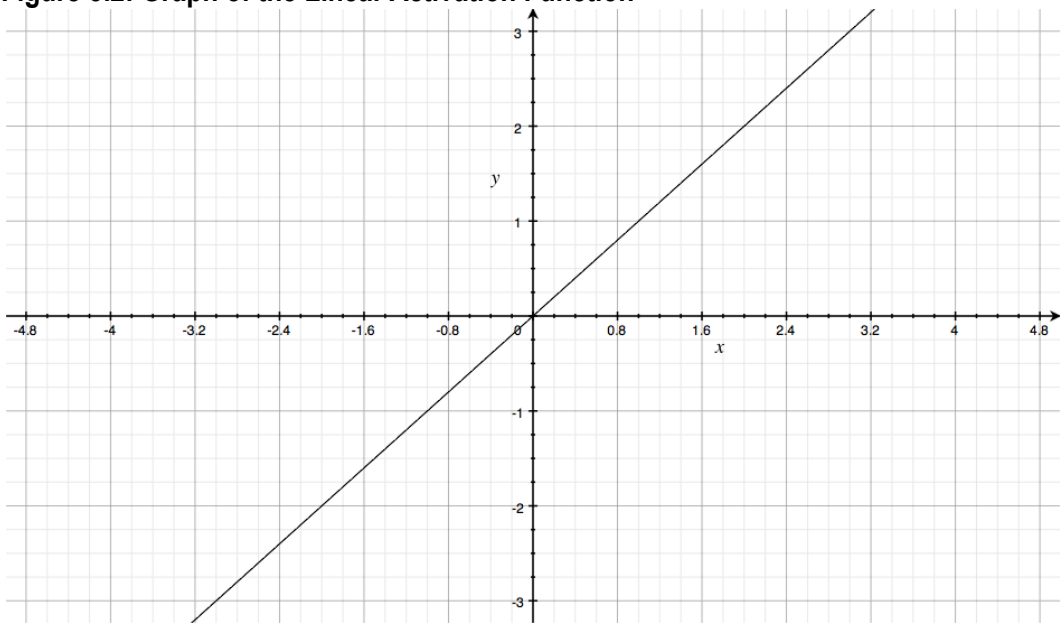
The **ActivationLinear** function is really no activation function at all. It simply implements the linear function. The linear function can be seen in Equation 5.2.

Equation 5.2: The Linear Activation Function

$$f(x) = x$$

The graph of the linear function is a simple line, as seen in Figure 5.2.

Figure 5.2: Graph of the Linear Activation Function



The C# implementation for the linear activation function is very simple. It does nothing. The input is returned as it was passed.

```
public void ActivationFunction(double[] d)
{
}
```

The linear function is used primarily for specific types of neural networks that have no activation function, such as the self-organizing map. The linear activation function has a constant derivative of one, so it can be used with propagation training. The output layer of a feedforward neural network trained with propagation sometimes uses linear layers.

5.2.5 ActivationLOG

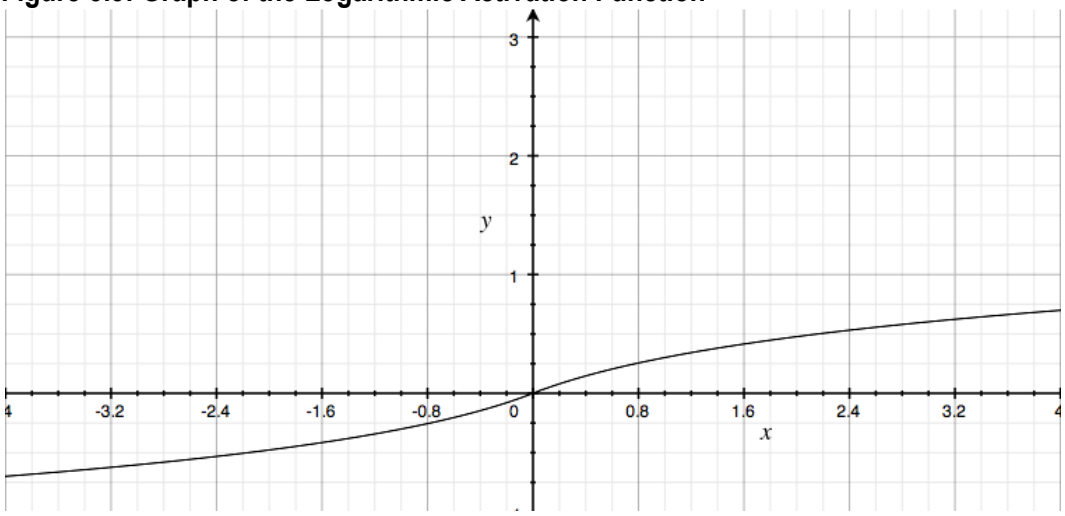
The **ActivationLog** activation function uses an algorithm based on the log function. The following C# code shows how this is calculated.

```
if (d[i] >= 0)
{
  d[i] = BoundMath.Log(1 + d[i]);
}
else
{
  d[i] = -BoundMath.Log(1 - d[i]);
}
```

This produces a curve similar to the hyperbolic tangent activation function, which will

be discussed later in this chapter. You can see the graph for the logarithmic activation function in Figure 5.3.

Figure 5.3: Graph of the Logarithmic Activation Function



The logarithmic activation function can be useful to prevent saturation. A hidden node of a neural network is considered saturated when, on a given set of inputs, the output is approximately 1 or -1 in most cases. This can slow training significantly. This makes the logarithmic activation function a possible choice when training is not successful using the hyperbolic tangent activation function.

As illustrated in Figure 5.3, the logarithmic activation function spans both positive and negative numbers. This means it can be used with neural networks where negative number output is desired. Some activation functions, such as the sigmoid activation function will only produce positive output. The logarithmic activation function does have a derivative, so it can be used with propagation training.

5.2.6 ActivationSigmoid

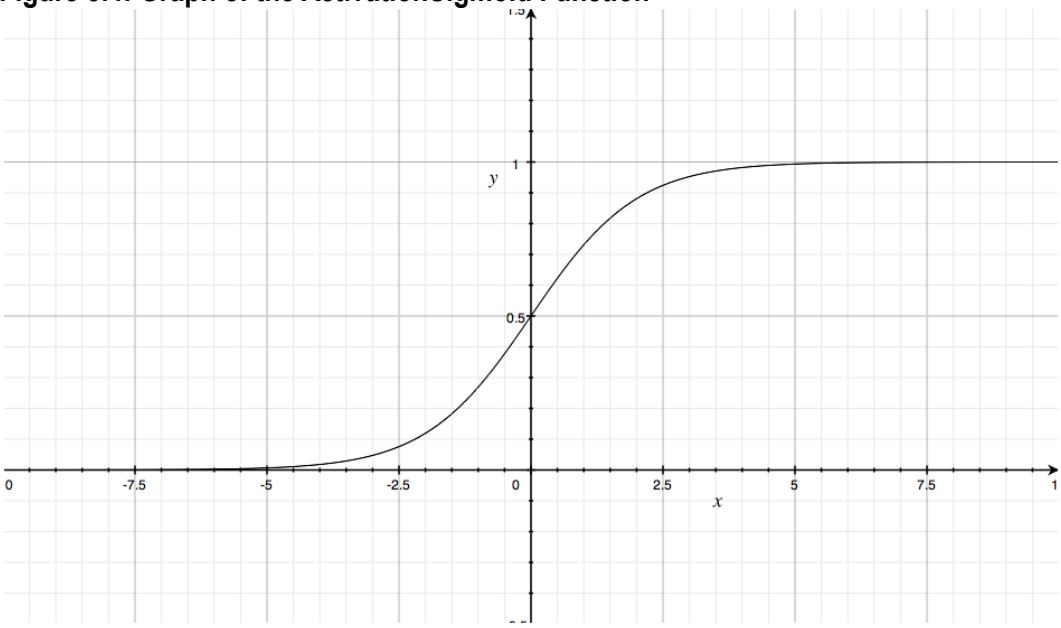
The `ActivationSigmoid` activation function should only be used when positive number output is expected, because the `ActivationSigmoid` function will only produce positive output. The equation for the `ActivationSigmoid` function can be seen in Equation 5.3.

Equation 5.3: The ActivationSigmoid Function

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

The `ActivationSigmoid` function will move negative numbers into the positive range. This can be seen in Figure 5.4, which shows the graph of the sigmoid function.

Figure 5.4: Graph of the ActivationSigmoid Function

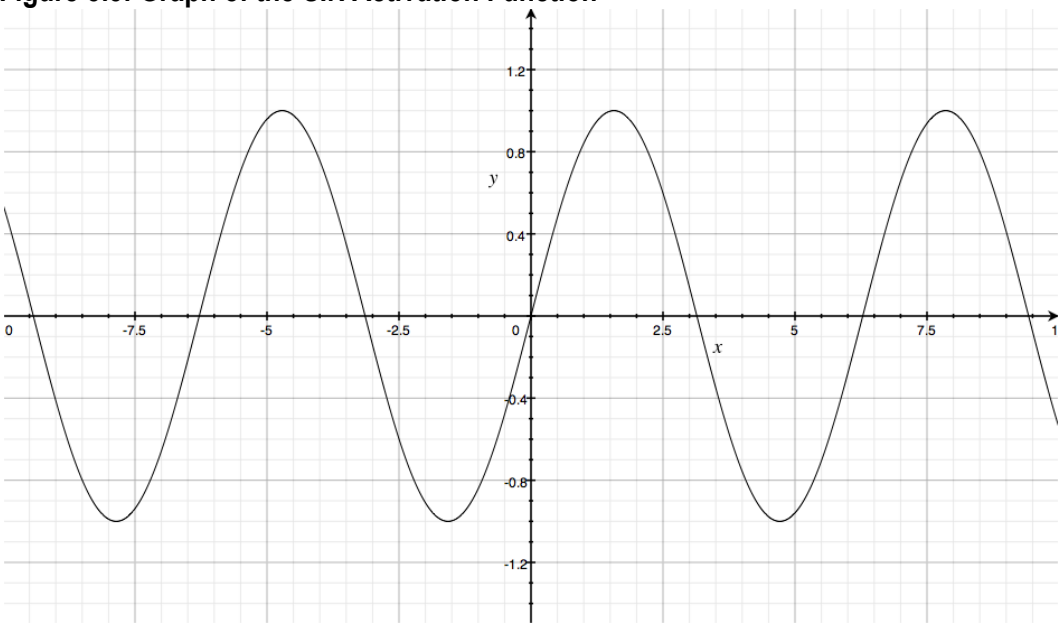


The `ActivationSigmoid` function is a very common choice for feedforward and simple recurrent neural networks. However, you must be sure that the training data does not expect negative output numbers. If negative numbers are required, consider using the hyperbolic tangent activation function.

5.2.7 ActivationSIN

The `ActivationSIN` activation function is based on the sine function. It is not a commonly used activation function. However, it is sometimes useful for certain data that periodically changes over time. The graph for the `ActivationSIN` function is shown in Figure 5.5.

Figure 5.5: Graph of the SIN Activation Function



The **ActivationSIN** function works with both negative and positive values. Additionally, the **ActivationSIN** function has a derivative and can be used with propagation training.

5.2.8 ActivationSoftMax

The **ActivationSoftMax** activation function is an activation that will scale all of the input values so that their sum will equal one. The **ActivationSoftMax** activation function is sometimes used as a hidden layer activation function.

The activation function begins by summing the natural exponent of all of the neuron outputs.

```
double sum = 0;
for (int i = 0; i < d.length; i++)
{
  d[i] = BoundMath.Exp(d[i]);
  sum += d[i];
}
```

The output from each of the neurons is then scaled according to this sum. This produces outputs that will sum to 1.

```
for (int i = 0; i < d.Length; i++)
{
  d[i] = d[i] / sum;
}
```

The **ActivationSoftMax** is generally used in the hidden layer of a neural network

or a classification neural network.

5.2.9 ActivationTANH

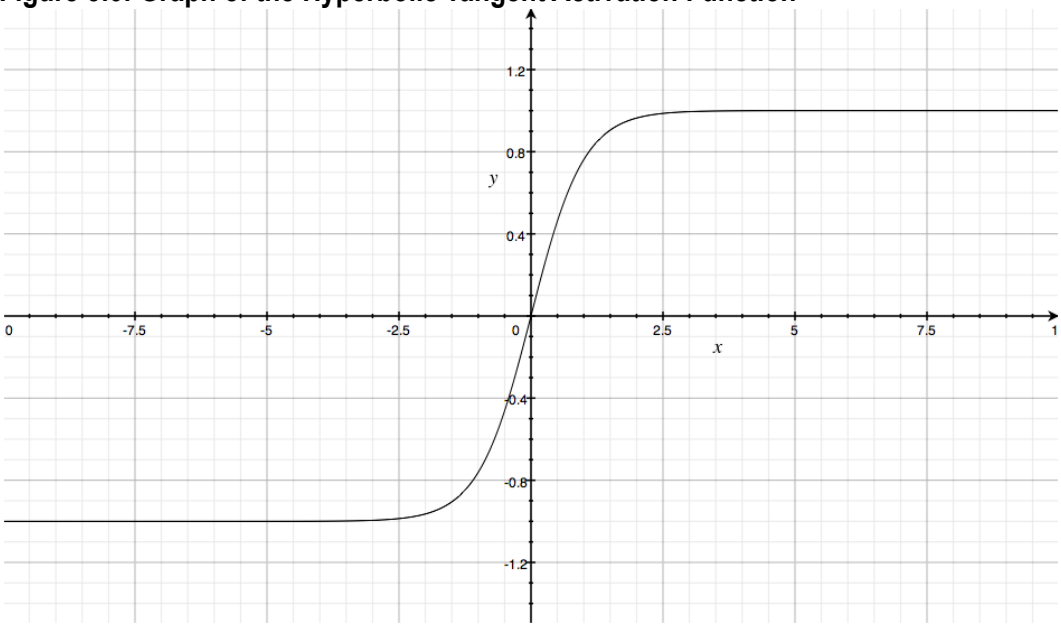
The `ActivationTANH` activation function is an activation function that uses the hyperbolic tangent function. The hyperbolic tangent activation function is probably the most commonly used activation function, as it works with both negative and positive numbers. The hyperbolic tangent function is the default activation function for Encog. The equation for the hyperbolic tangent activation function can be seen in Equation 5.4.

Equation 5.4: The Hyperbolic Tangent Activation Function

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The fact that the hyperbolic tangent activation function accepts both positive and negative numbers can be seen in Figure 5.6, which shows the graph of the hyperbolic tangent function.

Figure 5.6: Graph of the Hyperbolic Tangent Activation Function



The hyperbolic tangent function that you see above calls the natural exponent function twice. This is an expensive function call. We really do not need the exact hyperbolic tangent. An approximation will do. The following code does a fast approximation of the hyperbolic tangent function.

```
private double ActivationFunction(double d)
{
  return -1 + (2/ (1+BoundMath.Exp(-2* d ) ) );
}
```

The hyperbolic tangent function is a very common choice for feedforward and simple recurrent neural networks. The hyperbolic tangent function has a derivative, so it can be used with propagation training.

5.3 Summary

Encog uses activation functions to scale the output from neural network layers. By default, Encog will use a hyperbolic tangent function, which is a good general purposes activation function. Any class that acts as an activation function must implement the `IActivationFunction` interface. This interface requires the implementation of several methods. First an `ActivationFunction` method must be created to actually perform the activation function. Secondly, a `DerivativeFunction` method should be implemented to return the derivative of the activation function. If there is no way to take a derivative of the activation function, then an error should be thrown. Only activation functions that have a derivative can be used with propagation training.

The `ActivationBiPolar` activation function class is used when your network only accepts bipolar numbers. The `ActivationCompetitive` activation function class is used for competitive neural networks, such as the Self-Organizing Map. The `ActivationGaussian` activation function class is used when you want a Gaussian curve to represent the activation function. The `ActivationLinear` activation function class is used when you want to have no activation function at all. The `ActivationLOG` activation function class works similarly to the `ActivationTANH` activation function class except it will sometimes not saturate as a hidden layer. The `ActivationSigmoid` activation function class is similar to the `ActivationTANH` activation function class, except only positive numbers are returned. The `ActivationSIN` activation class can be used for periodic data. The `ActivationSoftMax` activation function class scales the output so that the sum is one.

Up to this point we have covered all of the major components of neural networks. Layers contain the neurons and threshold values. Synapses connect the layers together. Activation functions sit inside the layers and scale the output. Tags allow special layers to be identified. Properties allow configuration values to be associated with the neural network. The next chapter will introduce the Encog Workbench. The Encog Workbench is a GUI application that lets you build neural networks composed of all of these elements.

6 Using the Encog Workbench

- Creating a Neural Network
- Creating a Training Set
- Training a Neural Network
- Querying the Neural Network
- Generating Code

An important part of the Encog Framework is the Encog Workbench. The Encog Workbench is a GUI application that can be used to create and edit neural networks. Encog can persist neural networks to .EG files. These files are an XML representation of the neural networks, and other information in which Encog uses to store data.

The Encog workbench can be downloaded from the following URL:

http://www.encog.org

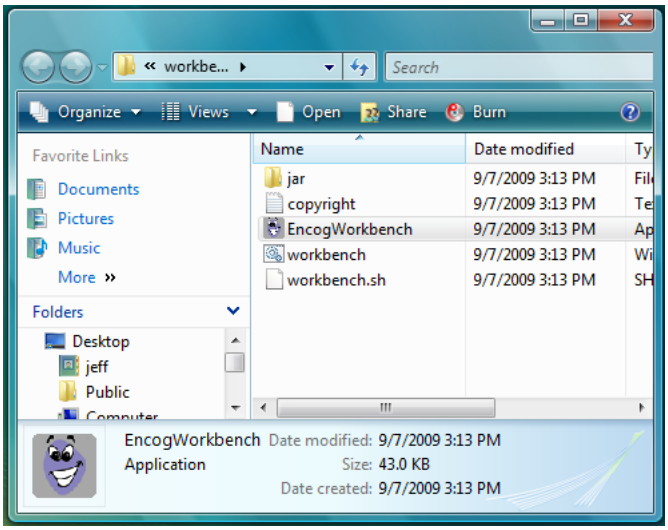
There are several different ways that the Encog Workbench is packaged. Depending on your computer system, you should choose one of the following:

- **Universal** – Packaged with shell scripts and batch files to launch the workbench under UNIX, Macintosh or Windows.
- **Windows Application** – Packaged with a Windows launcher. Simply double click the application executable and the application will start.
- **Macintosh Application** – Packaged with a Macintosh launcher. Simply double click the application icon and the application will start.

In this chapter I will assume that you are using the Windows Application package of Encog Workbench. The others will all operate very similarly. Once you download the Encog workbench and unzip it to a directory, the directory will look similar to Figure 6.1.

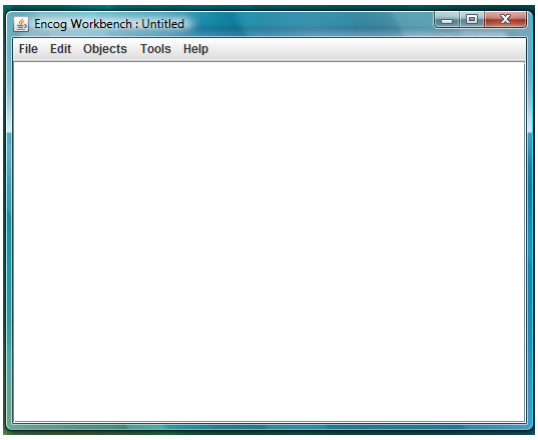
The Encog Workbench was implemented as a Java application. However, it is compatible with the .Net and Silverlight versions of Encog as well. Java was chosen as the language to write the Workbench in due to its ability to run on many different hardware platforms.

Figure 6.1: The Encog Workbench Folder



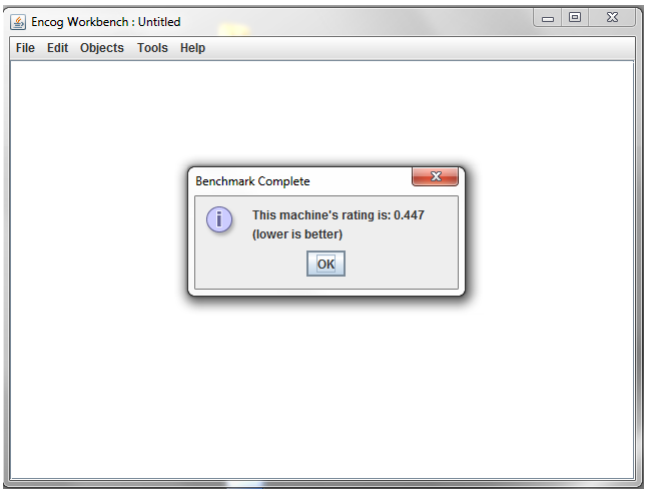
To launch the Encog workbench double click the "Encog Workbench" icon. This will launch the Encog Workbench application. Once the workbench starts, you will see something similar to what is illustrated in Figure 6.2.

Figure 6.2: The Encog Workbench Application



The Encog Workbench can run a benchmark to determine how fast Encog will run on this machine. This may take several minutes, as it runs Encog through a number of different neural network operations. The benchmark is also a good way to make sure that Encog is functioning properly on a computer. To run the benchmark, click the "Tools" menu and select "Benchmark Encog". The benchmark will run and display a progress bar. Once the benchmark is done, you will see the final benchmark number. This can be seen in Figure 6.3.

Figure 6.3: Benchmarking Encog



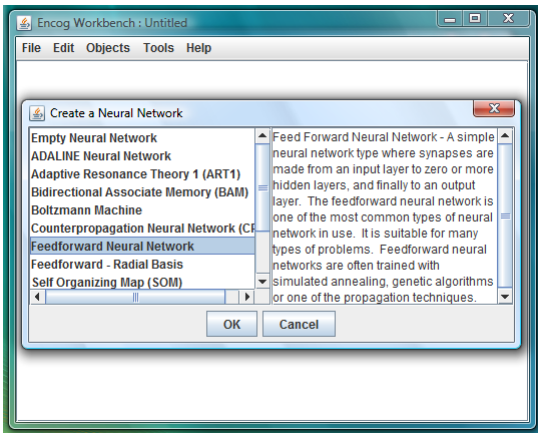
A lower number reflects a better score. The number is the amount of seconds that it took Encog to complete the benchmark tasks. Each part of the benchmark is run multiple times to try to produce consistent benchmark numbers. Encog's use of multicore processors will be reflected in this number. If the computer is already running other processes, this will slow down the benchmark. Because of this, you should not have other applications running while performing a benchmark using the Encog Workbench.

6.1 Creating a Neural Network

We will begin by creating a neural network. The Encog Workbench starts with an empty file. Once objects have been added to this empty file, it can be saved to an .EG file. This .EG file can then be loaded by the workbench again or loaded by Java or .Net Encog applications. The .Net and Java versions of Encog read exactly the same type of .EG files.

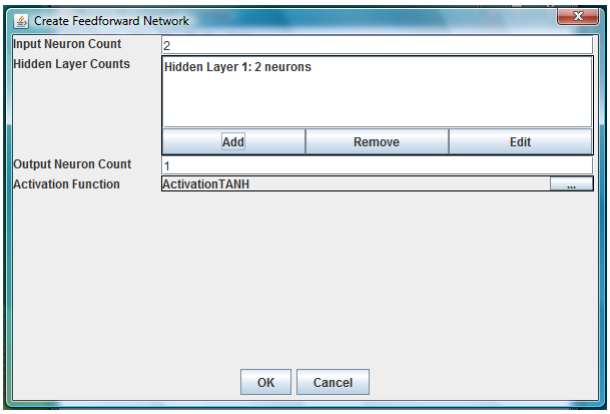
To create a neural network, select "Create Object on the "Objects menu". A small popup window will appear that asks for the type of object to create. Choose "Neural Network" to create a new neural network. This will bring up a window that lets you browse the available types of neural networks to create. These are predefined templates for many of the common neural network types supported by Encog. This window can be seen in Figure 6.4.

Figure 6.4: Create a Neural Network



You will notice that the first option is to create an "Empty Neural Network". Any of the neural networks shown here could be created this way. You would simply create an empty network and add the appropriate layers, synapses, tags and properties to create the neural network type you wish to create. However, if you would like to create one of the common neural network types, it is much faster to simply use one of these predefined templates. Choose the "Feedforward Neural Network". You will need to fill in some information about the type of feedforward neural network you would like to create. This dialog box is seen in Figure 6.5.

Figure 6.5: Create a Feedforward Neural Network



We are going to create a simple, neural network that learns the XOR operator. Such a neural network should be created as follows:

- Input Neuron Count: 2
- Hidden Layer 1 Neuron Count: 2
- Output Neuron Count: 1

The two input neurons are necessary because the XOR operator takes two input parameters. The one output neuron is needed because the XOR operator takes one output parameter. This can be seen from the following truth table for the XOR operator.

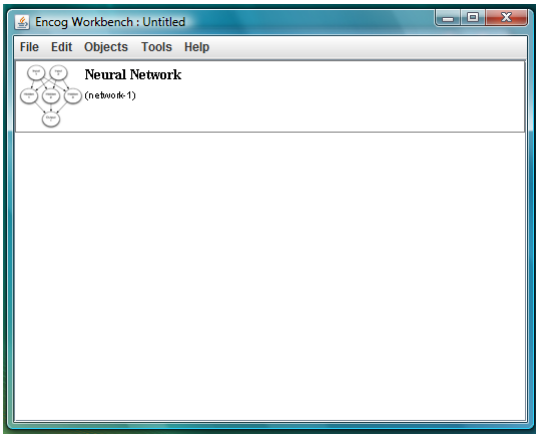
```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

As you can see from the code above, the XOR operator takes two parameters and produces one value. The XOR operator only returns true, or one, when the two input operators are different. This defines the input and output neuron counts.

The hidden layer count is two. The hidden neurons are necessary to assist the neural network in learning the XOR operator. Two is the minimum number of hidden neurons that can be provided for the XOR operator. You may be wondering how we knew to use two. Usually this is something of a trial and error process. You want to choose the minimum number of hidden neurons that still sufficiently solves the problem.

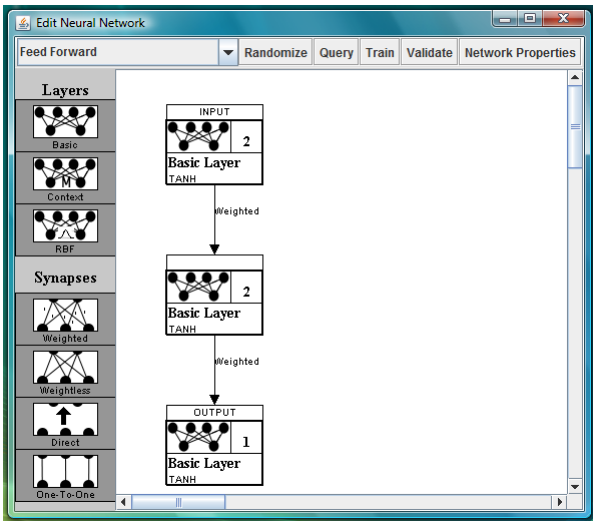
Now that the feedforward neural network has been created, you will see it in the workbench. Figure 6.6 shows the workbench with a neural network added.

Figure 6.6: Neural Network Added



If you double click the feedforward neural network shown in Figure 6.6, it will open. This allows you to see the layers and synapses. Figure 6.7 shows the feedforward neural network that was just created.

Figure 6.7: The Newly Created Neural Network



The above figure shows how neural networks are edited with Encog. You can add additional layers and synapses. You can also edit other aspects of the neural network, such as properties and the type of neural logic that it uses.

Now that the neural network has been created, a training set should be created. The training set will be used to train the neural network.

6.2 Creating a Training Set

A training set is a collection of data to be used to train the neural network. There are two types of training sets commonly used with Encog.

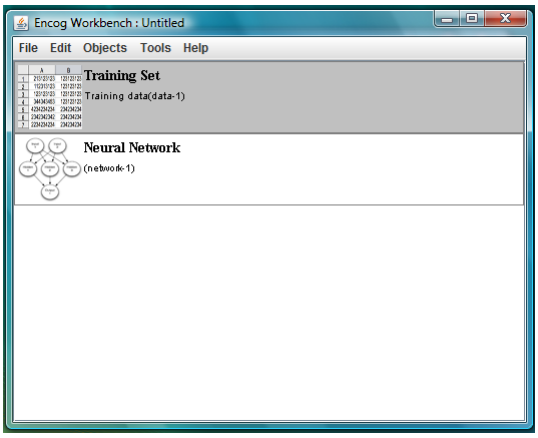
- Supervised Training
- Unsupervised Training

Supervised training data has both an input and expected output specified for the neural network. For example, a truth table above could be represented as a training set. There would be four rows, one for each of the combinations fed to the XOR operator. You would have two input columns and one output column. These correspond to the input and output neurons. The training sets are not concerned with hidden layers. Hidden layers are simply present to assist in learning.

Unsupervised training data only has input values. There are no expected outputs. The neural network will train, in an unsupervised way, and determine for itself what the outputs should be. Unsupervised training is often used for classification problems where you want the neural network to group input data.

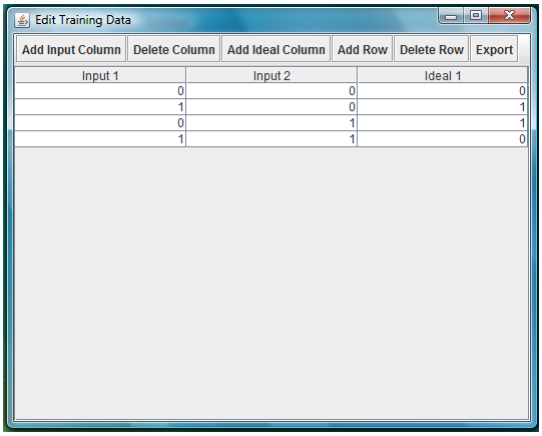
First, we must create a training set. Select "Create Object" from the "Objects" menu. Select a training set. Once the training set has been created it will be added along with the network that was previously created.

Figure 6.8: The Newly Created Training Set



Double clicking the training set will open it. The training set will open in a spreadsheet style window, as seen in Figure 6.9.

Figure 6.9: Editing the Training Set



Here you can see the training set. By default, Encog creates a training set for XOR. This is just the default. Usually you would now create the desired number of input and output columns. However, because we are training the XOR operator, the data is fine as it is.

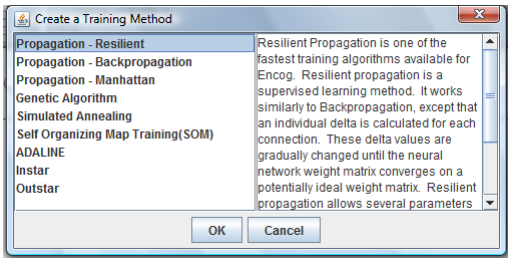
6.3 Training a Neural Network

Training a neural network is a process where the neural network's weights and thresholds are modified so that the neural network will produce output according to the training data. There are many different ways to train a neural network. The choice of training method will be partially determined by the neural network type you are creating. Not all neural network types work with all training methods.

To train the neural network open it as you did for Figure 6.7. Click the “Train” button

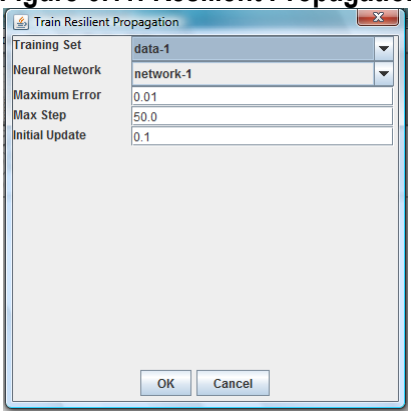
at the top of the window. This will display a dialog box that allows you to choose a training method, as seen in Figure 6.10.

Figure 6.10: Choosing a Training Method



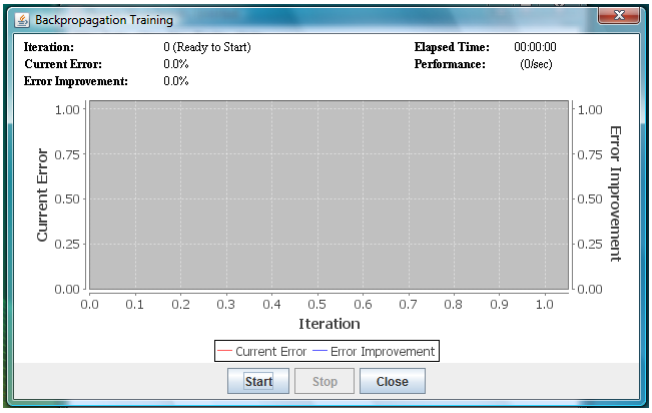
Choose the resilient training method, under propagation. This is usually the best training method available for a supervised feedforward neural network. There are several parameters you can set for the resilient training method. For resilient training it is very unlikely that you should ever change any of these options, other than perhaps the desired maximum error, which defaults to 1%. You can see this dialog box in Figure 6.11.

Figure 6.11: Resilient Propagation Training



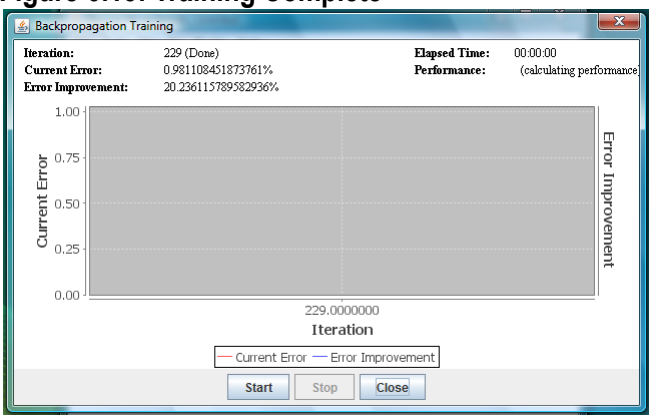
Selecting OK will open a window that will allow you to monitor the training progress, as seen in Figure 6.12.

Figure 6.12: About to Begin Training



To begin training, click the "Start" button on the training dialog box. The network will begin training. For complex networks, this process can go on for days. This is a very simple network that will finish in several hundred iterations. You will not likely even see the graph begin as the training will complete in a matter of seconds. Once the training is complete, you will see the following screen.

Figure 6.13: Training Complete

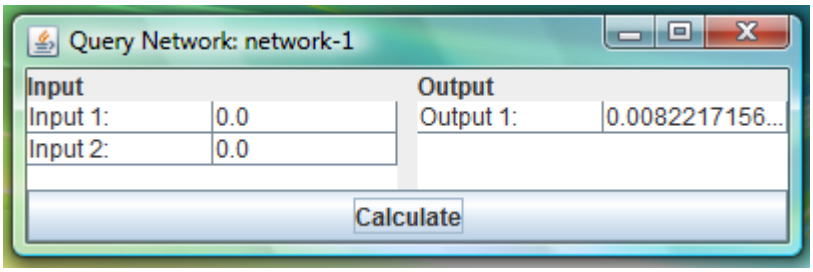


The training is complete because the current error fell below the maximum error allowed that was entered in Figure 6.11, which is 1%. Now that the network has been trained it can produce meaningful output when queried. The training finished very quickly. As a result, there were not enough iterations to draw a chart to show the training progress.

6.4 Querying the Neural Network

Querying the neural network allows you to specify values for the inputs to the neural network and observe the outputs. To query the neural network, click "Query" at the top of the network editor seen in Figure 6.7. This will open the query window as seen in Figure 6.14.

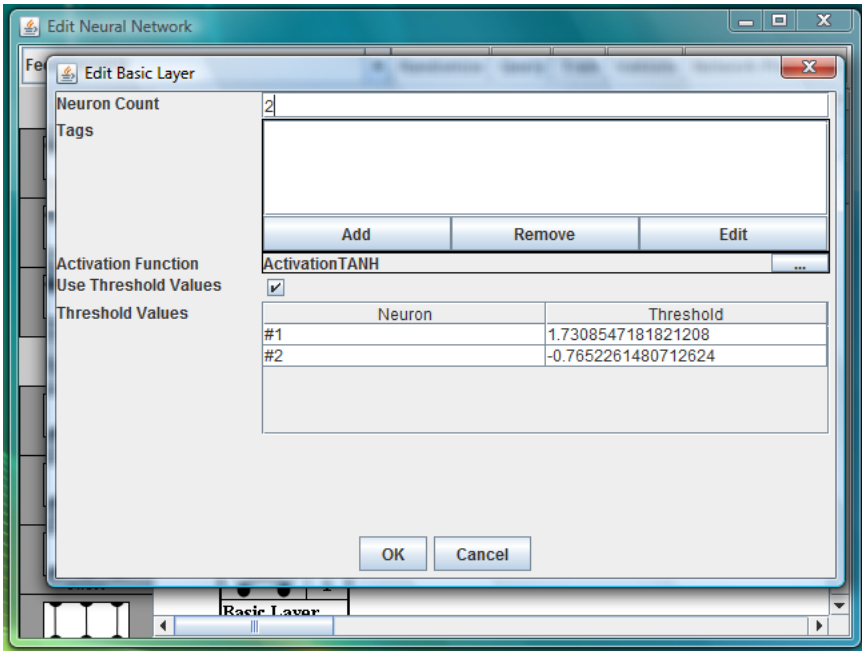
Figure 6.14: Query the Neural Network



As you can see from the above window, you are allowed to enter two values for the input neurons. When you click "Calculate", the output values will be shown. In the example above two zeros were entered, which resulted in 0.008. This is consistent with the XOR operator, as 0.008 is close to zero. To get a value even closer to zero, train the neural network to a lower error rate.

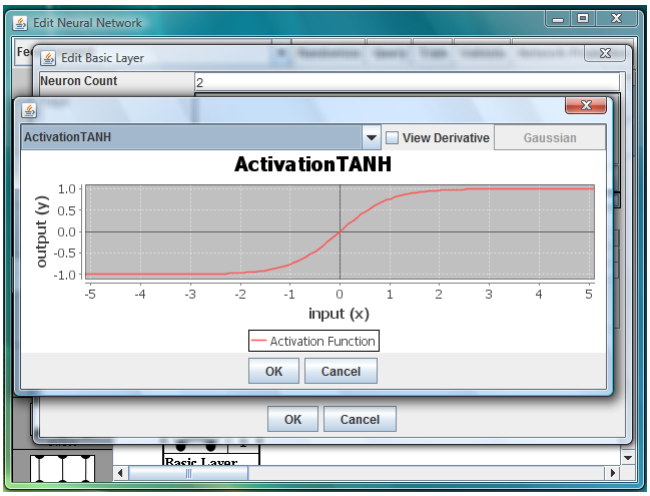
You can also view the weights and threshold values that were generated by the training. From the network editor, shown in Figure 6.7, right click the synapse and choose "Edit Weight Matrix" from the popup menu. Likewise, you can view the thresholds by right-clicking and choosing "Edit Layer" from the pop-up menu. Figure 6.15 shows the dialog used to edit the layer properties.

Figure 6.15: View Layer Properties



You can also browse available activation functions. If you choose to change the activation function you will see something similar to that shown in Figure 6.16.

Figure 6.16: Edit the Activation Function



In Figure 6.16 you can see that the current activation function is the hyperbolic

tangent. The graph for the hyperbolic tangent function is also shown for reference.

6.5 Generating Code

The Encog workbench provides two ways that you can make use of your neural network in C# code. First, you can save the neural network and training data to an .EG file. C# applications can then load data from this .EG file.

Another way to generate code is to use the Encog Workbench. The Encog workbench can generate code in the following languages.

- Java
- C#
- VB.Net

Code generation simply generates the code needed to create the neural network only. No code is generated to train or use the neural network. For the generated program to be of any use, you will need to add your own training code. Listing 6.1 shows the generated C# code from the XOR, feedforward neural network.

Listing 6.1: Generated C# Code

```
using Encog.Neural.Activation;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Layers;
using Encog.Neural.Networks.Synapse;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using org.encog.neural.networks.synapse.WeightedSynapse;

// Neural Network file generated by Encog.  This file shows just // a
simple neural network generated for the structure
// designed in the workbench.
// Additional code will be needed for training and processing.
//
// http://www.encog.org
//

namespace EncogSandbox
{

  public class Program {

    public static void Main()
    {

      BasicNetwork network = new BasicNetwork();

      ILayer outputLayer = new BasicLayer(
        new ActivationSigmoid(),true,1);
      ILayer hiddenLayer1 = new BasicLayer(
```

```
        new ActivationSigmoid(),true,2);
      ILayer inputLayer = new BasicLayer(
        new ActivationSigmoid(),true,2);

      ISynapse synapse1 =
        new WeightedSynapse(hiddenLayer1,outputLayer)
      ISynapse synapse2 =
        new WeightedSynapse(inputLayer,hiddenLayer1)

      hiddenLayer1.AddSynapse(synapse1);
      inputLayer.AddSynapse(synapse2);

      network.TagLayer("INPUT",inputLayer);
      network.TagLayer("OUTPUT",outputLayer);

      network.Structure.FinalizeStructure();
      network.Reset();
    }
  }

}
```

The same network could also have been generated in Java or VB.Net.

6.6 Summary

In this chapter you saw how to use the Encog Workbench. The Encog Workbench provides a way to edit the .EG files produced by the Encog Framework. There are also templates available to help you quickly create common neural network patterns. There is also a GUI network editor that allows networks to be designed using drag and drop functionality.

The workbench allows training data to be created as well. Training data can be manually entered or imported from a CSV file. Training data includes the input to the neural network, as well as the expected output. Training data that only includes input data will be used in unsupervised training. Training data that includes both input and expected output will be used in supervised training.

The neural network can be trained using many different training algorithms. For a feedforward neural network, one of the best choices is the resilient propagation algorithm. The Encog Workbench allows you to enter parameters for the training, and then watch the progress of the training.

The Encog Workbench will generate the code necessary to produce a neural network that was designed with it. The workbench can generate code in Java, C# or VB.Net. This code shows how to construct the neural network with the necessary layers, synapses, properties and layer tags.

The code generated in this chapter was capable of creating the neural network that was designed in the workbench. However, you needed to add your own training code to make the program functional. The next chapter will introduce some of the ways to train a

neural network.

7 Propagation Training

- How Propagation Training Works
- Backpropagation Training
- Manhattan Update Rule
- Resilient Propagation Training

Training is the means by which the weights and threshold values of a neural network are adjusted to give desirable outputs. This document will cover both supervised and unsupervised training. Propagation training is a form of supervised training, where the expected output is given to the training algorithm.

Encog also supports unsupervised training. With unsupervised training you do not provide the neural network with the expected output. Rather, the neural network is left to learn and make insights into the data with limited direction.

Propagation training can be a very effective form of training for feedforward, simple recurrent and other types of neural networks. There are several different forms of propagation training. This chapter will focus on the forms of propagation currently supported by Encog. These three forms are listed as follows:

- Backpropagation Training
- Manhattan Update Rule
- Resilient Propagation Training

All three of these methods work very similarly. However, there are some important differences. In the next section we will explore propagation training in general.

7.1 Understanding Propagation Training

Propagation training algorithms use supervised training. This means that the training algorithm is given a training set of inputs and the ideal output for each input. The propagation-training algorithm will go through a series of iterations. Each iteration will most likely improve the error rate of the neural network by some degree. The error rate is the percent difference between the actual output from the neural network and the ideal output provided by the training data.

Each iteration will completely loop through the training data. For each item of training data, some change to the weight matrix and thresholds will be calculated. These changes will be applied in batches. Encog uses batch training. Therefore, Encog updates the weight matrix and threshold values at the end of an iteration.

We will now examine what happens during each training iteration. Each training iteration begins by looping over all of the training elements in the training set. For each of these training elements a two-pass process is executed: a forward pass and a backward pass.

The forward pass simply presents data to the neural network as it normally would if no

training had occurred. The input data is presented, and the algorithm calculates the error, which is the difference between the actual output and the ideal output. The output from each of the layers is also kept in this pass. This allows the training algorithms to see the output from each of the neural network layers.

The backward pass starts at the output layer and works its way back to the input layer. The backward pass begins by examining the difference between each of the ideal outputs and the actual output from each of the neurons. The gradient of this error is then calculated. To calculate this gradient, the network the actual output of the neural network is applied to the derivative of the activation function used for this level. This value is then multiplied by the error.

Because the algorithm uses the derivative function of the activation function, propagation training can only be used with activation functions that actually have a derivative function. This derivative is used to calculate the error gradient for each connection in the neural network. How exactly this value is used depends on the training algorithm used.

7.1.1 Understanding Backpropagation

Backpropagation is one of the oldest training methods for feedforward neural networks. Backpropagation uses two parameters in conjunction with the gradient descent calculated in the previous section. The first parameter is the learning rate. The learning rate is essentially a percent that determines how directly the gradient descent should be applied to the weight matrix and threshold values. The gradient is multiplied by the learning rate and then added to the weight matrix or threshold value. This will slowly optimize the weights to values that will produce a lower error.

One of the problems with the backpropagation algorithm is that the gradient descent algorithm will seek out local minima. These local minima are points of low error, but they may not be a global minimum. The second parameter provided to the backpropagation algorithm seeks to help the backpropagation out of local minima. The second parameter is called momentum. Momentum specifies, to what degree, the weight changes from the previous iteration should be applied to the current iteration.

The momentum parameter is essentially a percent, just like the learning rate. To use momentum, the backpropagation algorithm must keep track of what changes were applied to the weight matrix from the previous iteration. These changes will be reapplied to the current iteration, except scaled by the momentum parameters. Usually the momentum parameter will be less than one, so the weight changes from the previous training iteration are less significant than the changes calculated for the current iteration. For example, setting the momentum to 0.5 would cause fifty percent of the previous training iteration's changes to be applied to the weights for the current weight matrix.

7.1.2 Understanding the Manhattan Update Rule

One of the problems with the backpropagation training algorithm is the degree to

which the weights are changed. The gradient descent can often apply too large of a change to the weight matrix. The Manhattan update rule and resilient propagation training algorithms only use the sign of the gradient. The magnitude is discarded. This means it is only important if the gradient is positive, negative or near zero.

For the Manhattan update rule, this magnitude is used to determine how to update the weight matrix or threshold value. If the magnitude is near zero, then no change is made to the weight or threshold value. If the magnitude is positive, then the weight or threshold value is increased by a specific amount. If the magnitude is negative, then the weight or threshold value is decreased by a specific amount. The amount by which the weight or threshold value is changed is defined as a constant. You must provide this constant to the Manhattan update rule algorithm.

7.1.3 Understanding Resilient Propagation Training

The resilient propagation training (RPROP) algorithm is usually the most efficient training algorithm provided by Encog for supervised feedforward neural networks. One particular advantage to the RPROP algorithm is that it requires no setting of parameters before using it. There are no learning rates, momentum values or update constants that need to be determined. This is good because it can be difficult to determine the exact learning rate that might be optimal.

The RPROP algorithms works similar to the Manhattan update rule, in that only the magnitude of the descent is used. However, rather than using a fixed constant to update the weights and threshold values, a much more granular approach is used. These deltas will not remain fixed, like in the Manhattan update rule or backpropagation algorithm. Rather these delta values will change as training progresses.

The RPROP algorithm does not keep one global update value, or delta. Rather, individual deltas are kept for every threshold and weight matrix value. These deltas are first initialized to a very small number. Every iteration through the RPROP algorithm will update the weight and threshold values according to these delta values. However, as previously mentioned, these delta values do not remain fixed. The gradient is used to determine how they should change, using the magnitude to determine how the deltas should be modified further. This allows every individual threshold and weight matrix value to be individually trained. This is an advantage that is not provided by either the backpropagation algorithm or the Manhattan update rule.

7.2 Propagation Training with Encog

Now that you understand the primary differences between the three different types of propagation training used by Encog, we will see how to actually implement each of them. The following sections will show C# examples that make use of all three. The XOR operator, which was introduced in the last chapter, will be used as an example. The XOR operator is trivial to implement, so it is a good example for a new training algorithm.

7.2.1 Using Backpropagation

In the last chapter we saw how to use the Encog Workbench to implement a solution with the XOR operator using a neural network. In this chapter we will now see how to do this with a C# program. Listing 7.1 shows a simple C# program that will train a neural network to recognize the XOR operator.

Listing 7.1: Using Backpropagation

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Layers;
using Encog.Neural.Activation;
using Encog.Neural.NeuralData;
using Encog.Neural.Data.Basic;
using Encog.Neural.Networks.Training;
using Encog.Neural.Networks.Training.Propagation.Back;
using Encog.Neural.Data;
using ConsoleExamples.Examples;

namespace Encog.Examples.XOR.Anneal.Backprop
{
    /// <summary>
    /// Learn to recognize the XOR pattern using
    /// a backpropagation training algorithm.
    /// </summary>
    public class XorBackprop : IExample
    {
        public static ExampleInfo Info
        {
            get
            {
                ExampleInfo info = new ExampleInfo(
                    typeof(XorBackprop),
                    "xor-backprop",
                    "XOR Operator with Backpropagation",
              "Use backpropagation to learn the XOR operator.");
                  return info;
            }
        }

        /// <summary>
        /// Input for the XOR function.
        /// </summary>
        public static double[][] XOR_INPUT ={
            new double[2] { 0.0, 0.0 },
            new double[2] { 1.0, 0.0 },
            new double[2] { 0.0, 1.0 },
            new double[2] { 1.0, 1.0 } };

        /// <summary>
        /// Ideal output for the XOR function.
```

```
        /// </summary>
        public static double[][] XOR_IDEAL = {
            new double[1] { 0.0 },
            new double[1] { 1.0 },
            new double[1] { 1.0 },
            new double[1] { 0.0 } };

        /// <summary>
        /// Program entry point.
        /// </summary>
        /// <param name="args">Not used.</param>
        public void Execute(IExampleInterface app)
        {
            BasicNetwork network = new BasicNetwork();
            network.AddLayer(new BasicLayer(
              new ActivationSigmoid(), true, 2));
            network.AddLayer(new BasicLayer(
              new ActivationSigmoid(), true, 3));
            network.AddLayer(new BasicLayer(
              new ActivationSigmoid(), true, 1));
            network.Structure.FinalizeStructure();
            network.Reset();

            INeuralDataSet trainingSet =
              new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

            // train the neural network
             ITrain train =
               new Backpropagation(network, trainingSet,
               0.7, 0.9);

            int epoch = 1;

            do
            {
                train.Iteration();
                Console.WriteLine("Epoch #" + epoch + " Error:"
                  + train.Error);
                epoch++;
            } while ((epoch < 5000) && (train.Error > 0.001));

            // test the neural network
            Console.WriteLine("Neural Network Results:");
            foreach (INeuralDataPair pair in trainingSet)
            {
                INeuralData output =
                  network.Compute(pair.Input);
                Console.WriteLine(pair.Input[0] + ","
                  + pair.Input[1]
                  + ", actual=" + output[0] + ",ideal="
                  + pair.Ideal[0]);
            }

        }
    }
```

```
}
```

We will now examine the parts of the program necessary to implement the XOR backpropagation example.

1.1.1 Truth Table Array

A truth table defines the possible inputs and ideal outputs for a mathematical operator. The truth table for XOR is shown below.

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

The backpropagation XOR example must store the XOR truth table as a 2D array. This will allow a training set to be constructed. We begin by creating **XOR_INPUT**, which will hold the input values for each of the rows in the XOR truth table.

```
public static double[][] XOR_INPUT ={
  new double[2] { 0.0, 0.0 },
  new double[2] { 1.0, 0.0 },
  new double[2] { 0.0, 1.0 },
  new double[2] { 1.0, 1.0 } };
```

Next we create the array **XOR_IDEAL**, which will hold the expected output for each of the inputs previously defined.

```
public static double[][] XOR_IDEAL = {
  new double[1] { 0.0 },
  new double[1] { 1.0 },
  new double[1] { 1.0 },
  new double[1] { 0.0 } };
```

You may wonder why it is necessary to use a 2D array for **XOR_IDEAL**. In this case it looks unnecessary, because the XOR neural network has a single output value. However, neural networks can have many output neurons. Because of this, a 2D array is used to allow each row to potentially have multiple outputs.

7.2.1.1 Constructing the Neural Network

First, the neural network must now be constructed. First we create a **BasicNetwork** class. The **BasicNetwork** class is very extensible. It is currently the only implementation of the more generic **INetwork** interface needed by Encog.

```
BasicNetwork network = new BasicNetwork();
```

This neural network will have three layers. The input layer will have two input neurons, the output layer will have a single output neuron. There will also be a three neuron hidden layer to assist with processing. All three of these layers can use the **BasicLayer** class. This implements a feedforward neural network, or a multilayer perceptron. Each of these layers makes use of the **ActivationSigmoid** activation

function. Sigmoid is a good activation function for XOR because the Sigmoid function only processes positive numbers. Finally, the **true** value specifies that this network should have thresholds.

```
network.AddLayer(new BasicLayer(new ActivationSigmoid(),true,2));
network.AddLayer(new BasicLayer(new ActivationSigmoid(),true,3));
network.AddLayer(new BasicLayer(new ActivationSigmoid(),true,1));
```

Lastly, the neural network structure is finalized. This builds temporary structures to allow the network to be quickly accessed. It is very important that **FinalizeStructure** is always called after the network has been built.

```
network.Structure.FinalizeStructure();
network.Reset();
```

Finally, the **Reset** method is called to initialize the weights and thresholds to random values. The training algorithm will organize these random values into meaningful weights and thresholds that produce the desired result.

7.2.1.2 Constructing the Training Set

Now that the network has been created, the training data must be constructed. We already saw the input and ideal arrays created earlier. Now, we must take these arrays and represent them as **INeuralDataSet**. The following code does this.

```
INeuralDataSet trainingSet = new BasicNeuralDataSet(
  XOR_INPUT, XOR_IDEAL);
```

A **BasicNeuralDataSet** is used, it is one of several training set types that implement the **INeuralDataSet** interface. Other implementations of **INeuralDataSet** can pull data from a variety of abstract sources, such as SQL, HTTP or image files.

7.2.1.3 Training the Neural Network

We now have a **BasicNetwork** object and a **INeuralDataSet** object. This is all that is needed to train a neural network. To implement backpropagation training we instantiate a **Backpropagation** object, as follows.

```
ITrain train = new Backpropagation(network, trainingSet,
  0.7, 0.8);
```

As previously discussed, backpropagation training makes use of a learning rate and a momentum. The value 0.7 is used for the learning rate, the value 0.8 is used for the momentum. Picking proper values for the learning rate and momentum is something of a trial and error process. Too high of a learning rate and the network will no longer decrease its error rate. Too low of a learning rate will take too long to train. If the error rate refuses to lower, even with a lower learning rate, the momentum should be increased to help the neural network get out of a local minimum.

Propagation training is very much an iterative process. The `Iteration` method is called over and over; each time the network is slightly adjusted for a better error rate. The following loop will loop and train the neural network until the error rate has fallen below one percent.

```
do
{
    train.Iteration();
    Console.WriteLine("Epoch #" + epoch + " Error:"
      + train.Error);
    epoch++;
} while ((epoch < 5000) && (train.Error > 0.001));
```

Each trip through the loop is called an epoch, or an iteration. The error rate is the amount that the actual output from the neural network differs from the ideal output provided to the training set.

7.2.1.4 Evaluating the Neural Network

Now that the neural network has been trained, it should be executed to see how well it functions. We begin by displaying a heading as follows:.

```
Console.WriteLine("Neural Network Results:");
```

We will now loop through each of the training set elements. A `INeuralDataSet` is made up of a collection of `INeuralDataPair` classes. Each `INeuralDataPair` class contains an input and an ideal property. Each of these two properties is a `INeuralData` object that essentially contains an array. This is how Encog stores the training data. We begin by looping over all of the `INeuralDataPair` objects contained in the `INeuralDataSet` object.

```
foreach (INeuralDataPair pair in trainingSet)
{
```

For each of the `INeuralDataPair` objects, we compute the neural network's output using the input property of the `INeuralDataPair` object.

```
INeuralData output = network.Compute(pair.Input);
```

We now display the ideal output, as well as the actual output for the neural network.

```
  Console.WriteLine(pair.Input[0] + ","
    + pair.Input[1]
    + ", actual=" + output[0] + ",ideal="
    + pair.Ideal[0]);
}
```

The output from this neural network is shown here.

```
Epoch #1 Error:0.504998283847474
Epoch #2 Error:0.504948046227928
```

```
Epoch #3 Error:0.5028968616826613
Epoch #4 Error:0.5034596686580215
Epoch #5 Error:0.5042340438643891
Epoch #6 Error:0.5034282078077391
Epoch #7 Error:0.501995999394481
Epoch #8 Error:0.5014532303103851
Epoch #9 Error:0.5016773751196401
Epoch #10 Error:0.5016348354128658
...
Epoch #3340 Error:0.01000800225100623
Epoch #3341 Error:0.010006374293649473
Epoch #3342 Error:0.01000474710532496
Epoch #3343 Error:0.010003120685432222
Epoch #3344 Error:0.010001495033371149
Epoch #3345 Error:0.009999870148542572
Neural Network Results:
0.0,0.0, actual=0.010977229866756838,ideal=0.0
1.0,0.0, actual=0.9905671966735671,ideal=1.0
0.0,1.0, actual=0.989931152973507,ideal=1.0
1.0,1.0, actual=0.009434016119752921,ideal=0.0
```

First, you will see the training epochs counting upwards and decreasing the error. The error starts out at 0.50, which is just above 50%. At epoch 3,345, the error has dropped below one percent and training can stop.

The program then evaluates the neural network by cycling through the training data and presenting each training element to the neural network. You will notice from the above data that the results do not exactly match the ideal results. For instance the value 0.0109 does not exactly match 0.0. However, it is close. Remember that the network was only trained to a one percent error. As a result, the data is not going to match precisely.

In this example, we are evaluating the neural network with the very data that it was trained with. This is fine for a simple example, where we only have four training elements. However, you will usually want to hold back some of your data to with which to validate the neural network. Validating the network with the same data that it was trained with does not prove much. However, validating good results with data other than what the neural network was trained with proves that the neural network has gained some sort of an insight into the data that it is processing.

Something else that is interesting to note is the number of iterations it took to get an acceptable error. Backpropagation took 3,345 iterations to get to an acceptable error. Different runs of this example produce different results, as we are starting from randomly generated weights and thresholds. However, the number 3,345 is a fairly good indication of the efficiency of the backpropagation algorithm. This number will be compared to the other propagation training algorithms.

7.2.2 Using the Manhattan Update Rule

Next, we will look at how to implement the Manhattan update rule. There are very few changes that are needed to the backpropagation example to cause it to use the

Manhattan update rule. Listing 7.2 shows the complete Manhattan update rule example.

Listing 7.2: Using the Manhattan Update Rule

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Layers;
using Encog.Neural.Activation;
using Encog.Neural.NeuralData;
using Encog.Neural.Data.Basic;
using Encog.Neural.Networks.Training;
using Encog.Neural.Data;
using Encog.Neural.Networks.Training.Propagation.Manhattan;
using ConsoleExamples.Examples;

namespace Encog.Examples.XOR.Manhattan
{
    /// <summary>
    /// Learn to recognize the XOR pattern using a
    /// Manhattan update rule training algorithm.
    /// </summary>
    public class XORManhattan:IExample
    {
        public static ExampleInfo Info
        {
            get
            {
                ExampleInfo info = new ExampleInfo(
                    typeof(XORManhattan),
                    "xor-manhattan",
                    "XOR Operator with Manhattan Update Rule",
     "Use the Manhattan Update Rule to learn the XOR operator.");
                return info;
            }
        }

        /// <summary>
        /// Input for the XOR function.
        /// </summary>
        public static double[][] XOR_INPUT ={
            new double[2] { 0.0, 0.0 },
            new double[2] { 1.0, 0.0 },
            new double[2] { 0.0, 1.0 },
            new double[2] { 1.0, 1.0 } };

        /// <summary>
        /// Ideal output for the XOR function.
        /// </summary>
        public static double[][] XOR_IDEAL = {
            new double[1] { 0.0 },
            new double[1] { 1.0 },
            new double[1] { 1.0 },
            new double[1] { 0.0 } };
```

```
        /// <summary>
        /// Program entry point.
        /// </summary>
        /// <param name="args">Not used.</param>
        public void Execute(IExampleInterface app)
        {
            BasicNetwork network = new BasicNetwork();
            network.AddLayer(
              new BasicLayer(new ActivationSigmoid(), true, 2));
            network.AddLayer(
              new BasicLayer(new ActivationSigmoid(), true, 3));
            network.AddLayer(
              new BasicLayer(new ActivationSigmoid(), true, 1));
            network.Structure.FinalizeStructure();
            network.Reset();

            INeuralDataSet trainingSet =
              new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

            // train the neural network
            ITrain train =
              new ManhattanPropagation(network,
              trainingSet, 0.0001);

            int epoch = 1;

            do
            {
                train.Iteration();
                Console.WriteLine(
                  "Epoch #" + epoch + " Error:" + train.Error);
                epoch++;
            } while (train.Error > 0.001);

            // test the neural network
            Console.WriteLine("Neural Network Results:");
            foreach (INeuralDataPair pair in trainingSet)
            {
                INeuralData output =
                  network.Compute(pair.Input);
                Console.WriteLine(pair.Input[0] + ","
                  + pair.Input[1]
                  + ", actual=" + output[0]
                  + ",ideal=" + pair.Ideal[0]);
            }

        }
    }
}
```

There is really only one line that has changed from the backpropagation example. Because the **ManhattanPropagation** object uses the same **ITrain** interface, there are very few changes needed. We simply create a

ManhattanPropagation object in place of the **Backpropagation** class that was used in the previous section.

```
ITrain train =
  new ManhattanPropagation(network,
  trainingSet, 0.0001);
```

As previously discussed, the Manhattan update rule works by using a single constant value to adjust the weights and thresholds. This is usually a very small number so as not to introduce rapid of change into the network. For this example, the number 0.0001 was chosen. Picking this number usually comes down to trial and error, as was the case with backpropagation. A value that is too high causes the network to change randomly and never converge to a number.

The Manhattan update rule will tend to behave somewhat randomly at first. The error rate will seem to improve and then worsen. But it will gradually trend lower. After 710,954 iterations the error rate is acceptable.

```
Epoch #710941 Error:0.011714647667850289
Epoch #710942 Error:0.011573263349587842
Epoch #710943 Error:0.011431878106128258
Epoch #710944 Error:0.011290491948778713
Epoch #710945 Error:0.011149104888883382
Epoch #710946 Error:0.011007716937768005
Epoch #710947 Error:0.010866328106765183
Epoch #710948 Error:0.010724938407208937
Epoch #710949 Error:0.010583547850435736
Epoch #710950 Error:0.010442156447783919
Epoch #710951 Error:0.010300764210593727
Epoch #710952 Error:0.01015937115020837
Epoch #710953 Error:0.010017977277972472
Epoch #710954 Error:0.009876582605234318
Neural Network Results:
0.0,0.0, actual=-0.013777528025884167,ideal=0.0
1.0,0.0, actual=0.9999999999999925,ideal=1.0
0.0,1.0, actual=0.9999961061923577,ideal=1.0
1.0,1.0, actual=-0.013757731687977337,ideal=0.0
```

As you can see the Manhattan update rule took considerably more iterations to find a solution than the backpropagation. There are certain cases where the Manhattan rule is preferable to backpropagation training. However, for a simple case like the XOR problem, backpropagation is a better solution than the Manhattan rule. Finding a better delta value may improve the efficiency of the Manhattan update rule.

7.2.3 Using Resilient Propagation

One of the most difficult aspects of the backpropagation and the Manhattan update rule learning is picking the correct training parameters. If a bad choice is made for the learning rate, training momentum or delta values will not be as successful as it might have been. Resilient propagation does have training parameters, but it is extremely rare that they need to be changed from their default values. This makes resilient propagation

a very easy way to use a training algorithm. Listing 7.3 shows an XOR example using the resilient propagation algorithm.

Listing 7.3: Using Resilient Propagation

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Encog.Neural.Networks;
using Encog.Neural.Networks.Layers;
using Encog.Neural.Activation;
using Encog.Neural.Data.Basic;
using Encog.Neural.NeuralData;
using Encog.Neural.Networks.Training;
using Encog.Neural.Data;
using Encog.Neural.Networks.Training.Propagation.Resilient;
using ConsoleExamples.Examples;

namespace Encog.Examples.XOR.Resilient
{
    /// <summary>
    /// XOR: This example is essentially the "Hello World" of
    /// neural network
    /// programming.  This example shows how to construct an
    /// Encog neural
    /// network to predict the output from the XOR operator.
    /// This example
    /// uses RPROP to train the neural network.
    /// </summary>
    public class XORResilient : IExample
    {
        public static ExampleInfo Info
        {
            get
            {
                ExampleInfo info = new ExampleInfo(
                    typeof(XORResilient),
                    "xor-rprop",
                    "XOR Operator with Resilient Propagation",
                    "Use RPROP to learn the XOR operator.");
                return info;
            }
        }
        /// <summary>
        /// Input for the XOR function.
        /// </summary>
        public static double[][] XOR_INPUT ={
            new double[2] { 0.0, 0.0 },
            new double[2] { 1.0, 0.0 },
                  new double[2] { 0.0, 1.0 },
            new double[2] { 1.0, 1.0 } };

        /// <summary>
        /// Ideal output for the XOR function.
        /// </summary>
```

```
        public static double[][] XOR_IDEAL = {
            new double[1] { 0.0 },
            new double[1] { 1.0 },
            new double[1] { 1.0 },
            new double[1] { 0.0 } };

        /// <summary>
        /// Program entry point.
        /// </summary>
        /// <param name="args">Not used.</param>
        public void Execute(IExampleInterface app)
        {
            BasicNetwork network = new BasicNetwork();
            network.AddLayer(
              new BasicLayer(new ActivationSigmoid(), true, 2));
            network.AddLayer(
              new BasicLayer(new ActivationSigmoid(), true, 6));
            network.AddLayer(
              new BasicLayer(new ActivationSigmoid(), true, 1));
            network.Structure.FinalizeStructure();
            network.Reset();

            INeuralDataSet trainingSet =
              new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

            // train the neural network
            // train the neural network
            ITrain train =
              new ResilientPropagation(network, trainingSet);

            int epoch = 1;

            do
            {
                train.Iteration();
                Console.WriteLine("Epoch #" + epoch
                  + " Error:" + train.Error);
                epoch++;
            } while ((epoch < 5000) && (train.Error > 0.001));

            // test the neural network
            Console.WriteLine("Neural Network Results:");
            foreach (INeuralDataPair pair in trainingSet)
            {
                INeuralData output =
                network.Compute(pair.Input);
                Console.WriteLine(pair.Input[0] + ","
                + pair.Input[1]
                + ", actual=" + output[0] + ",ideal="
                + pair.Ideal[0]);
            }

        }
    }
}
```

The following line of code creates a **ResilientPropagation** object that will be used to train the neural network.

```
ITrain train =
new ResilientPropagation(network, trainingSet);
```

As you can see there are no training parameters provided to the **ResilientPropagation** object. Running this example program will produce the following results.

```
Epoch #1 Error:0.5108505683309112
Epoch #2 Error:0.5207537811846186
Epoch #3 Error:0.5087933421445957
Epoch #4 Error:0.5013907858935785
Epoch #5 Error:0.5013907858935785
Epoch #6 Error:0.5000489677062201
Epoch #7 Error:0.49941437656150733
Epoch #8 Error:0.49798185395576444
Epoch #9 Error:0.4980795840636415
Epoch #10 Error:0.4973134271412919
...
Epoch #270 Error:0.010865894525995278
Epoch #271 Error:0.010018272841993655
Epoch #272 Error:0.010068462218315439
Epoch #273 Error:0.009971267210982099
Neural Network Results:
0.0,0.0, actual=0.00426845952539745,ideal=0.0
1.0,0.0, actual=0.9849930511468161,ideal=1.0
0.0,1.0, actual=0.9874048605752819,ideal=1.0
1.0,1.0, actual=0.0029321659866812233,ideal=0.0
```

Not only is the resilient propagation algorithm easier to use, it is also considerably more efficient than backpropagation or the Manhattan update rule.

7.3 Propagation and Multithreading

As of the writing of this document, single core computers are becoming much less common than multi core computers. A dual core computer effectively has two complete processors in a single chip. Quadcore computers have four processors on a single chip. The latest generation of Quadcores, the Intel i7, comes with hyperthreading as well. Hyperthreading allows one core processor to appear as two by simultaneously executing multiple instructions. A computer that uses hyperthreading technology will actually report twice the number of cores that is actually installed.

Processors seem to have maxed out their speeds at around 3 gigahertz. Growth in computing power will not be in the processing speed of individual processors. Rather, future growth will be in the number of cores a computer has. However, taking advantage of these additional cores can be a challenge for the computer programmer. To take advantage of these cores you must write multithreaded software.

Entire books are written on multithreaded programming, so it will not be covered in

depth here. However, the general idea is to take a large problem and break it down into manageable pieces that be executed independently by multiple threads. The final solution must then be pieced back together from each of the threads. This process is called aggregation.

Encog makes use of multithreading in many key areas. One such area is training. By default the propagation training techniques will use multithreading if it appears that multithreading will help performance. Specifically, there should be more than one core and sufficient training data for multithreading to be worthwhile. If both of these elements are present, any of the propagation techniques will make use of multithreading.

It is possible to tell Encog to use a specific number of threads, or disable threading completely. The `NumThreads` property provided by all of the propagation training algorithms does this. To run in single threaded mode, specify one thread. To specify a specific number of threads specify the number of threads desired. Finally, to allow Encog to determine the optimal number of threads, specify zero threads. Zero is the default value for the number of threads.

When Encog is requested to determine the optimal number of threads to use, several things are considered. Encog considers the number of cores that are available. Encog also considers the size of the training data. Multithreaded training works best with larger training sets.

7.3.1 How Multithreaded Training Works

Multithreaded training works particularly well with larger training sets and machines multiple cores. If Encog does not detect that both are present, it will fall back to single threaded. When there is more than one processing core, and enough training set items to keep both cores busy, multithreaded training will function significantly faster than single threaded.

We've already looked at three propagation-training techniques. All propagation-training techniques work similarly. Whether it is backpropagation, resilient propagation or the Manhattan update rule, the technique is similar. There are two three distinct steps:

1. Perform a Regular Feed Forward Pass.

2. Process the levels backwards, and determine the errors at each level.

3. Apply the changes to the weights and thresholds.

First, a regular feed forward pass is performed. The output from each level is kept so the error for each level can be evaluated independently. Second, the errors are calculated at each level, and the derivatives of each of the activation functions are used to calculate gradient descents. These gradients show the direction that the weight must be modified to improve the error of the network. These gradients will be used in the third step.

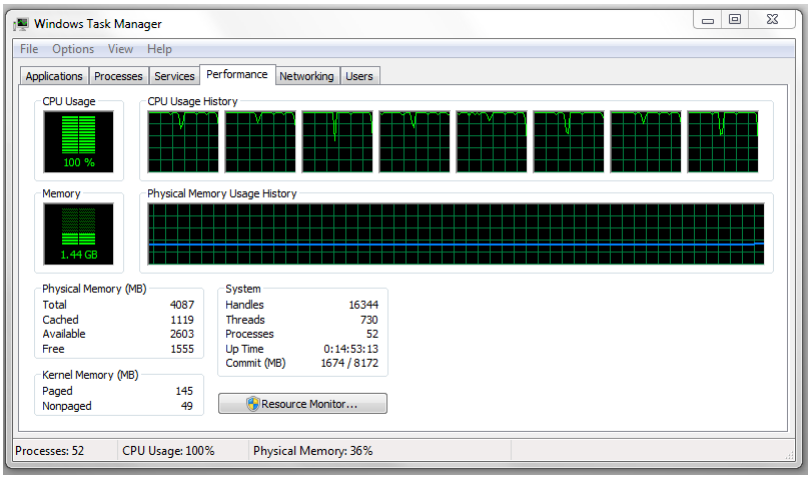
The third step is what varies among the different training algorithms. Backpropagation simply takes the gradient descents and scales them by a learning rate. The scaled gradient

descents are then directly applied to the weights and thresholds. The Manhattan Update Rule only uses the sign of the gradient to decide in which direction to affect the weight. The weight is then changed in either the positive or negative direction by a fixed constant.

RPROP keeps an individual delta value for every weight and thresholds and only uses the sign of the gradient descent to increase or decrease the delta amounts. The delta amounts are then applied to the weights and thresholds.

The multithreaded algorithm uses threads to perform Steps 1 and 2. The training data is broken into packets that are distributed among the threads. At the beginning of each iteration, threads are started to handle each of these packets. Once all threads have completed, a single thread aggregates all of the results from the threads and applies them to the neural network. There is a very brief amount of time where only one thread is executing, at the end of the iteration. This can be seen from Figure 7.1.

Figure 7.1: Encog Training on a Hyperthreaded Quadcore



As you can see from the above image, the i7 is currently running at 100%. You can clearly see the end of each iteration, where each of the processors falls briefly. Fortunately, this is a very brief time, and does not have a large impact on overall training efficiency. I did try implementations where I did not force the threads to wait at the end of the iteration for a resynchronization. However, these did not provide efficient training because the propagation training algorithms need all changes applied before the next iteration begins.

7.3.2 Using Multithreaded Training

To see multithreaded training really shine, a larger training set is needed. In the next chapter we will see how to gather information for Encog, and larger training sets will be used. However, for now, we will look a simple benchmarking example that generates a random training set and compares multithreaded and single-threaded training times.

A simple benchmark is shown that makes use of an input layer of 40 neurons, a hidden

layer of 60 neurons, and an output layer of 20 neurons. A training set of 50,000 elements is used. This example is shown in Listing 7.4.

Listing 7.4: Using Multithreaded Training

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ConsoleExamples.Examples;
using Encog.Neural.Networks;
using Encog.Neural.NeuralData;
using Encog.Neural.Networks.Layers;
using Encog.Util.Banchmark;
using Encog.Neural.Networks.Training.Propagation.Resilient;
using Encog.Util.Logging;

namespace Encog.Examples.MultiBench
{
    public class MultiThreadBenchmark:IExample
    {
        public const int INPUT_COUNT = 40;
        public const int HIDDEN_COUNT = 60;
        public const int OUTPUT_COUNT = 20;

        private IExampleInterface app;

        public static ExampleInfo Info
        {
            get
            {
                ExampleInfo info = new ExampleInfo(
                    typeof(MultiThreadBenchmark),
                    "multibench",
                    "Multithreading Benchmark",
     "See the effects that multithreading has on performance.");
                return info;
            }
        }


        public BasicNetwork generateNetwork()
        {
            BasicNetwork network = new BasicNetwork();
            network.AddLayer(
              new BasicLayer(MultiThreadBenchmark.INPUT_COUNT));
            network.AddLayer(
              new BasicLayer(MultiThreadBenchmark.HIDDEN_COUNT));
            network.AddLayer(
              new BasicLayer(MultiThreadBenchmark.OUTPUT_COUNT));
            network.Structure.FinalizeStructure();
            network.Reset();
            return network;
        }

        public INeuralDataSet generateTraining()
```

```
        {
            INeuralDataSet training =
              RandomTrainingFactory.Generate(50000,
              INPUT_COUNT, OUTPUT_COUNT, -1, 1);
            return training;
        }

        public double evaluateRPROP(
          BasicNetwork network, INeuralDataSet data)
        {

            ResilientPropagation train =
              new ResilientPropagation(network, data);
            long start = DateTime.Now.Ticks;
            Console.WriteLine(
              "Training 20 Iterations with RPROP");
            for (int i = 1; i <= 1; i++)
            {
                train.Iteration();
                Console.WriteLine("Iteration #" + i + " Error:"
                  + train.Error);
            }
            //train.FinishTraining();
            long stop = DateTime.Now.Ticks;
            double diff = new TimeSpan(stop - start).Seconds;
            Console.WriteLine("RPROP Result:" + diff
              + " seconds.");
            Console.WriteLine("Final RPROP error: "
              + network.CalculateError(data));
            return diff;
        }

        public double evaluateMPROP(
          BasicNetwork network, INeuralDataSet data)
        {

            ResilientPropagation train =
              new ResilientPropagation(network, data);
            long start = DateTime.Now.Ticks;
            Console.WriteLine(
              "Training 20 Iterations with MPROP");
            for (int i = 1; i <= 20; i++)
            {
                train.Iteration();
                Console.WriteLine("Iteration #" + i + " Error:"
                  + train.Error);
            }
            //train.finishTraining();
            long stop = DateTime.Now.Ticks;
            double diff = new TimeSpan(stop - start).Seconds;
            Console.WriteLine("MPROP Result:"
              + diff + " seconds.");
            Console.WriteLine("Final MPROP error: "
              + network.CalculateError(data));
            return diff;
```

```
        }
        public void Execute(IExampleInterface app)
        {
            this.app = app;
            Logging.StopConsoleLogging();
            BasicNetwork network = generateNetwork();
            INeuralDataSet data = generateTraining();

            double rprop = evaluateRPROP(network, data);
            double mprop = evaluateMPROP(network, data);
            double factor = rprop / mprop;
            Console.WriteLine("Factor improvement:" + factor);
        }
    }
}
```

I executed this program on a Quadcore i7 with Hyperthreading. The following was the result.

```
Training 20 Iterations with Single-threaded
Iteration #1 Error:1.0594453784075148
Iteration #2 Error:1.0594453784075148
Iteration #3 Error:1.0059791059086385
Iteration #4 Error:0.955845375587124
Iteration #5 Error:0.934169803870454
Iteration #6 Error:0.9140418793336804
Iteration #7 Error:0.8950880473422747
Iteration #8 Error:0.8759150228219456
Iteration #9 Error:0.8596693523930371
Iteration #10 Error:0.843578483629412
Iteration #11 Error:0.8239688415389107
Iteration #12 Error:0.8076160458145523
Iteration #13 Error:0.7928442431442133
Iteration #14 Error:0.7772585699972144
Iteration #15 Error:0.7634533283610793
Iteration #16 Error:0.7500401666509937
Iteration #17 Error:0.7376158116045242
Iteration #18 Error:0.7268954113068246
Iteration #19 Error:0.7155784667628093
Iteration #20 Error:0.705537166118038
RPROP Result:35.134 seconds.
Final RPROP error: 0.6952141684716632
Training 20 Iterations with Multithreading
Iteration #1 Error:0.6952126315707992
Iteration #2 Error:0.6952126315707992
Iteration #3 Error:0.90915249248788
Iteration #4 Error:0.8797061675258835
Iteration #5 Error:0.8561169673033431
Iteration #6 Error:0.7909509694056177
Iteration #7 Error:0.7709539415065737
Iteration #8 Error:0.7541971172618358
Iteration #9 Error:0.7287094412886507
Iteration #10 Error:0.715814914438935
Iteration #11 Error:0.7037730808705016
Iteration #12 Error:0.6925902585055886
```

```
Iteration #13 Error:0.6784038181007823
Iteration #14 Error:0.6673310323078667
Iteration #15 Error:0.6585209150749294
Iteration #16 Error:0.6503710867148986
Iteration #17 Error:0.6429473784897797
Iteration #18 Error:0.6370962075614478
Iteration #19 Error:0.6314478792705961
Iteration #20 Error:0.6265724296587237
Multi-Threaded Result:8.793 seconds.
Final Multi-thread error: 0.6219704300851074
Factor improvement:4.0106783805299674
```

As you can see from the above results, the single threaded RPROP algorithm finished in 128 seconds, the multithreaded RPROP algorithm finished in only 31 seconds. Multithreading improved performance by a factor of four. Your results running the above example will depend on how many cores your computer has. If your computer is single core, with no hyperthreading, then the factor will be close to one. This is because the second multi-threading training will fall back to a single thread.

7.4 Summary

In this chapter you saw how to use three different propagation algorithms with Encog. Propagation training is a very common class of supervised training algorithms. In this chapter you saw how to use three different propagation training algorithms. Resilient propagation training is usually the best choice; however; the Manhattan update rule and backpropagation may be useful for certain situations.

Backpropagation was one of the original training algorithms for feedforward neural networks. Though Encog supports it mostly for historic purposes, it can sometimes be used to further refine a neural network after resilient propagation has been used. Backpropagation uses a learning rate and momentum. The learning rate defines how quickly the neural network will learn; the momentum helps the network get out of local minima.

The Manhattan update rule uses a delta value to change update the weight and threshold values. It can be difficult to choose this delta value correctly. Too high of a value will cause the network to learn nothing at all.

Resilient propagation (RPROP) is one of the best training algorithms offered by Encog. It does not require you to provide training parameters, like the other two propagation-training algorithms. This makes it much easier to use. Additionally, resilient propagation is considerably more efficient than Manhattan update rule or backpropagation.

Multithreaded training is a training technique that adapts propagation training to perform faster with multicore computers. Given a computer with multiple cores and a large enough training set, multithreaded training is considerably faster than single-threaded training. Encog can automatically set an optimal number of threads. If these conditions are not present, Encog will fall back to single threaded training.

This document provided an introduction into Encog programming. For a more advanced, and lengthy, manual on Encog programming you may be interested in my book "Programming Neural Networks with Encog 2 in C#". ISBN 1604390107. This book is available in both paperback and ebook formats. This book adds many chapters and is nearly 500 pages long. Your purchase of our book supports the Encog project! For more information visit.

http://www.heatonresearch.com/book/programming-neural-networks-encog-cs.html