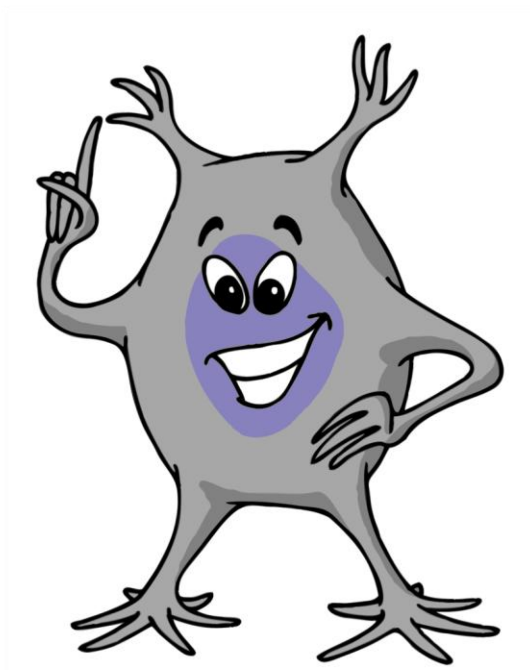
Introduction to Encog 2.2 for Java

Revision 1



http://www.heatonresearch.com/encog

http://www.heatonresearch.com

This ebook serves as the primary documentation for the Encog Artificial Intelligence framework for Java. This ebook shows how to install, configure and use the basics of Encog. I will, at some point, have commercial books available for Encog that will cover neural network programming in even greater depth. However, this ebook should contain enough information to begin constructing neural networks with Encog.

Contents

1 What is Encog? 5
1.1 The History of Encog 5
1.1.1 Problem Solving with Neural Networks 6
1.1.2 Problems Not Suited to a Neural Network Solution 6
1.1.3 Problems Suited to a Neural Network 6
2 Installing and Using Encog 8
2.1 Installing Encog 8
2.2 Compiling the Encog Core 9
2.3 Compiling and Executing Encog Examples 11
2.3.1 Third-Party Libraries Used by Encog 11
2.3.2 Running an Example from the Command Line 12
2.4 Using Encog with the Eclipse IDE 13
2.4.1 Resolving Path Errors 15
2.4.2 Running an Example 16
3 Introduction to Encog 18
3.1 What is a Neural Network? 18
3.1.1 Understanding the Input Layer 20
3.1.2 Understanding the Output Layer 21
3.1.3 Hidden Layers 21
3.1.4 Using a Neural Network 22
3.1.5 The XOR Operator and Neural Networks 22
3.2 Structuring a Neural Network for XOR 23
3.3 Training a Neural Network 25
3.4 Executing a Neural Network 27
3.5 Chapter Summary 30
4 Using Activation Functions 32
4.1 The Role of Activation Functions 32
4.1.1 The ActivationFunction Interface 32
4.1.2 Derivatives of Activation Functions 33
4.2 Encog Activation Functions 34
4.2.1 ActivationBiPolar 34
4.2.2 Activation Competitive 34

4.2.3 ActivationGaussian ... 35
4.3 ActivationLinear ... 36
4.4 ActivationLOG ... 37
4.5 ActivationSigmoid ... 38
4.6 ActivationSIN ... 39
4.7 ActivationSoftMax ... 40
4.8 ActivationTANH ... 41
4.9 Summary ... 42
5 Using the Encog Workbench ... 43
5.1 Creating a Neural Network ... 45
5.2 Creating a Training Set ... 48
5.3 Training a Neural Network ... 49
5.4 Querying the Neural Network ... 51
5.5 Generating Code ... 52
5.6 Summary ... 54
6 Propagation Training ... 55
6.1 Understanding Propagation Training ... 55
6.1.1 Understanding Backpropagation ... 56
6.1.2 Understanding the Manhattan Update Rule ... 57
6.1.3 Understanding Resilient Propagation Training ... 57
6.2 Propagation Training with Encog ... 58
6.2.1 Using Backpropagation ... 58
6.2.2 Truth Table Array ... 59
6.2.3 Constructing the Neural Network ... 60
6.2.4 Constructing the Training Set ... 61
6.2.5 Training the Neural Network ... 61
6.2.6 Evaluating the Neural Network ... 62
6.3 Using the Manhattan Update Rule ... 64
6.4 Using Resilient Propagation ... 67
6.5 Propagation and Multithreading ... 69
6.5.1 How Multipropagation Works ... 70
6.5.2 Using Multipropagation ... 71
6.6 Summary ... 76

1 What is Encog?

Encog is an Artificial Intelligence (AI) Framework for Java and DotNet. 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 book, 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 Java programming language. Particularly, you should be familiar with the following:

- Java 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.

1.1.1 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.

1.1.2 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.

1.1.3 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 book 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 appendix 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 Java. This appendix assumes that you have already downloaded and installed Java JSE version 1.6 or later on your computer. The latest version of Java can be downloaded from the following Web site:

http://java.sun.com/

Java is a cross-platform programming language, so Encog can run on a variety of platforms. Encog has been used on Macintosh and Linux operating systems. However, this appendix assumes that you are using the Windows operating system. The screen shots illustrate procedures on the Windows 7 operating. However, Encog should run just fine on Windows XP or later.

It is also possible to use Encog with an IDE. Encog was developed primary using the Eclipse IDE. However, there is no reason why it should not work with other Java IDE's such as Netbeans or IntelliJ.

2.1 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 book, you will need to download the first three files (Encog Core, and Encog Examples and Encog Workbench).

. 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 book.

You should extract the Encog Core and Examples files for this first example. All of the Encog projects are built using ANT scripts. You can obtain a copy of ANT from the following URL.

http://ant.apache.org/

Encog contains an API reference in the core download. This documentation is contained in the standard Javadoc format. Instructions for installing Ant can be found at the above Web site. If you are going to use Encog with an IDE, it is not necessary to install Ant. Once you have correctly installed Ant, you should be able to issue the `ant` command from a command prompt. Figure 2.1 shows the expected output of the `ant` command.

Figure 2.1: Ant Successfully Installed

You should also extract the Encog Core, Encog Examples and Encog Workbench files into local directories. This appendix will assume that they have been extracted into the following directories:

- c:\encog-java-core-2.3.0\
- c:\encog-java-examples-2.3.0\
- c:\encog-workbench-win-2.3.0\

Now that you have installed Encog and Ant on your computer, you are ready to compile the core and examples. If you only want to use an IDE, you can skip to that section in this Appendix.

2.2 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 JAR file. It is very easy to recompile the Encog core using Ant. Open a command prompt and move to the following directory.

```
c:\encog-java-core-2.3.0\
```

From here, issue the following Ant command.

```
ant
```

This will rebuild the Encog core. If this command is successful, you should see output similar to the following:

```
C:\encog-java-core-2.3.0>ant
Buildfile: build.xml

init:

compile:

doc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source files for package org.encog...
  [javadoc] Loading source files for package
org.encog.bot...
  [javadoc] Loading source files for package
org.encog.bot.browse...
  [javadoc] Loading source files for package
org.encog.bot.browse.extract...
  [javadoc] Loading source files for package
org.encog.bot.browse.range...
  [javadoc] Loading source files for package
org.encog.bot.dataunit...
  [javadoc] Loading source files for package
org.encog.bot.rss...
  [javadoc] Loading source files for package
org.encog.matrix...
  [javadoc] Loading source files for package
org.encog.neural...
  [javadoc] Loading source files for package
org.encog.neural.activation...
...
  [javadoc] Loading source files for package
org.encog.util.math...
  [javadoc] Loading source files for package
org.encog.util.math.rbf...
  [javadoc] Loading source files for package
org.encog.util.randomize...
  [javadoc] Loading source files for package
org.encog.util.time...
  [javadoc] Constructing Javadoc information...
```

```
  [javadoc] Standard Doclet version 1.6.0_16
  [javadoc] Building tree for all the packages and
classes...
  [javadoc] Building index for all the packages and
classes...
  [javadoc] Building index for all classes...

dist:

BUILD SUCCESSFUL
Total time: 4 seconds
C:\encog-java-core-2.3.0>
```

This will result in a new Encog core JAR file being placed inside of the **lib** directory.

2.3 Compiling and Executing Encog Examples

The Encog examples are placed in a hierarchy of directories. The root example directory is located here.

```
c:\encog-java-examples-2.3.0\
```

The actual example JAR file is placed in a **lib** subdirectory off of the above directory. The examples archive that you downloaded already contains such a JAR file. It is not necessary to recompile the examples JAR file unless you make changes to one of the examples. To compile the examples, move to the root examples directory, given above.

2.3.1 Third-Party Libraries Used by Encog

Encog makes use of several third-party libraries. These third party libraries provide Encog with needed functionality. Rather than "reinvent the wheel", Encog makes use of these third-party libraries for needed functionality. That being said, Encog does try to limit the number of third-party libraries used so that installation is not terribly complex. The third-party libraries are contained in the following directory:

```
c:\encog-java-core-2.3.0\jar\
```

You will see the following JAR files there.

- hsqldb.jar
- junit-4.6.jar
- sl4j-api-1.5.6.jar
- sl4j-jdk14-1.5.6.jar

You may see different version numbers of these JARs as later versions will be released and included with Encog. These names of these JARs are listed here.

- The Hypersonic SQL Database
- JUnit
- Simple Logging Facade for Java (SLF4J)
- SLFJ Interface for JDK Logging

The Hypersonic SQL database is used internally by the Encog Unit Tests. As a result, the HSQL JAR does not need to be used when Encog is actually run. The same is true for JUnit, which is only used for unit tests.

The two SLF4J JARs are required. Encog will log certain events to make debugging and monitoring easier. Encog uses SLF4J to accomplish this. SLF4J is not an actual logging system, but rather a facade for many of the popular logging systems. This allows frameworks, such as Encog, to not need to dictate which logging API's an application using their framework should use. The SLF4J JAR must always be used with a second SLF4J JAR file which defines what actual logging API to use. Here we are using **sl4j-jdk14-1.5.6.jar**, which states that we are using the JDK logging features that are built into Java. However, by using a different interface JAR we could easily switch to other Java logging systems, such as LOG4J.

2.3.2 Running an Example from the Command Line

When you execute a Java application that makes use of Encog, the appropriate third-party JARs must be present in the Java **classpath**. The following command shows how you might want to execute the **XORResilient** example:

```
java -server -classpath .\jar\encog-core-
2.3.0.jar;.\jar\slf4j-api-1.5.6.jar;.\jar\slf4j-jdk14-
1.5.6.jar;.\lib\encog-examples-2.3.0.jar
org.encog.examples.neural.xorresilient.XORResilient
```

If the command does not work, make sure that the JAR files located in the **lib** and **jar** directories are present and named correctly. There may be new versions of these JAR files since this document was written. If this is the case, you will need to update the above command to match the correct names of the JAR files.

The examples download for Encog contains many examples. The Encog examples are each designed to be relatively short, and are usually console applications. This makes them great starting points for creating your own application to use a similar neural network technology as the example you are using. To run a different example, specify the package name and class name as was done above for **XORResilient**.

You will also notice from the above example that the **-server** option was specified. This runs the application in “Java Server Mode”. Java Server mode is very similar to the regular client mode. Programs run the same way, except in server mode it takes longer to start the program. But for this longer load time, you are rewarded with greater processing performance. Neural network applications are usually “processing intense”. As a result, it always pays to run them in “Server Mode”.

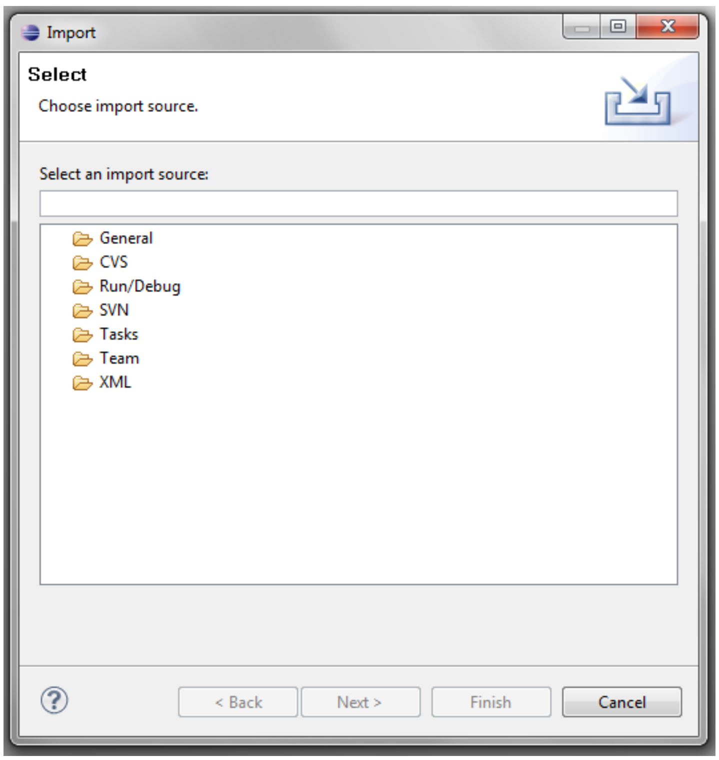
2.4 Using Encog with the Eclipse IDE

The examples can also be run from an IDE. I will show you how to use the examples with the Eclipse IDE. The examples download comes with the necessary Eclipse IDE files. As a result, you can simply import the Encog examples into Eclipse. Eclipse is not the only IDE with which Encog can be used. Encog should work fine with any Java IDE. Eclipse can be downloaded from the following URL:

http://www.eclipse.org/

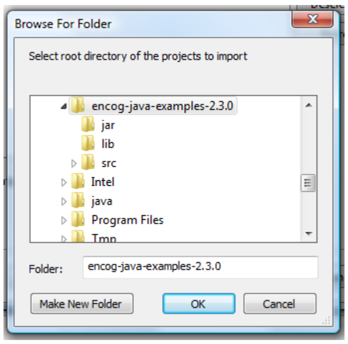
Once Eclipse has been started you should choose to import a project. To do this choose "Import" from the "File" menu. Once you have done this, you should see the Eclipse Import dialog box, as seen in Figure 2.2.

Figure 2.2: Import into Eclipse



From the "Import into Eclipse" dialog box, choose "Existing Projects into Workspace". This will allow you to choose the folder to import, as seen in Figure 2.3.

Figure 2.3: Choose Source Folder

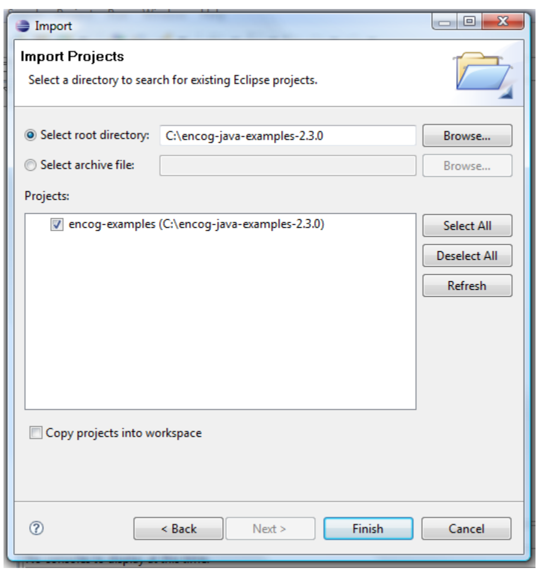


You should choose whichever directory you installed the Encog examples into, such as the following directory:

```
c:\encog-java-examples-2.3.0\
```

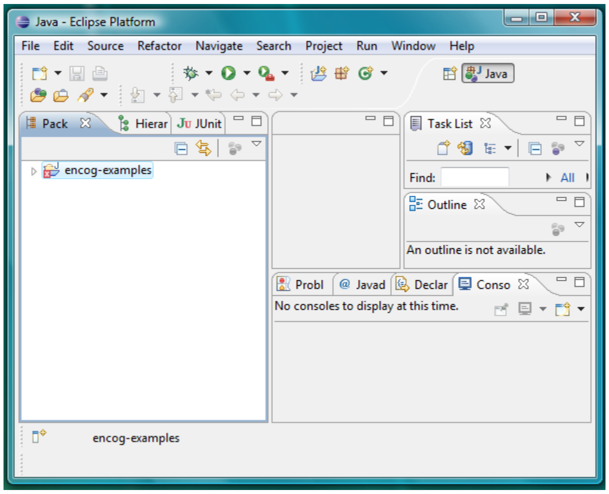
Once you have chosen your directory, you will be given a list of the projects available in that directory. There should only be one project, as shown in Figure 2.4.

Figure 2.4: Choose the Project to Import



Once the project has been imported, it will appear as a folder inside of the IDE. This folder will have a “Red X” over it if any sort of error occurs. You can see a project with an import error in Figure 2.5.

Figure 2.5: A Project with Errors

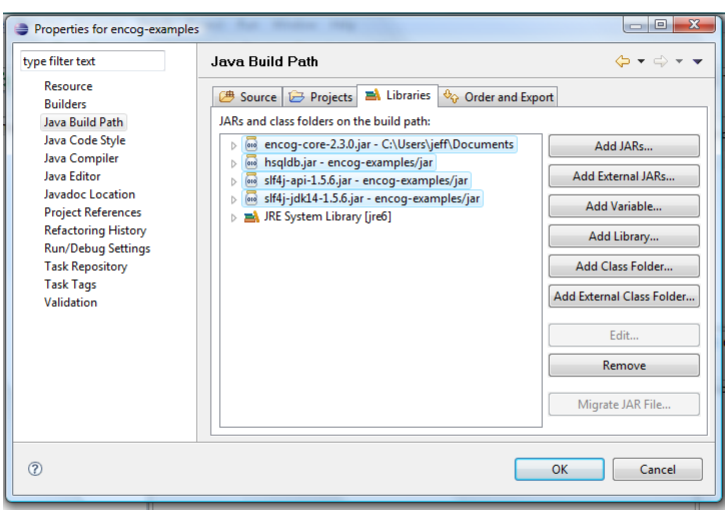


If you had any errors importing the project, the next section describes how to address them. If there were no errors, you can safely skip the next section and continue with "Executing an Example".

2.4.1 Resolving Path Errors

It is not unusual to have errors when importing the Encog examples project. This is usually because Eclipse failed to figure out the correct paths of the JAR files used by the examples. To fix this, it is necessary to remove, and then re-add the JAR files used by the examples. To do this, right click the project folder in Eclipse and choose properties. Select "Java Build Path", and you will see Figure 2.6.

Figure 2.6: The Java Build Path

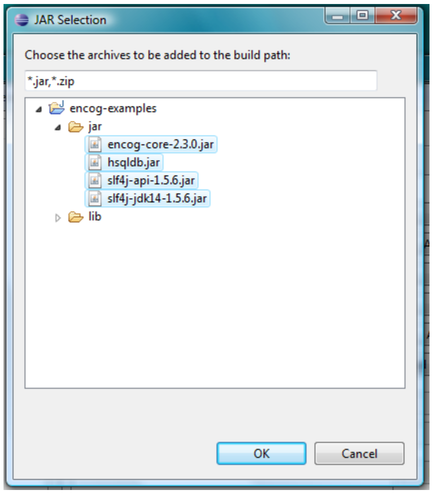


Select the four JAR files used by the examples and choose "Remove". Once the JARs are removed, you must re-add them so the examples will compile. To add the JAR files select the “Add JARs” button. This will present you with a file selection dialog box that allows you to navigate to the four required JAR files. They will be located in the following directory:

```
c:\encog-java-examples-2.3.0\jar\
```

Figure 2.7 shows the four JAR files being selected.

Figure 2.7: JAR Files

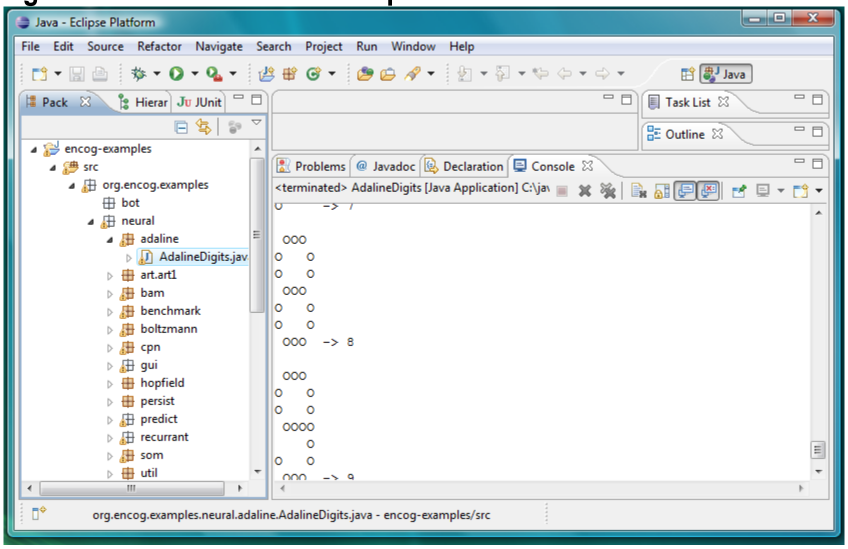


Now that the JAR files have been selected, there should be no errors remaining. We are now ready to run an example.

2.4.2 Running an Example

To run an example, simply navigate to the class that you wish to import in the IDE. Right click the class and choose “Run As” and then select “Run as Java Application”. This will run the example, and show the output in the IDE. Figure 2.8 shows the ADALINE example run from the IDE.

Figure 2.8: The ADALINE Example



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 book 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 book explains how to use neural networks with Encog and the Java programming language. The emphasis 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 Java, you may be interested in the book “Introduction to Neural Networks with Java” (ISBN: 978-1604390087).

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/

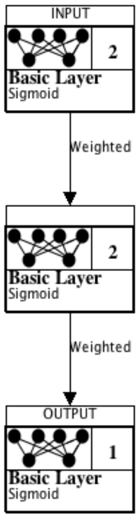
Encog is released under the Lessor GNU Public License (LGPL). All of the source code for Encog is provided in a Subversion (SVN) source code repository provided by the Google Code project. Encog is also available for the Microsoft .Net platform.

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 also 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?

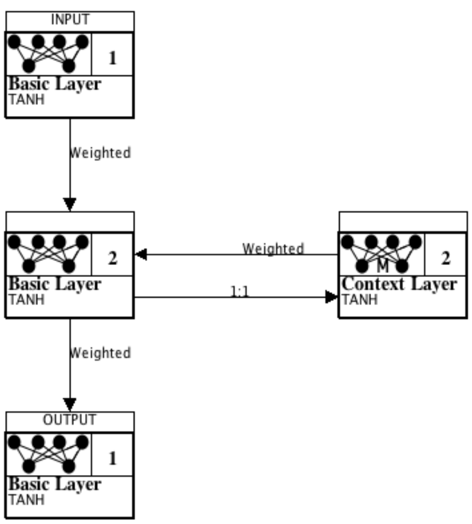
We will begin by examining what exactly a neural network is. A simple feedforward neural network can be seen in Figure 3.1. This diagram was created with the Encog Workbench. It is not just a diagram; this is an actual functioning neural network from Encog as you would actually edit it.

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

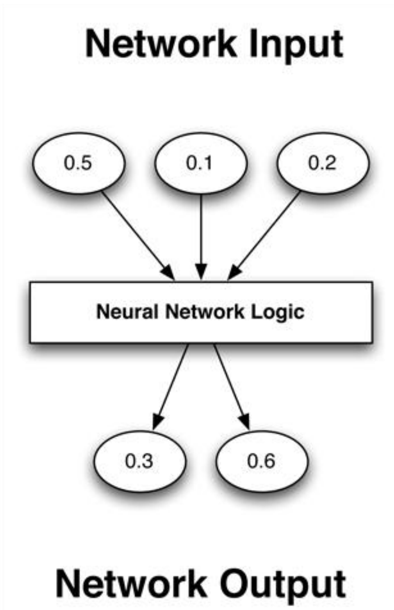


Looking at the above two neural networks you will notice that they are composed of layers, represented by the boxes. These layers are connected by lines, which represent synapses. Synapses and layers are the primary building blocks for neural networks created by Encog. The next chapter focuses solely 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 3.1 and 3.2. 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 book will focus on what happens between these two layers.

Every neural network seen in this book 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 book 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 hidden layer. Encog uses the class **NeuralData** to hold these arrays. You could easily convert the above array into a

NeuralData object with the following line of code.

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

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

The **BasicNeuralData** class simply provides a memory-based data holder for the neural network. Once the neural network processes the input, a **NeuralData** 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 **NeuralData** 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 **NeuralData** 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.

Encog supports 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.

3.1.4 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.

3.1.5 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 = 0
```

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 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 book 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:

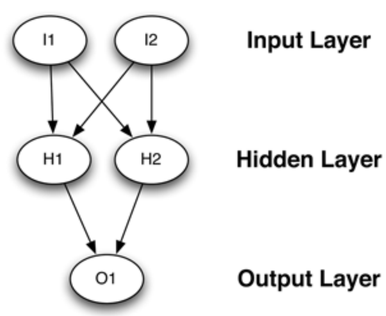
```
0.0,0.0,0.0,
0.0,1.0,1.0,
1.0,0.0,1.0,
1.0,1.0,0.0
```

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 book we will consider the first model we looked at for the XOR data.

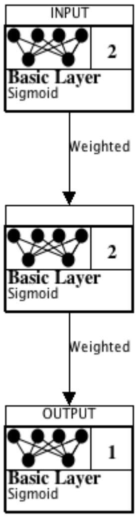
Because the XOR operator has two inputs and one output, the neural network will follow suit. Additionally, the neural network will have a single hidden layer, with two neurons to help process the data. The choice for 2 neurons in the hidden layer is arbitrary, and often comes down to trial and error. 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.getStructure().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.

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. We will learn more about weights and threshold values in a later chapter.

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

3.3 Training a Neural Network

To train the neural network, we must construct a `NeuralDataSet` 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[][] = {
{ 0.0, 0.0 },
{ 1.0, 0.0 },
{ 0.0, 1.0 },
{ 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[][] = {
{ 0.0 },
{ 1.0 },
{ 1.0 },
{ 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 a **NeuralDataSet** object must be created to hold the training set. This object is created as follows:

```
NeuralDataSet 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.

```
final Train train = new ResilientPropagation(network,
trainingSet);
```

All training classes implement the **Train** interface. The PROP 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 **Train** class until the error is below a specific value.

```
int epoch = 1;
do {
  train.iteration();
  System.out.println("Epoch #" + epoch + " Error:"
    + train.getError());
  epoch++;
} while(train.getError() > 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.

3.4 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.

```
System.out.println("Neural Network Results:");

for(NeuralDataPair pair: trainingSet ) {
  final NeuralData output =
    network.compute(pair.getInput());

  System.out.println(pair.getInput().getData(0)
    + "," + pair.getInput().getData(1)
    + ", actual=" + output.getData(0) + ",ideal=" +

  pair.getIdeal().getData(0));
}
```

The compute method accepts a **NeuralData** class and also returns a **NeuralData** 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 3.1.

Listing 3.1: Solve XOR with RPROP

```
package org.encog.examples.neural.xorresilient;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
```

```
import org.encog.neural.networks.training.Train;
import
org.encog.neural.networks.training.propagation.resilient.Re
silientPropagation;
import org.encog.util.logging.Logging;

/**
 * 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 resilient propagation (RPROP) to train the neural
network.
 * RPROP is the best general purpose supervised training
method provided by
 * Encog.
 *
 * For the XOR example with RPROP I use 4 hidden neurons.
XOR can get by on just
 * 2, but often the random numbers generated for the
weights are not enough for
 * RPROP to actually find a solution.  RPROP can have
issues on really small
 * neural networks, but 4 neurons seems to work just fine.
 */
public class XORResilient {

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

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

     public static void main(final String args[]) {

          Logging.stopConsoleLogging();

          BasicNetwork network = new BasicNetwork();
          network.addLayer(new BasicLayer(new
ActivationSigmoid(), false,2));
          network.addLayer(new BasicLayer(new
ActivationSigmoid(), false,4));
          network.addLayer(new BasicLayer(new
```

```
ActivationSigmoid(), false,1));
          network.getStructure().finalizeStructure();
          network.reset();

          NeuralDataSet trainingSet = new
BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

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


          int epoch = 1;

          do {
               train.iteration();
               System.out
                        .println("Epoch #" + epoch + "
Error:" + train.getError());
               epoch++;
          } while(train.getError() > 0.01);

          // test the neural network
          System.out.println("Neural Network Results:");
          for(NeuralDataPair pair: trainingSet ) {
               final NeuralData output =
network.compute(pair.getInput());

     System.out.println(pair.getInput().getData(0) + "," +
pair.getInput().getData(1)
                        + ", actual=" + output.getData(0)
+ ",ideal=" + pair.getIdeal().getData(0));
          }
     }
}
```

3.5 Chapter Summary

Encog is a framework that allows you to create neural networks or bot applications. This book focuses on using Encog to create neural network applications. This book focuses on the overall layout of a neural network. In this chapter, you also 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.

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 output layer. The output layer presents the data 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

- 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.

4.1 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.getStructure().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.getStructure().finalizeStructure();
network.reset();
```

The sigmoid tangent activation function is assigned to the variable **a** and passed to each of the **addLayer** calls. If no activation function is provided, Encog defaults to the hyperbolic tangent activation function.The true value, that was also introduced, specifies that the **BasicLayer** should also have threshold values.

4.1.1 The ActivationFunction Interface

All classes that are to serve as activation functions must implement the **ActivationFunction** interface. This interface is shown in Listing 4.1.

Listing 4.1: The ActivationFunction Interface

```
public interface ActivationFunction extends
EncogPersistedObject {
  void activationFunction(double[] d);
  void derivativeFunction(double[] d);
  boolean hasDerivative();
}
```

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 void activationFunction(final 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.

4.1.2 Derivatives of Activation Functions

If you would like to use propagation training with your activation function, then the activation function must have a derivative. Propagation training will be covered in greater detail in a later chapter. The derivative is calculated by a function named **derivativeFunction.**

```
public void derivativeFunction(final 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.

4.2 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.

4.2.1 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.

4.2.2 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.

```
final boolean[] winners = new boolean[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.NEGATIVE_INFINITY;
  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 is 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.

4.2.3 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 4.1.

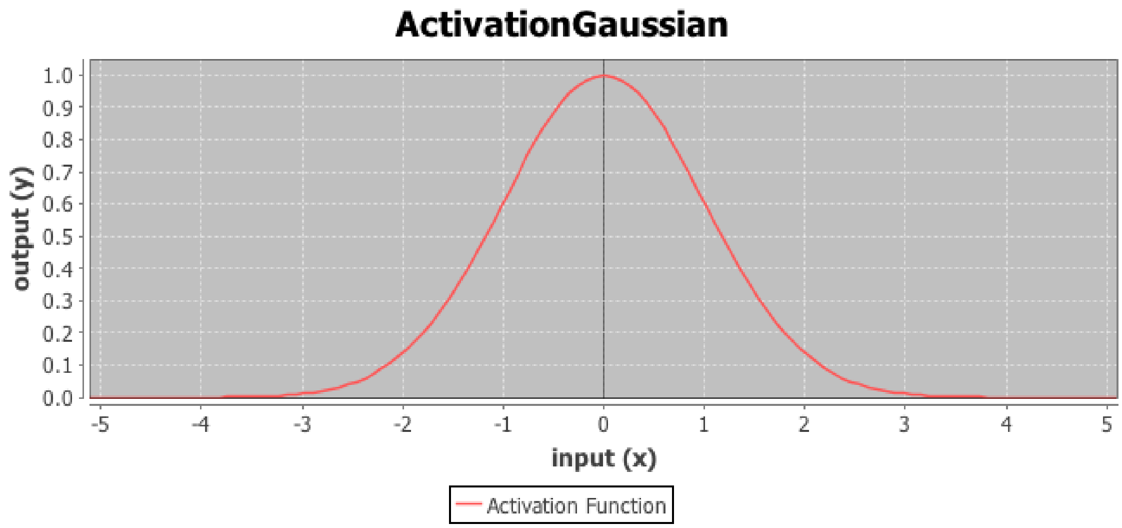
Equation 4.1: The Gaussian Function

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

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

The constant c represents the width of the curve. Figure 4.1 shows the Gaussian function.

Figure 4.1: The Graph of the Gaussian Function



The Gaussian function is implemented in Java as follows.

```
return this.peak *
Math.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.

4.3 ActivationLinear

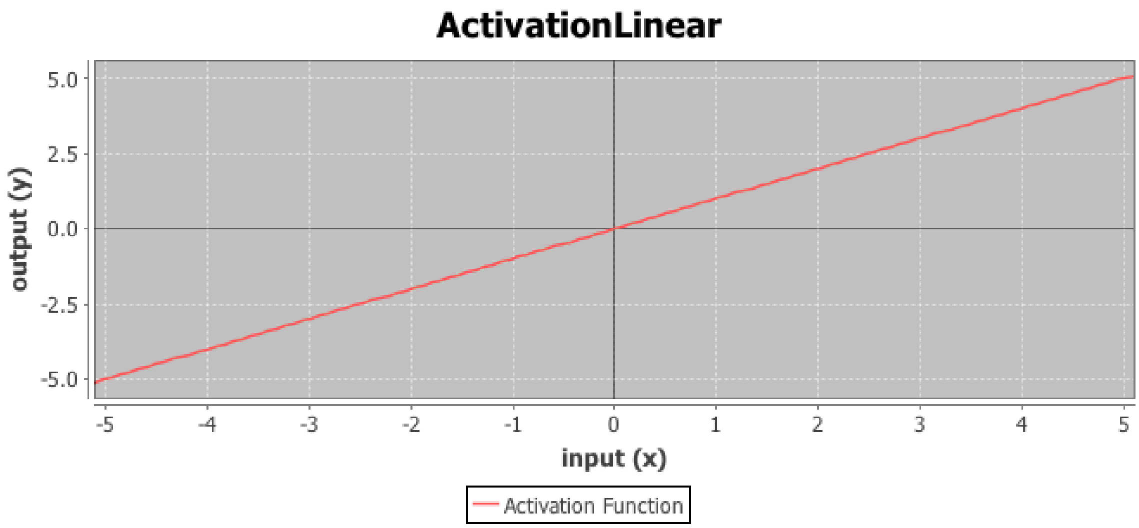
The `ActivationLinear` function is really no activation function at all. It simply implements the linear function. The linear function can be seen in Equation 4.2.

Equation 4.2: The Linear Activation Function

$$f(x) = x$$

The graph of the linear function is a simple line, as seen in Figure 4.2.

Figure 4.2: Graph of the Linear Activation Function



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

```
public void activationFunction(final 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 does not have a derivative, so it cannot be used with propagation training.

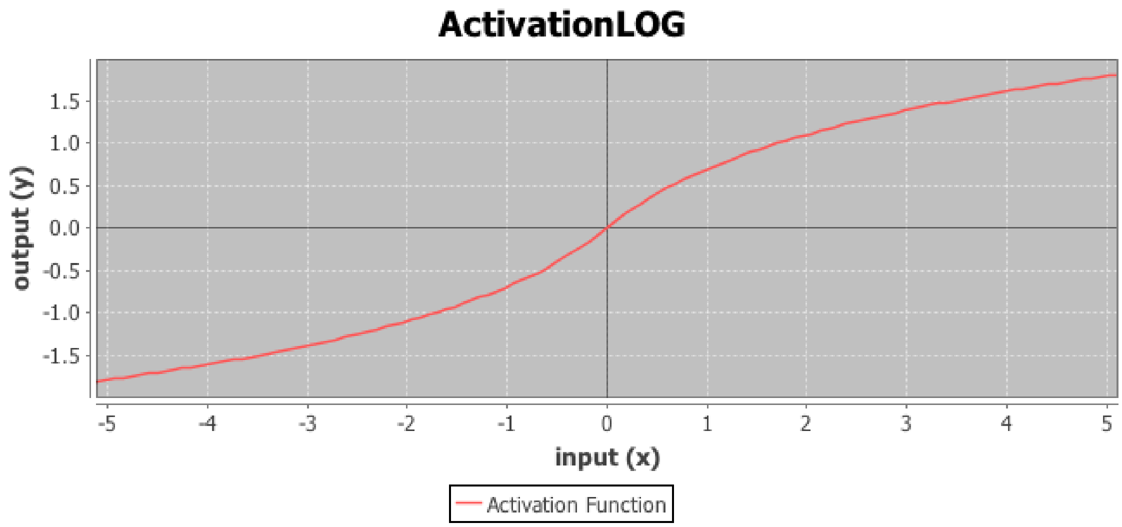
4.4 ActivationLOG

The **ActivationLog** activation function uses an algorithm based on the log function. The following Java 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 4.3.

Figure 4.3: 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 3.3, 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.

4.5 ActivationSigmoid

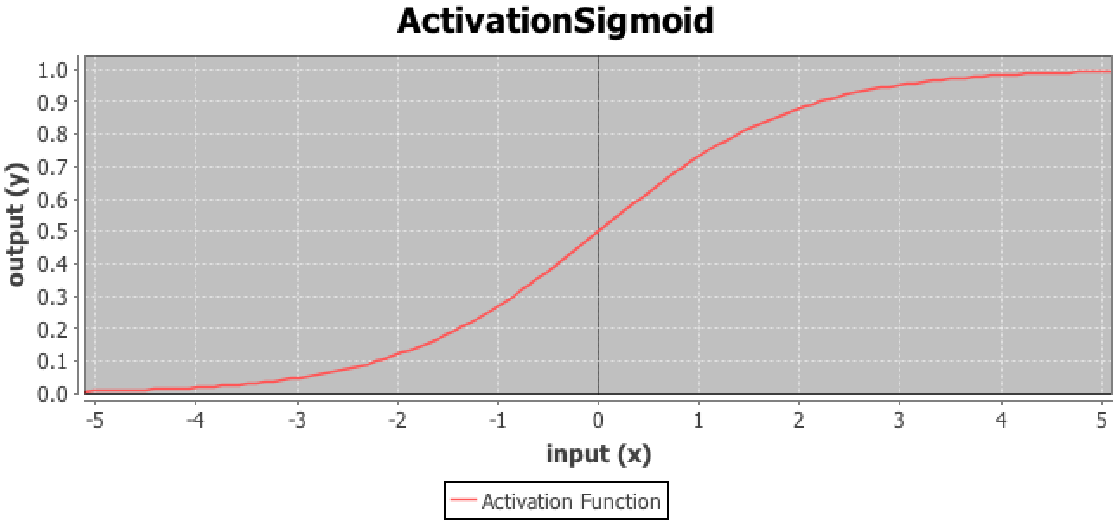
The `ActivationSigmoid` activation function should only be used when positive number output is expected. The `ActivationSigmoid` function will block negative numbers. The equation for the `ActivationSigmoid` function can be seen in Equation 4.3.

Equation 4.3: The Sigmoid Function

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

The fact that the `ActivationSigmoid` function will block negative numbers can be seen in Figure 4.4, which shows the graph of the sigmoid function.

Figure 4.4: Graph of the Sigmoid Function

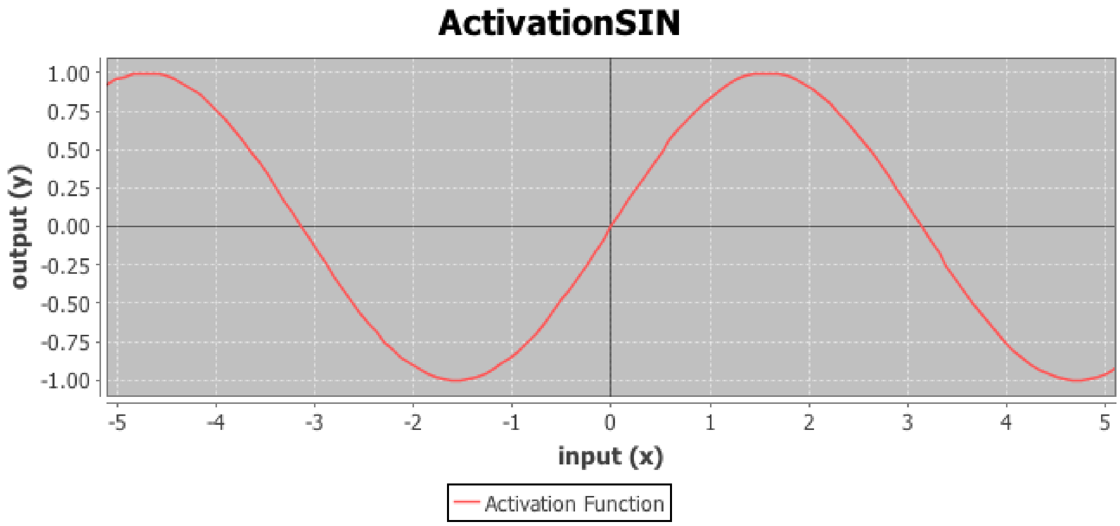


The **ActivationSigmoid** function is a very common choice for feedforward and simple recurrent neural networks. However, you must be sure that the training data does not expect negative output numbers. If negative numbers are required, consider using the hyperbolic tangent activation function.

4.6 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 4.5.

Figure 4.5: 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.

4.7 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.

4.8 ActivationTANH

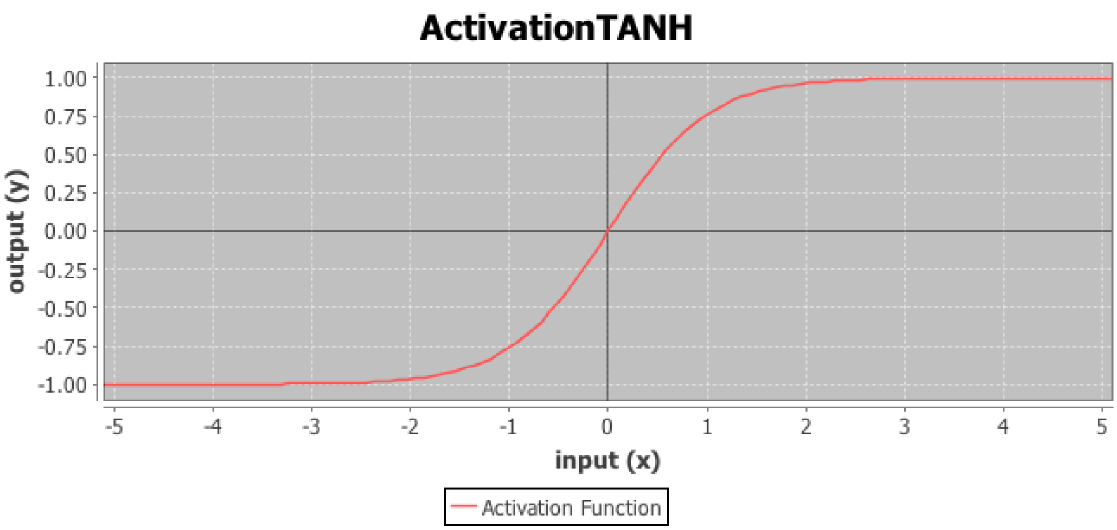
The `ActivationTANH` activation function is an activation function that uses the hyperbolic tangent function. The hyperbolic tangent activation function is probably the most commonly used activation function, as it works with both negative and positive numbers. The hyperbolic tangent function is the default activation function for Encog. The equation for the hyperbolic tangent activation function can be seen in Equation 4.4.

Equation 4.4: The Hyperbolic Tangent Activation Function

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

The fact that the hyperbolic tangent activation function accepts positive numbers can be seen in Figure 4.6, which shows the graph of the hyperbolic tangent function.

Figure 4.6: 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. Even using Java's new `Math.tanh` is still fairly slow. 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(final 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.

4.9 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 **ActivationFunction** 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 let you build neural networks composed of all of these elements.

5 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:

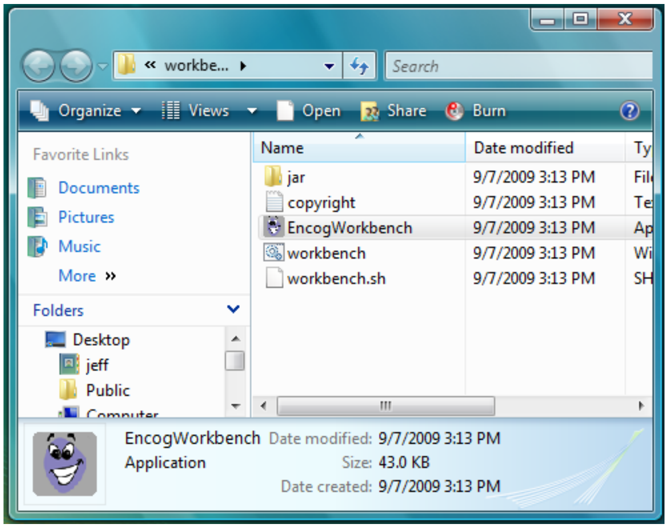
http://www.encog.org

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.

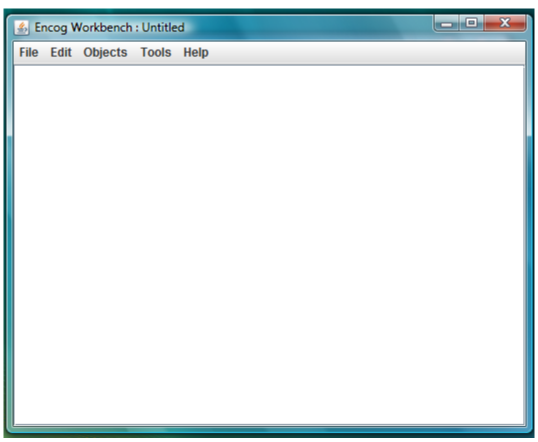
In this chapter I will assume that you are using the Windows Application package of Encog Workbench. The others will all operate very similarly. Once you download the Encog workbench and unzip it to a directory, the directory will look similar to Figure 5.1.

Figure 5.1: The Encog Workbench Folder



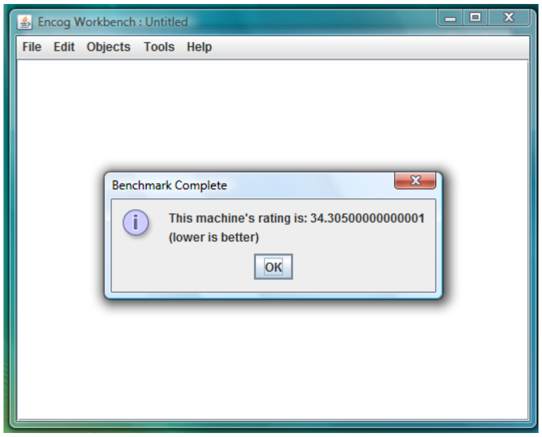
To launch the Encog workbench double click the "Encog Workbench" icon. This will launch the Encog Workbench application. Once the workbench starts, you will see something similar to what is illustrated in Figure 5.2.

Figure 5.2: 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 5.3.

Figure 5.3: Benchmarking Encog



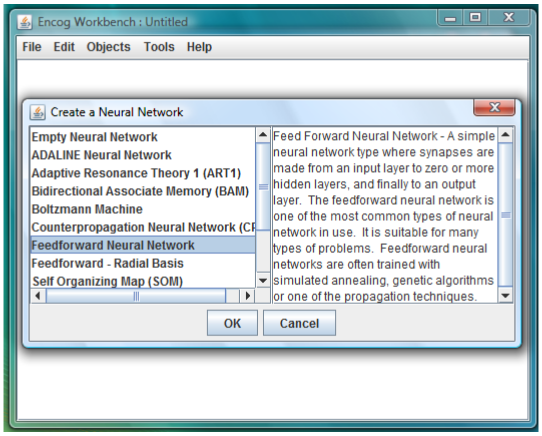
A lower number reflects a better score. The number is the amount of seconds that it took Encog to complete the benchmark tasks. Each part of the benchmark is run multiple times to try to produce consistent benchmark numbers. Encog's use of multicore processors will be reflected in this number. If the 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.

5.1 Creating a Neural Network

We will begin by creating a neural network. The Encog Workbench starts with an empty file. Once things 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 C# Encog applications. The C# and Java versions of Encog read exactly the same type of .EG files.

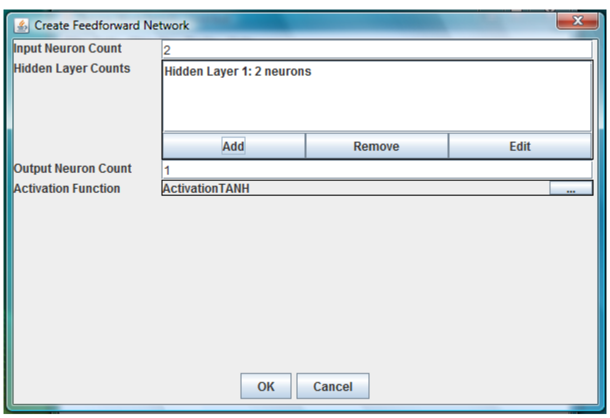
To create a neural network, select “Create Object on the “Objects menu”. A small popup window will appear that asks for the type of object to create. Choose “Neural Network” to create a new neural network. This will bring up a window that lets you browse the available types of neural networks to create. These are predefined templates for many of the common neural network types supported by Encog. This window can be seen in Figure 5.4.

Figure 5.4: 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 5.5.

Figure 5.5: 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
- Output Neuron Count: 2
- Hidden Layer 1 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.

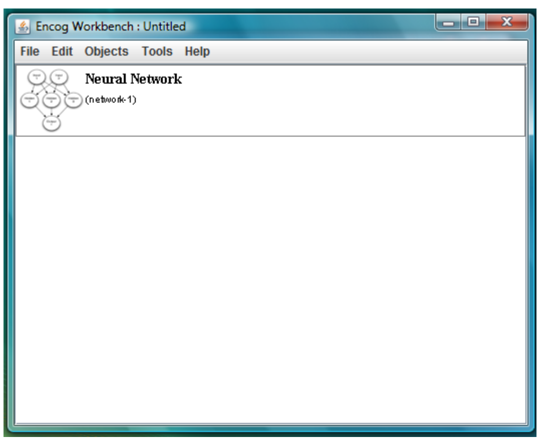
```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

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. Encog can help with this trial and error process

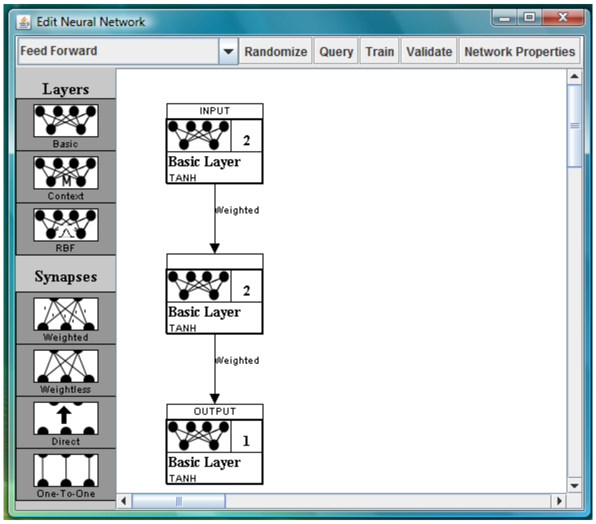
Now that the feedforward neural network has been created, you will see it in the workbench. Figure 5.6 shows the workbench with a neural network added.

Figure 5.6: Neural Network Added



If you double click the feedforward neural network shown in Figure 5.6, it will open. This allows you to see the layers and synapses. Figure 5.7 shows the feedforward neural network that was just created.

Figure 5.7: 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.

5.2 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.

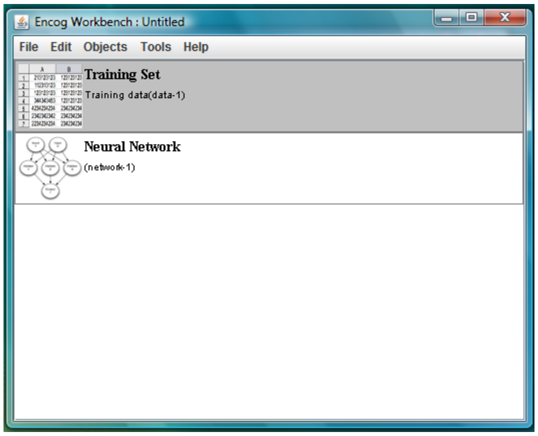
- Supervised Training
- Unsupervised Training

Supervised training data has both an input and expected output specified for the neural network. For example, the 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.

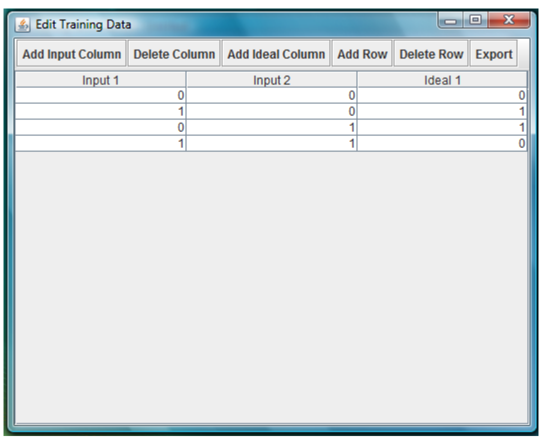
First, we must create a training set. Select “Create Object” from the “Objects” menu. Select a training set. Once the training set has been created it will be added along with the network that was previously created. Figure 5.8 shows the newly created training set.

Figure 5.8: 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 5.9.

Figure 5.9: Editing the Training Set



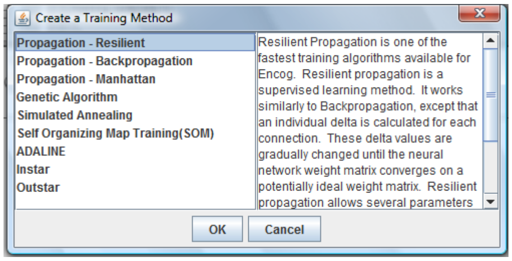
Here you can see the training set. By default, Encog creates a training set for XOR. This is just the default. Usually you would now create the desired number of input and output columns. However, because we are training the XOR operator, the data is fine as it is.

5.3 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.

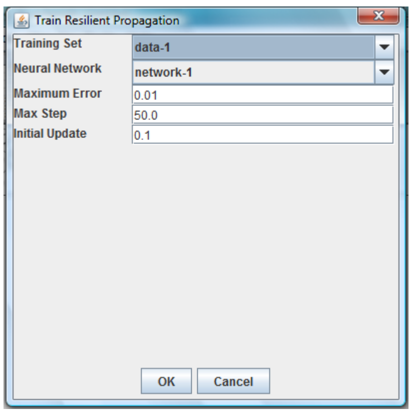
To train the neural network open it as you did for Figure 5.7. Click the “Train” button at the top of the window. This will display a dialog box that allows you to choose a training method, as seen in Figure 5.10.

Figure 5.10: Choosing a Training Method



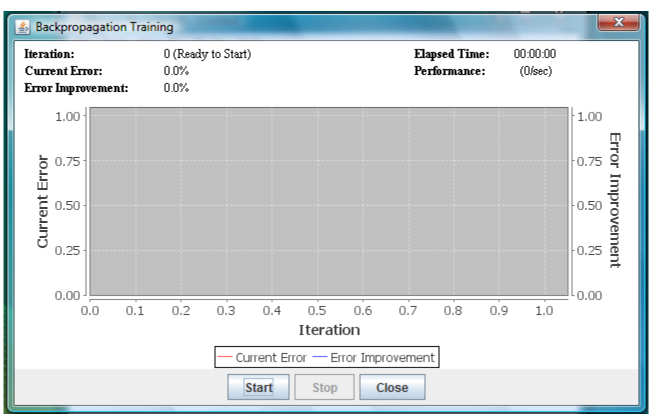
Choose the resilient training method, under propagation. This is usually the best training method available for a supervised feedforward neural network. There are several parameters you can set for the resilient training method. For resilient training it is very unlikely that you should ever change any of these options, other than perhaps the desired maximum error, which defaults to 1%. You can see this dialog box in Figure 5.11.

Figure 5.11: Resilient Propagation Training



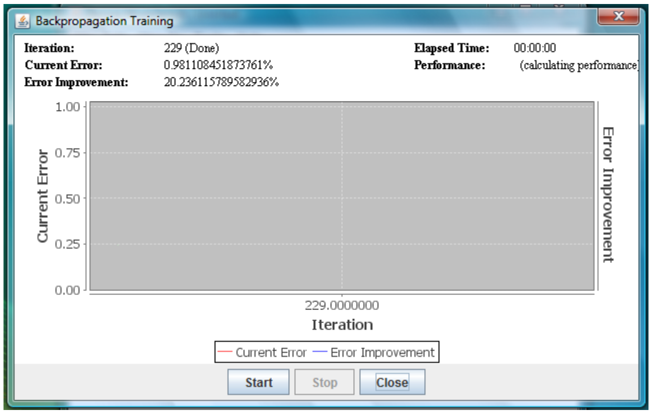
Selecting okay will open a window that will allow you to monitor the training progress, as seen in Figure 5.12.

Figure 5.12: 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 5.13: Training Complete

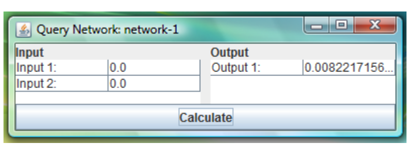


The training is complete because the current error fell below the maximum error allowed that was entered in Figure 5.11, which is 1%. Now that the network has been trained it can produce meaningful output when queried.

5.4 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 5.7. This will open the query window as seen in Figure 5.14.

Figure 5.14: Query the Neural Network

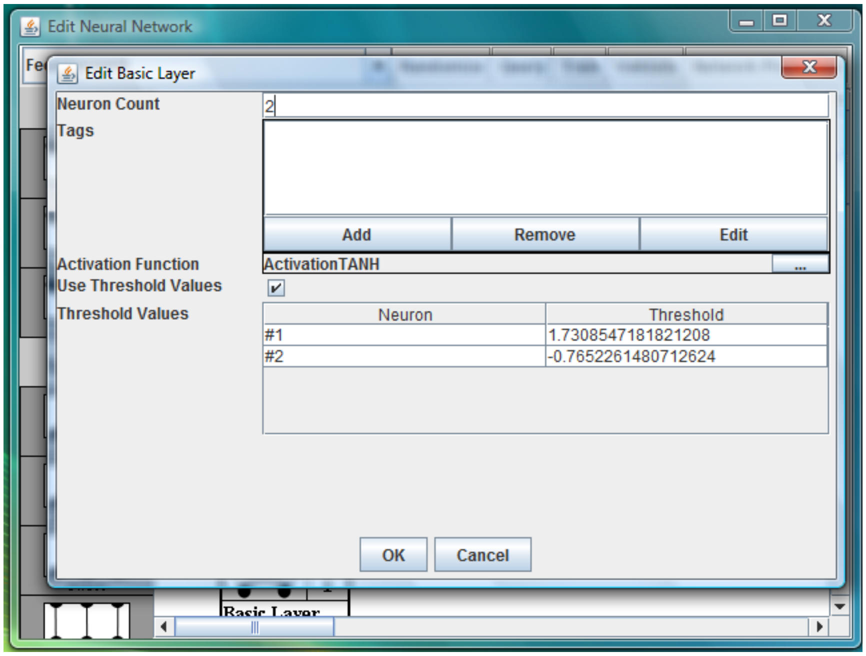


As you can see from the above window, you are allowed to enter two values for the input neurons. When you click "Calculate", the output values will be shown. In the example above two zeros were entered, which resulted in 0.008. This is consistent with the XOR operator, as 0.008 is close to zero. To get 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 5.7, 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

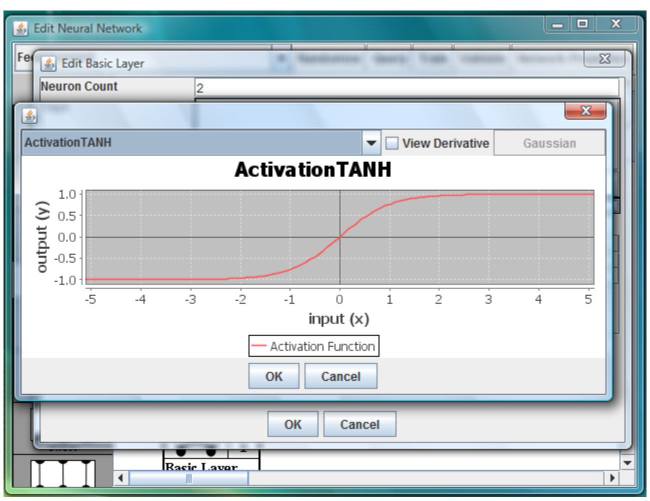
5.15 shows the dialog used to edit the layer properties.

Figure 5.15: 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 5.16.

Figure 5.16: Edit the Activation Function



In Figure 5.16 you can see that the current activation function is the hyperbolic tangent. The graph for the hyperbolic tangent function is also shown for reference.

5.5 Generating Code

The Encog workbench provides two ways that you can make use of your neural network in Java code. First, you can save the neural network and training data to an .EG file. Java 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. For the generated program to be of any use, you will need to add your own training code. Listing 5.1 shows the generated Java code from the XOR, feedforward neural network.

Listing 5.1: Generated Java Code

```
import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.layers.Layer;

/**
 * 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
 *
 */

public class EncogGeneratedClass {

  public static void main(final String args[]) {

    BasicNetwork network = new BasicNetwork();

    Layer inputLayer = new BasicLayer( new
ActivationSigmoid(),true,2);
    inputLayer.addNext(inputLayer);

    Layer hiddenLayer1 = new BasicLayer( new
ActivationSigmoid(),true,2);
    inputLayer.addNext(hiddenLayer1);

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

```
    hiddenLayer1.addNext(outputLayer);
    network.tagLayer("INPUT",inputLayer);
    network.tagLayer("OUTPUT",outputLayer);
    network.getStructure().finalizeStructure();
    network.reset();
  }
}
```

5.6 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.

6 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 book 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. Chapter 8 will discuss unsupervised training.

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.

6.1 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. If online training is desired, the weights will be updated for each training element. If batch training is desired, the weights will not be updated until a certain number of training elements have been processed, or the end of the iteration has occurred. The iteration ends when there are no more training elements.

By default, Encog always uses a batch size equal to the number of training elements.

Therefore, Encog updates the weight matrix and threshold values at the end of an iteration by default. Using a smaller batch size will decrease the efficiency of training but will give a more accurate training result that may converge to a lower error rate quicker.

We will now examine what happens in 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. How exactly this value is used depends on the training algorithm used.

6.1.1 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.

6.1.2 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 magnitude of the gradient. 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.

6.1.3 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 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.

6.2 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 Java 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.

6.2.1 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 Java program. Listing 6.1 shows a simple Java program that will train a neural network to recognize the XOR operator.

Listing 6.1: Using Backpropagation

```
package org.encog.examples.neural.xorbackprop;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.logic.FeedforwardLogic;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.
  back.Backpropagation;
import org.encog.util.logging.Logging;

public class XorBackprop {

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

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

     public static void main(final String args[]) {

          Logging.stopConsoleLogging();

          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.setLogic(new FeedforwardLogic());
          network.getStructure().finalizeStructure();
          network.reset();

          NeuralDataSet trainingSet = new
BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

          // train the neural network
          final Train train = new Backpropagation(network,
trainingSet, 0.7, 0.8);

          int epoch = 1;

          do {
               train.iteration();
               System.out
                         .println("Epoch #" + epoch + "
Error:" + train.getError());
               epoch++;
          } while(train.getError() > 0.01);

          // test the neural network
          System.out.println("Neural Network Results:");
          for(NeuralDataPair pair: trainingSet ) {
               final NeuralData output =
network.compute(pair.getInput());

     System.out.println(pair.getInput().getData(0) + "," +
pair.getInput().getData(1)
                         + ", actual=" + output.getData(0)
+ ",ideal=" + pair.getIdeal().getData(0));
          }
     }
}
```

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

6.2.2 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[][] = {
  { 0.0, 0.0 },
  { 1.0, 0.0 },
  { 0.0, 1.0 },
  { 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[][] = {
  { 0.0 },
  { 1.0 },
  { 1.0 },
  { 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.

6.2.3 Constructing the Neural Network

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 Network 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));
```

The **FeedforwardLogic** class is used to provide the logic for this neural network. The default logic type of **SimpleRecurrentLogic** would have also worked, but **FeedforwardLogic** will provide better performance because there are no recurrent connections in this network.

```
network.setLogic(new FeedforwardLogic());
```

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.getStructure().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.

6.2.4 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 **NeuralDataSet**. The following code does this.

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

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

6.2.5 Training the Neural Network

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

```
final Train 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 process. 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();

  System.out.println("Epoch #"
  + epoch + " Error:" + train.getError());

  epoch++;
} while (train.getError() > 0.01);
```

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.

6.2.6 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:.

```
System.out.println("Neural Network Results:");
```

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

```
for(NeuralDataPair pair: trainingSet ) {
```

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

```
  final NeuralData output =
network.compute(pair.getInput());
```

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

```
  System.out.println(pair.getInput().getData(0) + "," +
pair.getInput().getData(1)
  + ", actual=" + output.getData(0) + ",ideal=" +
```

```
pair.getIdeal().getData(0));
}
```

The output from this neural network is shown here.

```
Epoch #1 Error:0.504998283847474
Epoch #2 Error:0.504948046227928
Epoch #3 Error:0.502896861682661З
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 3345, 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 3345 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 3345 is a fairly good indication of the efficiency of the backpropagation algorithm. This number will be compared to the other propagation training algorithms.

6.3 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 6.2 shows the complete Manhattan update rule example.

Listing 6.2: Using the Manhattan Update Rule

```
package org.encog.examples.neural.xormanhattan;

import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import
org.encog.neural.networks.training.propagation.manhattan.Ma
nhattanPropagation;
import org.encog.util.logging.Logging;

/**
 * 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 the Manhattan
 * update rule to train the neural network.
 *
 * @author $Author$
 * @version $Revision$
 */
public class XORManhattan {

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

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

```
}, { 1.0 }, { 0.0 } };

     public static void main(final String args[]) {

          Logging.stopConsoleLogging();

          final BasicNetwork network = new BasicNetwork();
          network.addLayer(new BasicLayer(2));
          network.addLayer(new BasicLayer(3));
          network.addLayer(new BasicLayer(1));
          network.getStructure().finalizeStructure();
          network.reset();

          final NeuralDataSet trainingSet = new
BasicNeuralDataSet(
                    XORManhattan.XOR_INPUT,
XORManhattan.XOR_IDEAL);

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

          int epoch = 1;

          do {
               train.iteration();
               System.out
                         .println("Epoch #" + epoch + "
Error:" + train.getError());
               epoch++;
          } while (train.getError() > 0.01);

          // test the neural network
          System.out.println("Neural Network Results:");
          for (final NeuralDataPair pair : trainingSet) {
               final NeuralData output =
network.compute(pair.getInput());

     System.out.println(pair.getInput().getData(0) + ","
                         + pair.getInput().getData(1) + ",
actual="
                         + output.getData(0) + ",ideal="
                         + pair.getIdeal().getData(0));
          }
     }
```

```
}
```

There is really only one line that has changed from the backpropagation example. Because the **ManhattanPropagation** object uses the same **Train** 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.

```
final Train 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.

6.4 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 6.3 shows an XOR example using the resilient propagation algorithm.

Listing 6.3: Using Resilient Propagation

```
package org.encog.examples.neural.xorresilient;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import
org.encog.neural.networks.training.propagation.resilient.Re
silientPropagation;
import org.encog.util.logging.Logging;

/**
 * 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 resilient propagation (RPROP) to train the neural
network.
 * RPROP is the best general purpose supervised training
method provided by
 * Encog.
 *
 * For the XOR example with RPROP I use 4 hidden neurons.
XOR can get by on just
 * 2, but often the random numbers generated for the
weights are not enough for
 * RPROP to actually find a solution.  RPROP can have
issues on really small
 * neural networks, but 4 neurons seems to work just fine.
```

```
 */
public class XORResilient {

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

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

     public static void main(final String args[]) {

          Logging.stopConsoleLogging();

          BasicNetwork network = new BasicNetwork();
          network.addLayer(new BasicLayer(new
ActivationSigmoid(), false,2));
          network.addLayer(new BasicLayer(new
ActivationSigmoid(), false,4));
          network.addLayer(new BasicLayer(new
ActivationSigmoid(), false,1));
          network.getStructure().finalizeStructure();
          network.reset();

          NeuralDataSet trainingSet = new
BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

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


          int epoch = 1;

          do {
               train.iteration();
               System.out
                         .println("Epoch #" + epoch + "
Error:" + train.getError());
               epoch++;
          } while(train.getError() > 0.01);

          // test the neural network
          System.out.println("Neural Network Results:");
          for(NeuralDataPair pair: trainingSet ) {
               final NeuralData output =
network.compute(pair.getInput());
```

```
     System.out.println(pair.getInput().getData(0) + "," +
pair.getInput().getData(1)
                         + ", actual=" + output.getData(0)
+ ",ideal=" + pair.getIdeal().getData(0));
          }
     }
}
```

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

```
final Train 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=0.004268459525397450,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.

6.5 Propagation and Multithreading

As of the writing of this book, it is nearly impossible to buy a single core computer, as dual core computers become the norm. 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. There is a fourth type of propagation training supported by Encog that is particularly adept at multithreaded training. This training type is called Multipropagation (MPROP). MPROP is based resilient propagation (RPROP).

6.5.1 How Multipropagation Works

Multipropagation (MPROP) works particularly well with larger training sets and multiple cores. If MPROP does not detect that both are present, it will fall back to RPROP. When there is more than one processing core, and enough training set items to keep both cores busy, MPROP will function significantly faster than RPROP.

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.

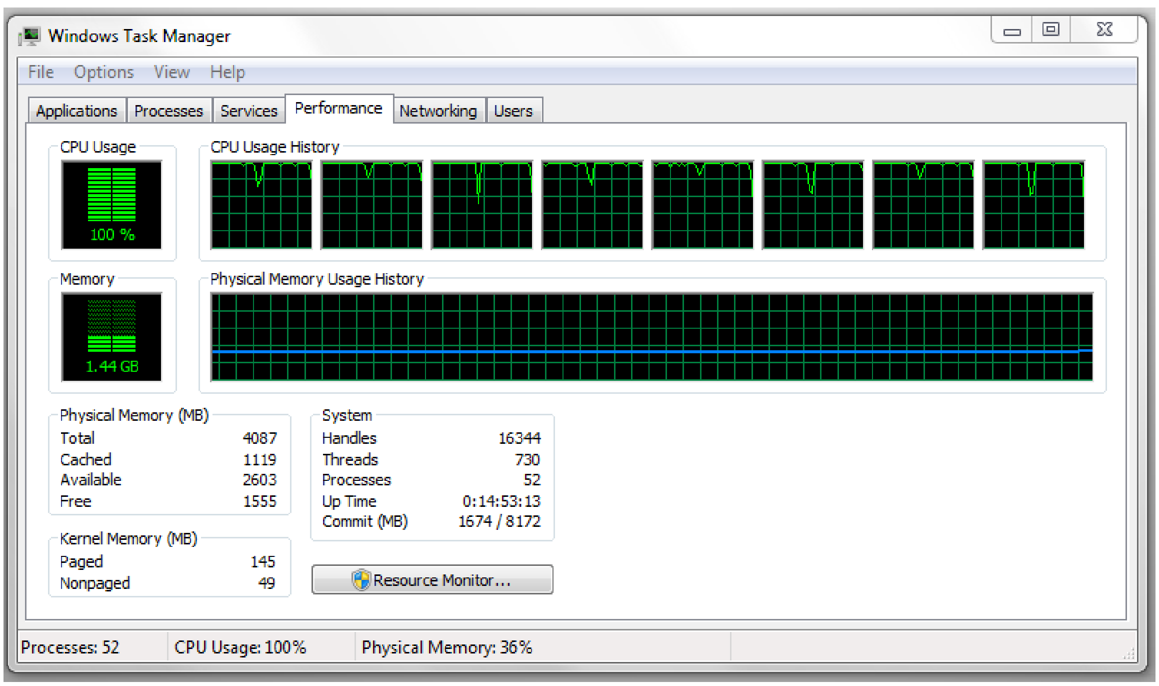
The third step is what varies among the different training algorithms. Backpropagation simply takes the gradient descents and scales them by a learning rate. The scaled gradient descents are then directly applied to the weights and thresholds. The Manhattan Update Rule only uses the sign of the gradient 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 MPROP 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 6.1.

Figure 6.1: 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 RPROP algorithm, upon which MPROP is based, needs all changes applied before the next iteration begins.

6.5.2 Using Multipropagation

Just as was the case with the other propagation training algorithms, the MPROP algorithm implements the `Train` interface. The following line of code creates a

MultiPropagation object that will be used to train the neural network.

```
final Train train = new MultiPropagation(network,
trainingSet);
```

The above line will create a MPROP trainer with a number of threads determined by Encog. The number of threads takes into account the size of the training data and the number of cores available. The optimal number of threads is determined to be one, and then MPROP will fall back to using RPROP. Because MPROP can decide on the optimal number of threads, as well as if it is even the most optimal training algorithm, MPROP is considered to be the best general purpose supervised training algorithm currently offered by Encog.

It is also possible to force MPROP to use a specific number of threads. The following line of code forces MPROP to use four threads:

```
final Train train = new MultiPropagation(network,
trainingSet, 4);
```

The iteration loop remains the same as the previous three examples. We could substitute MPROP for the XOR examples previously used. However, MPROP works best with training sets of at least a few thousand elements. If you let Encog pick the number of threads for XOR, it will fall back to RPROP. If you force Encog to use four threads, it will train the XOR problem, but PROP would have been more efficient.

To see PROP 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 RPROP and MPROP 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 6.4.

Listing 6.4: Using Multipropagation

```
package org.encog.examples.neural.benchmark;

import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import
org.encog.neural.networks.training.propagation.multi.MultiP
ropagation;
import
org.encog.neural.networks.training.propagation.resilient.Re
silientPropagation;
import org.encog.util.benchmark.RandomTrainingFactory;
import org.encog.util.logging.Logging;
```

```
public class MultiBench {

     public static final int INPUT_COUNT = 40;
     public static final int HIDDEN_COUNT = 60;
     public static final int OUTPUT_COUNT = 20;

     public static BasicNetwork generateNetwork()
     {
          final BasicNetwork network = new BasicNetwork();
          network.addLayer(new
BasicLayer(MultiBench.INPUT_COUNT));
          network.addLayer(new
BasicLayer(MultiBench.HIDDEN_COUNT));
          network.addLayer(new
BasicLayer(MultiBench.OUTPUT_COUNT));
          network.getStructure().finalizeStructure();
          network.reset();
          return network;
     }

     public static NeuralDataSet generateTraining()
     {
          final NeuralDataSet training =
RandomTrainingFactory.generate(50000,
                    INPUT_COUNT, OUTPUT_COUNT, -1, 1);
          return training;
     }

     public static double evaluateRPROP(BasicNetwork
network,NeuralDataSet data)
     {

          ResilientPropagation train = new
ResilientPropagation(network,data);
          long start = System.currentTimeMillis();
          System.out.println("Training 20 Iterations with
RPROP");
          for(int i=1;i<=20;i++)
          {
               train.iteration();
               System.out.println("Iteration #" + i + "
Error:" + train.getError());
          }
          train.finishTraining();
          long stop = System.currentTimeMillis();
          double diff = ((double)(stop - start))/1000.0;
```

```
			System.out.println("RPROP Result:" + diff + " 
seconds." );
			System.out.println("Final RPROP error: " + 
network.calculateError(data));
			return diff;
	}

	public static double evaluateMPROP(BasicNetwork 
network,NeuralDataSet data)
	{

			MultiPropagation train = new 
MultiPropagation(network,data);
			long start = System.currentTimeMillis();
			System.out.println("Training 20 Iterations with 
MPROP");
			for(int i=1;i<=20;i++)
			{
				train.iteration();
				System.out.println("Iteration #" + i + " 
Error:" + train.getError());
			}
			train.finishTraining();
			long stop = System.currentTimeMillis();
			double diff = ((double)(stop - start))/1000.0;
			System.out.println("MPROP Result:" + diff + " 
seconds." );
			System.out.println("Final MPROP error: " + 
network.calculateError(data));
			return diff;
	}

	public static void main(String args[])
	{
			Logging.stopConsoleLogging();
			BasicNetwork network = generateNetwork();
			NeuralDataSet data = generateTraining();

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

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

```
Training 20 Iterations with RPROP
Iteration #1 Error:1.0592062021803321
Iteration #2 Error:1.0112968157018771
Iteration #3 Error:0.9650583848127503
Iteration #4 Error:0.9269433225981621
Iteration #5 Error:0.8947162095367102
Iteration #6 Error:0.8714873694194031
Iteration #7 Error:0.8445288449926142
Iteration #8 Error:0.8186688302191717
Iteration #9 Error:0.7952278955734976
Iteration #10 Error:0.7717422560410586
Iteration #11 Error:0.7475048877257578
Iteration #12 Error:0.7235382011165326
Iteration #13 Error:0.7026047081990957
Iteration #14 Error:0.6843757761100023
Iteration #15 Error:0.6685206160475999
Iteration #16 Error:0.6539311876046258
Iteration #17 Error:0.6412660225209257
Iteration #18 Error:0.630790400329957
Iteration #19 Error:0.6211146795350724
Iteration #20 Error:0.6136882493691617
RPROP Result:128.562 seconds.
Final RPROP error: 0.6075224766406004
Training 20 Iterations with MPROP
Iteration #1 Error:0.6075212244066446
Iteration #2 Error:0.8665463281875874
Iteration #3 Error:0.8316846996192032
Iteration #4 Error:0.7451195340393163
Iteration #5 Error:0.7005024644028119
Iteration #6 Error:0.6691870245157884
Iteration #7 Error:0.649034289358449
Iteration #8 Error:0.6339114535879514
Iteration #9 Error:0.6208812103003265
Iteration #10 Error:0.6111566730037973
Iteration #11 Error:0.6056166450414902
Iteration #12 Error:0.6003765685919015
Iteration #13 Error:0.5964873091129251
Iteration #14 Error:0.5932816072550446
Iteration #15 Error:0.5905725872184455
Iteration #16 Error:0.5882703219084173
Iteration #17 Error:0.5863667500894574
Iteration #18 Error:0.5848003831853418
Iteration #19 Error:0.5835759158206529
Iteration #20 Error:0.5823759906353797
```

```
MPROP Result:31.88 seconds.
Final MPROP error: 0.5814082684159508
Factor improvement:4.032685069008783
```

As you can see from the above results, the RPROP algorithm finished in 128 seconds, the MPROP algorithm finished in only 31 seconds. MPROP 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 MPROP training will fall back to RPROP as well.

6.6 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.

Multipropagation (MPROP) is a training technique that adapts RPROP to perform faster with multicore computers. Given a computer with multiple cores and a large enough training set, MPROP is considerably faster than RPROP. If these conditions are not present, MPROP will fall back to RPROP. Because of this MPROP is the best general purpose supervised training technique currently offered by Encog.

This completes the “Introduction to Encog” ebook. For more information about Encog refer to the JavaDocs. We are also working on additional Encog-centered books to publish soon.