Gettint Started with Encog3 in Java

Gettint Started with Encog3 in Java

Jeff Heaton

Heaton Research, Inc.
St. Louis, MO, USA

Title	Gettint Started with Encog3 in Java
Author	Jeff Heaton
Published	September 20, 2011
Copyright	Copyright 2011 by Heaton Research, Inc., All Rights Reserved.
ISBN	978-1-60439-031-5
Price	FREE
File Created	Tue Sep 20 06:31:06 CDT 2011

Publisher: Heaton Research, Inc
Getting Started with Encog 3 in Java
September, 2011
Author: Jeff Heaton
Editor: WordsRU.com
Cover Art: Carrie Spear
ISBN's for all Editions:
978-1-60439-031-5, PDF/AZW

SOFTWARE LICENSE AGREEMENT: TERMS AND CONDITIONS

SOFTWARE SUPPORT

WARRANTY

DISCLAIMER

SHAREWARE DISTRIBUTION

Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

Contents

Chapter 1

Installing and Using Encog

- Downloading Encog
- Running Examples
- Running the Workbench

This guide is intended to get you started with Encog. It is distributed as a free ebook with the Encog framework. In addition this guide I also sell additional, more in-depth, books on Encog, machine learning and neural networks. You can find more information about these books from the following URL. These books can be purchased in paperback or ebook form.

`http://www.heatonresearch.com/book`

This book is a shortened version of the full book "Programming Neural Networks with Encog3 in Java, 2nd Edition". If you would like to buy the full ebook(or paperback) visit the following URL.

`http://www.heatonresearch.com/book/programming-neural-networks-encog3-java.html`

This chapter shows how to install and use Encog. This consists of downloading Encog from the Encog Web site, installing and then running the examples. You will also be shown how to run the Encog Workbench. Encog makes use of Java. This chapter assumes that you have already downloaded and installed Java JSE version 6 or later on your computer. The latest version of Java can be downloaded from the following Web site:

`http://java.sun.com/`

The examples in this book were tested with JSE 6 and 7. 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 chapter 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.

You can also find useful information on setting up Encog from the following web address.

`http://www.heatonresearch.com/wiki/Encog_Examples`

1.1 Installing Encog

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

`http://www.heatonresearch.com/encog/`

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

For this book, you will need to download the examples and the workbench. The workbench is distributed as a universal executable JAR. You should be able to simply double-click this JAR file on either Windows, Linux or Mac. This JAR file, and a few information files, are all that is included with the Encog workbench release.

Encog includes both Ant and Maven files to assist you with building the examples. Additionally, you can use an IDE. For information in using Encog with an IDE refer to the following page. There are several IDE tutorials here.

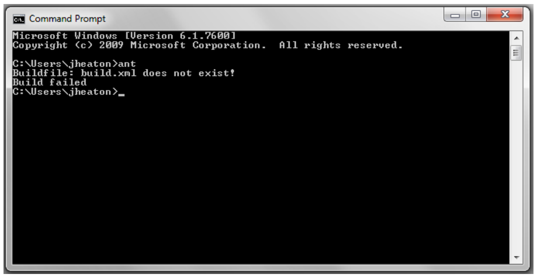
`http://www.heatonresearch.com/wiki/Getting_Started`

Apache Ant can be obtained 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 1.1 shows the expected output of the **ant** command.

Figure 1.1: Ant Successfully Installed



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\jheaton>ant
Buildfile: build.xml does not exist!
Build failed
C:\Users\jheaton>_
```

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

- c:\encog-java-core-3.0.0\
- c:\encog-java-examples-3.0.0\
- c:\encog-workbench-win-3.0.0\

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 chapter.

1.2 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-3.0.0\
```

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-3.0.0>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-3.0.0>
```

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

1.3 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-3.0.0\
```

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.

1.3.1 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 **XORHelloWorld** example:

```
java -cp ./lib/encog-core-3.0.0.jar;./lib/examples.jar org.encog.examples.
    neural.xor.XORHelloWorld
```

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 **XORHelloWorld**.

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".

Chapter 2

The Encog Workbench

- Structure of the Encog Workbench
- A Simple XOR Example
- Using the Encog Analyst
- Encog Analyst Reports

The Encog Workbench is a GUI application that enables many different machine learning tasks without writing Java or C# code. The Encog Workbench itself is written in Java, but generates files that can be used with any Encog framework.

The Encog Workbench is distributed as a single self-executing JAR file. On most operating systems, the Encog Workbench JAR file is started simply by double-clicking. This includes Microsoft Windows, Macintosh and some variants of Linux. To start from the command line, the following command is used.

```
java Űjar ./encog−workbench−3.0.0−executable
```

Depending on the version of Encog, the above JAR file might have a different name. No matter the version, the file will have "encog-workbench" and "executable" somewhere in its name. No other JAR files are necessary for the workbench as all third party JAR files were are placed inside this JAR.

2.1 Structure of the Encog Workbench

Before studying how the Encog Workbench is actually used, we will learn about its structure. The workbench works with a project directory that holds all of the files needed for a project. The Encog Workbench project contains no subdirectories. Also, if a subdirectory is added into an Encog Workbench project, it simply becomes another independent project.

There is also no main "project file" inside an Encog Workbench project. Often a readme.txt or readme.html file is placed inside of an Encog Workbench project to explain what to do with the project. However, this file is included at the discretion of the project creator.

There are several different file types that might be placed in an Encog workbench project. These files are organized by their file extension. The extension of a file is how the Encog Workbench knows what to do with that file. The following extensions are recognized by the Encog Workbench:

- .csv
- .eg
- .ega
- .egb
- .gif
- .html
- .jpg
- .png
- .txt

The following sections will discuss the purpose of each file type.

2.1.1 Workbench CSV Files

An acronym for “comma separated values,” CSV files hold tabular data. However, CSV files are not always “comma separated.” This is especially true in parts of the world that use a decimal comma instead of a decimal point. The CSV files used by Encog can be based on a decimal comma. In this case, a semicolon (;) should be used as the field separator.

CSV files may also have headers to define what each column of the CSV file means. Column headers are optional, but very much suggested. Column headers name the attributes and provide consistency across the both the CSV files created by Encog and provided by the user.

A CSV file defines the data used by Encog. Each row in the CSV file defines a training set element and each column defines an attribute. If a particular attribute is not known for a training set element, then the “?” character should be placed in that row/column. Encog deals with missing values in various ways. This is discussed later in this chapter in the Encog analyst discussion.

A CSV file cannot be used to directly train a neural network, but must first be converted into an EGB file. To convert a CSV file to an EGB file, right-click the CSV file and choose “Export to Training (EGB).” EGB files nicely define what columns are input and ideal data, while CSV files do not offer any distinction. Rather, CSV files might represent raw data provided by the user. Additionally, some CSV files are generated by Encog as raw user data is processed.

2.1.2 Workbench EG Files

Encog EG files store a variety of different object types, but in themselves are simply text files. All data inside of EG files is stored with decimal points and comma separator, regardless of the geographic region in which Encog is running. While CSV files can be formatted according to local number formatting rules, EG files cannot. This is to keep EG files consistent across all Encog platforms.

The following object types are stored in EG files.

- Machine Learning Methods (i.e. Neural Networks)
- NEAT Populations
- Training Continuation Data

The Encog workbench will display the object type of any EG file that is located in the project directory. An Encog EG file only stores one object per file. If multiple objects are to be stored, they must be stored in separate EG files.

2.1.3 Workbench EGA Files

Encog Analyst script files, or EGA files, hold instructions for the Encog analyst. These files hold statistical information about what a CSV file is designed to analyze. EGA files also hold script information that describes how to process raw data. EGA files are executable by the workbench.

A full discussion of the EGA file and every possible configuration/script item is beyond the scope of this book. However, a future book will be dedicated to the Encog Analyst. Additional reference information about the Encog Analyst script file can be found here:

`http://www.heatonresearch.com/wiki/EGA_File`

Later in this chapter, we will create an EGA file to analyze the iris dataset.

2.1.4 Workbench EGB Files

Encog binary files, or EGB files, hold training data. As previously discussed, CSV files are typically converted to EGB for Encog. This data is stored in a platform-independent binary format. Because of this, EGB files are read much faster than a CSV file. Additionally, the EGB file internally contains the number of input and ideal columns present in the file. CSV files must be converted to EGB files prior to training. To convert a CSV file to an EGB file, right-click the selected CSV file and choose "Export to Training (EGB)."

2.1.5 Workbench Image Files

The Encog workbench does not directly work with image files at this point, but can be displayed by double-clicking. The Encog workbench is capable of displaying PNG, JPG and GIF files.

2.1.6 Workbench Text Files

Encog Workbench does not directly use text files. However, text files are a means of storing instructions for project file users. For instance, a readme.txt file can be added to a project and displayed inside of the analyst. The Encog Workbench can display both text and HTML files.

2.2 A Simple XOR Example

There are many different ways that the Encog Workbench can be used. The Encog Analyst can be used to create projects that include normalization, training and analysis. However, all of the individual neural network parts can also manually created and trained. If the data is already normalized, Encog Analyst may not be necessary.

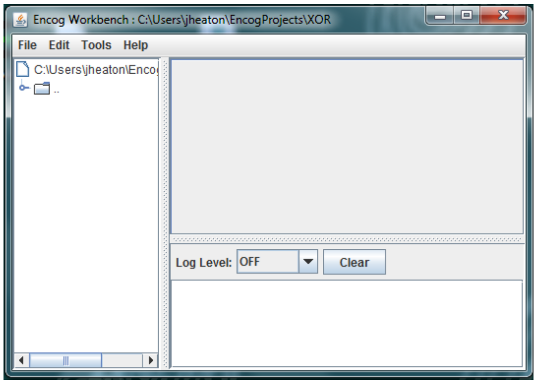
In this section we will see how to use the Encog Workbench without the Encog Analyst by creating a simple XOR neural network. The XOR dataset does not require any normalization as itis already in the 0 to 1 range.

We will begin by creating a new project.

2.2.1 Creating a New Project

First create a new project by launching the Encog Workbench. Once the Encog Workbench starts up, the options of creating a new project, opening an existing project or quitting will appear. Choose to create a new project and name it “XOR.” This will create a new empty folder named XOR. You will now see the Encog Workbench in Figure 2.1.

Figure 2.1: The Encog Workbench



This is the basic layout of the Encog Workbench. There are three main areas. The tall rectangle on the left is where all project files are shown. Currently this project has no files. You can also see the log output and status information. The rectangle just above the log output is where documents are opened. The look of the Encog Workbench is very much like IDE and should be familiar to developers.

2.2.2 Generate Training Data

The next step is to obtain training data. There are several ways to do this. First, Encog Workbench supports drag and drop. For instance, CSVs can be dragged from the operating system and dropped into the project as a copy, leaving the original file unchanged. These files will then appear in the project tree.

The Encog Workbench comes with a number of built-in training sets. Additionally, it can download external data such as stock prices and even sunspot information. The sunspot information can be used for time-series prediction experiments.

The Encog Workbench also has a built-in XOR training set. To access it, choose Tools->Generate Training Data. This will open the "Create Training Data" dialog. Choose "XOR Training Set" and name it "xor.csv." Your new CSV file will appear in the project tree.

If you double-click the "xor.csv" file, you will see the following training data in Listing 2.1:

Listing 2.1: XOR Training Data

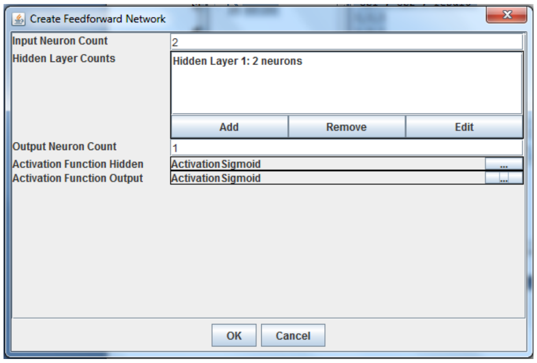
```
"op1","op2","result"
0,0,0
1,0,1
0,1,1
1,1,0
```

It is important to note that the file does have headers. This must be specified when the EGB file is generated.

2.2.3 Create a Neural Network

Now that the training data has been created, a neural network should be created learn the XOR data. To create a neural network, choose "File->New File." Then choose "Machine Learning Method" and name the neural network "xor.eg." Choose "Feedforward Neural Network." This will display the dialog shown in Figure 2.2:

Figure 2.2: Create a Feedforward Network


Create Feedforward Network

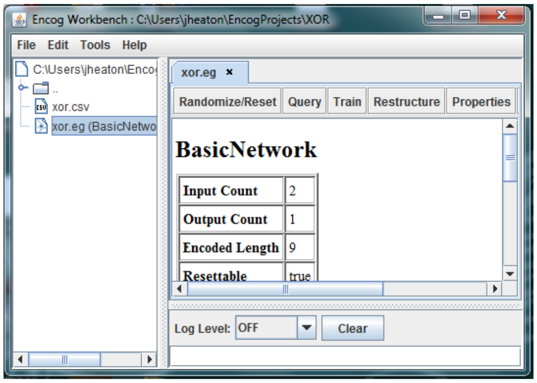
Input Neuron Count: 2
Hidden Layer Counts: Hidden Layer 1: 2 neurons
Output Neuron Count: 1
Activation Function Hidden: ActivationSigmoid
Activation Function Output: ActivationSigmoid

Make sure to fill in the dialog exactly as above. There should be two input neurons, one output neuron and a single hidden layer with two neurons. Choose both activation functions to be sigmoid. Once the neural network is created, it will appear on the project tree.

2.2.4 Train the Neural Network

It is now time to train the neural network. The neural network that you see currently is untrained. To easily determine if the neural network is untrained, double-click the EG file that contains the neural network. This will show Figure 2.3.

Figure 2.3: Editing the Network

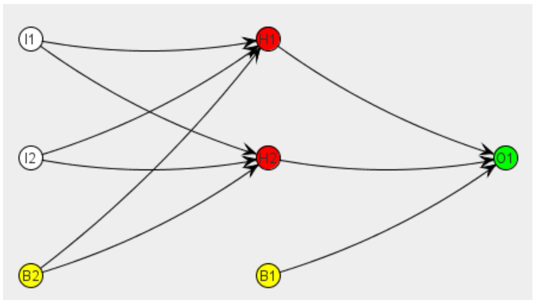


BasicNetwork

Input Count	2
Output Count	1
Encoded Length	9
Resettable	true

This screen shows some basic stats on the neural network. To see more detail, select the "Visualize" button and choose "Network Structure." This will show Figure 2.4.

Figure 2.4: Network Structure



The input and output neurons are shown from the structure view. All of the connections between with the hidden layer and bias neurons are also visible. The bias neurons, as well as the hidden layer, help the neural network to learn.

With this complete, it is time to actually train the neural network. Begin by closing the histogram visualization and the neural network. There should be no documents open inside of the workbench.

Right-click the “xor.csv” training data. Choose “Export to Training (EGB).” Fill in two input neurons and one output neuron on the dialog that appears. On the next dialog, be sure to specify that there are headers. Once this is complete, an EGB file will be added to the project tree. This will result in three files: an EG file, an EGB file and a CSV file.

To train the neural network, choose “Tools->Train.” This will open a dialog to choose the training set and machine learning method. Because there is only one EG file and one EGB file, this dialog should default to the correct values. Leave the “Load to Memory” checkbox clicked. As this is such a small training set, there is no reason to not load to memory.

There are many different training methods to choose from. For this example, choose “Propagation - Resilient.” Accept all default parameters for this training type. Once this is complete, the training progress tab will appear. Click “Start” to begin training.

Training will usually finish in under a second. However, if the training continues for several seconds, the training may need to be reset by clicking the drop list titled “<Select Option>.” Choose to reset the network. Because a neural network starts with random weights, training times will vary. On a small neural network such as XOR, the weights can potentially be bad enough that the network never trains. If this is the case, simply reset the network as it trains.

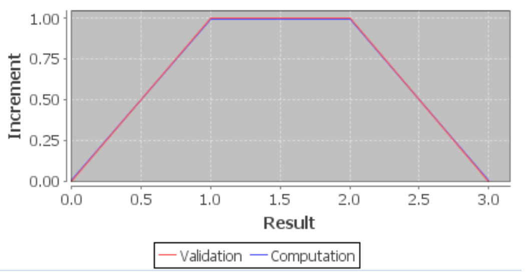
2.2.5 Evaluate the Neural Network

There are two ways to evaluate the neural network. The first is to simply calculate the neural network error by choosing “Tools->Evaluate Network.” You will be prompted for the machine learning method and training data to use. This will show you the neural network error when evaluated against the specified training set.

For this example, the error will be a percent. When evaluating this percent, the lower the percent the better. Other machine learning methods may generate an error as a number or other value.

For a more advanced evaluation, choose “Tools->Validation Chart.” This will result in an output similar to Figure 2.5.

Figure 2.5: Validation Chart for XOR



This graphically depicts how close the neural network's computation matches the ideal value (validation). As shown in this example, they are extremely close.

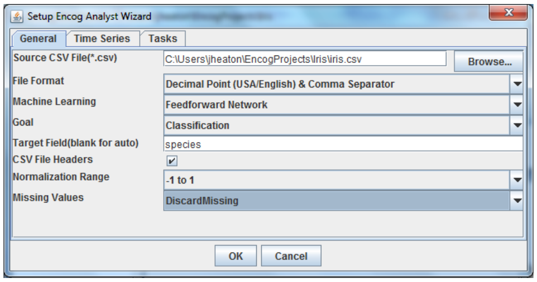
2.3 Using the Encog Analyst

In the last section we used the Workbench with a simple data set that did not need normalization. In this section we will use the Encog Analyst to work with a more complex data set - the iris data set that has already been demonstrated several times. The normalization procedure is already explored. However, this will provide an example of how to normalize and produce a neural network for it using the Encog Analyst

The iris dataset is built into the Encog Workbench, so it is easy to create a dataset for it. Create a new Encog Workbench project as described in the previous section. Name this new project "Iris." To obtain the iris data set, choose "Tools->Generate Training Data." Choose the "Iris Dataset" and name it "iris.csv."

Right-click the "iris.csv" file and choose "Analyst Wizard." This will bring up a dialog like Figure 2.6.

Figure 2.6: Encog Analyst Wizard



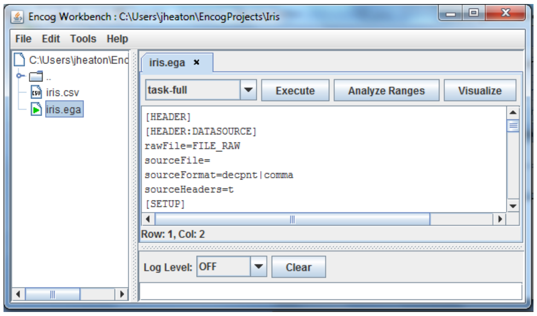
Source CSV File(*.csv): C:\Users\jheaton\EncogProjects\Iris\iris.csv
File Format: Decimal Point (USA/English) & Comma Separator
Machine Learning: Feedforward Network
Goal: Classification
Target Field(blank for auto): species
CSV File Headers: ✓
Normalization Range: -1 to 1
Missing Values: DiscardMissing

You can accept most default values. However, "Target Field" and "CSV File Headers" fields should be changed. Specify "species" as the target and indicate that there are headers. The other two tabs should remain unchanged. Click "OK" and the wizard will generate an EGA file.

This exercise also gave the option to show how to deal with missing values. While the iris dataset has no missing values, this is not the case with every dataset. The default action is to discard them. However, you can also choose to average them out.

Double click this EGA file to see its contents as in Figure 2.7.

Figure 2.7: Edit an EGA File



```
[HEADER]
[HEADER:DATASOURCE]
rawFile=FILE_RAW
sourceFile=
sourceFormat=decpnt|comma
sourceHeaders=t
[SETUP]
```

From this tab you can execute the EGA file. Click "Execute" and a status dialog will be displayed. From here, click "Start" to begin the process. The entire execution should take under a minute on most computers.

- Step 1: Randomize - Shuffle the file into a random order.
- Step 2: Segregate - Create a training data set and an evaluation data set.
- Step 3: Normalize - Normalize the data into a form usable by the selected Machine Learning Method
- Step 4: Generate - Generate the training data into an EGB file that can be used to train.
- Step 5: Create - Generate the selected Machine Learning Method.
- Step 6: Train - Train the selected Machine Learning Method.
- Step 7: Evaluate - Evaluate the Machine Learning Method.

This process will also create a number of files. The complete list of files, in this project is:

- iris.csv - The raw data.
- iris.ega - The EGA file. This is the Encog Analyst script.
- iris_eval.csv - The evaluation data.
- iris_norm.csv - The normalized version of iris_train.csv.
- iris_output.csv - The output from running iris_eval.csv.
- iris_random.csv - The randomized output from running iris.csv.
- iris_train.csv - The training data.
- iris_train.eg - The Machine Learning Method that was trained.
- iris_train.egb - The binary training data, created from iris_norm.egb.

If you change the EGA script file or use different options for the wizard, you may have different steps.

To see how the network performed, open the iris_output.csv file. You will see Listing 2.2.

Listing 2.2: Evaluation of the Iris Data

```
"sepal_l","sepal_w","petal_l","petal_w","species","Output:species"
6.5,3.0,5.8,2.2,Iris-virginica,Iris-virginica
6.2,3.4,5.4,2.3,Iris-virginica,Iris-virginica
7.7,3.0,6.1,2.3,Iris-virginica,Iris-virginica
6.8,3.0,5.5,2.1,Iris-virginica,Iris-virginica
6.5,3.0,5.5,1.8,Iris-virginica,Iris-virginica
6.3,3.3,4.7,1.6,Iris-versicolor,Iris-versicolor
5.6,2.9,3.6,1.3,Iris-versicolor,Iris-versicolor
...
```

This illustrates how the neural network attempts to predict what iris species each row belongs to. As you can see, it is correct for all of the rows shown here. These are data items that the neural network was not originally trained with.

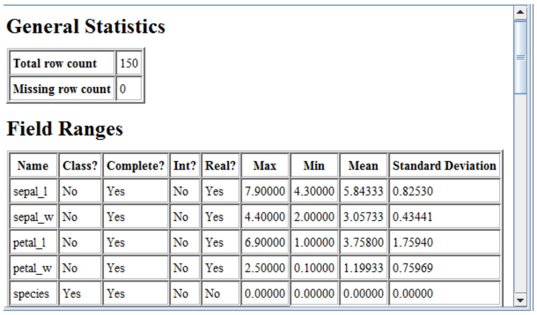
2.4 Encog Analyst Reports

This section will discuss how the Encog Workbench can also produce several Encog Analyst reports. To produce these reports, open the EGA file as seen in Figure 2.7. Clicking the "Visualize" button gives you several visualization options. Choose either a "Range Report" or "Scatter Plot." Both of these are discussed in the next sections.

2.4.1 Range Report

The range report shows the ranges of each of the attributes that are used to perform normalization by the Encog Analyst. Figure 2.8 shows the beginning of the range report.

Figure 2.8: Encog Analyst Range Report



General Statistics

Total row count	150
Missing row count	0

Field Ranges

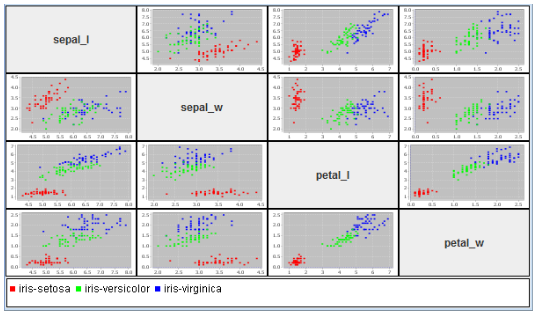
Name	Class?	Complete?	Int?	Real?	Max	Min	Mean	Standard Deviation
sepal_l	No	Yes	No	Yes	7.90000	4.30000	5.84333	0.82530
sepal_w	No	Yes	No	Yes	4.40000	2.00000	3.05733	0.43441
petal_l	No	Yes	No	Yes	6.90000	1.00000	3.75800	1.75940
petal_w	No	Yes	No	Yes	2.50000	0.10000	1.19933	0.75969
species	Yes	Yes	No	No	0.00000	0.00000	0.00000	0.00000

This is only the top portion. Additional information is available by scrolling down.

2.4.2 Scatter Plot

It is also possible to display a scatter plot to view the relationship between two or more attributes. When choosing to display a scatter plot, Encog Analyst will prompt you to choose which attributes to relate. If you choose just two, you are shown a regular scatter plot. If you choose all four, you will be shown a multivariate scatter plot as seen in Figure 2.9.

Figure 2.9: Encog Analyst Multivariate Scatter Plot Report



This illustrates how four variables relate. To see how to variables relate, choose two squares on the diagonal. Follow the row and column on each and the square that intersects is the relationship between those two attributes. It is also important to note that the triangle formed above the diagonal is the mirror image (reverse) of the triangle below the diagonal.

2.5 Summary

This chapter introduced the Encog Workbench. The Encog Workbench is a GUI application that visually works with neural networks and other machine learning methods. The workbench is a Java application that produces data that it works across any Encog platforms.

This chapter also demonstrated how to use Encog Workbench to directly create and train a neural network. For cases where data is already normalized, this is a good way to train and evaluate neural networks. The workbench creates and trains neural networks to accomplish this.

For more complex data, Encog Analyst is a valuable tool that performs automatic normalization. It also organizes a neural network project as a series of tasks to be executed. The iris dataset was used to illustrate how to use the Encog Analyst.

Chapter 3

Constructing Neural Networks in Java

- Constructing a Neural Network
- Activation Functions
- Encog Persistence
- Using the Encog Analyst from Code

This chapter will show how to construct feedforward and simple recurrent neural networks with Encog and how to save these neural networks for later use. Both of these neural network types are created using the **BasicNetwork** and **BasicLayer** classes. In addition to these two classes, activation functions are also used. The role of activation functions will be discussed as well.

Neural networks can take a considerable amount of time to train. Because of this it is important to save your neural networks. Encog neural networks can be persisted using Java's built-in serialization. This persistence can also be achieved by writing the neural network to an EG file, a cross-platform text file. This chapter will introduce both forms of persistence.

In the last chapter, the Encog Analyst was used to automatically normalize data. The Encog Analyst can also automatically create neural networks based on CSV data. This chapter will show how to use the Encog analyst to create neural networks from code.

3.1 Constructing a Neural Network

A simple neural network can quickly be created using **BasicLayer** and **BasicNetwork** objects. 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();
```

This network will have an input layer of two neurons, a hidden layer with three neurons and an output layer with a single neuron. To use an activation function other than the hyperbolic tangent function, use code similar to the following:

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(null,true,2));
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,3));
network.addLayer(new BasicLayer(new ActivationSigmoid(),false,1));
network.getStructure().finalizeStructure();
network.reset();
```

The sigmoid activation function is passed to the **AddLayer** calls for the hidden and output layer. The **true** value that was also introduced specifies that the **BasicLayer** should have a bias neuron. The output layer does not have bias neurons, and the input layer does not have an activation function. This is because the bias neuron affects the next layer, and the activation function affects data coming from the previous layer.

Unless Encog is being used for something very experimental, always use a bias neuron. Bias neurons allow the activation function to shift off the origin of zero. This allows the neural network to produce a zero value even when the inputs are not zero. The following URL provides a more mathematical justification for the importance of bias neurons:

`http://www.heatonresearch.com/wiki/Bias`

Activation functions are attached to layers and 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 an activation function is not specified for **BasicLayer**, the hyperbolic tangent activation will be defaulted.

It is also possible to create context layers. A context layer can be used to create an Elman or Jordan style neural networks. The following code could be used to create an Elman neural network.

```
BasicLayer input, hidden;
BasicNetwork network = new BasicNetwork();
network.addLayer(input = new BasicLayer(1));
network.addLayer(hidden = new BasicLayer(2));
network.addLayer(new BasicLayer(1));
input.setContextFedBy(hidden);
network.getStructure().finalizeStructure();
network.reset();
```

Notice the **hidden.setContextFedBy** line? This creates a context link from the output layer to the hidden layer. The hidden layer will always be fed the output from the last iteration. This creates an Elman style neural network. Elman and Jordan networks are discussed in other books about Encog. For more information, visit:

`http://www.heatonresearch.com/book/programming-neural-networks-encog3-java.html`

3.2 The Role of Activation Functions

The last section illustrated how to assign activation functions to layers. 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. The next sections will introduce these activation functions.

Activation functions are attached to layers and 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 an activation function is not specified for **BasicLayer**, the hyperbolic tangent activation will be the defaulted. All classes that serve as activation functions must implement the **ActivationFunction** interface.

Activation functions play a very important role in training neural networks. Propagation training, which will be covered in the next chapter, requires than an activation function have a valid derivative. Not all activation functions have valid derivatives. Determining if an activation function has a derivative may be an important factor in choosing an activation function.

3.3 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, it is important to consider how the type of neural network being used dictates the activation function required. Secondly, consider the necessity of training the neural network using propagation. Propagation training requires an activation function that provides a derivative. Finally, consider the range of numbers to be used. Some activation functions deal with only positive numbers or numbers in a particular range.

3.3.1 ActivationBiPolar

The **ActivationBiPolar** activation function is used with neural networks that require bipolar values. Bipolar values 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 shown above, 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.

3.3.2 Activation Competitive

The **ActivationCompetitive** function is used to force only a select group of neurons to win. The winner is the group of neurons with the highest output. The outputs of each of these neurons are held in the array passed to this function. The size of the winning neuron group 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. The number of winners is also counted.

```
final boolean[] winners = new boolean[x.length];
double sumWinners = 0;
```

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

```
// find the desired number of winners
for (int i = 0; i < this.params[0]; i++) {
  double maxFound = Double.NEGATIVE_INFINITY;
  int winner = -1;
```

Now, one winner must be determined. Loop over all of the neuron outputs and find the one with the highest output.

```
  for (int j = start; j < start + size; j++) {
```

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

```
    if (!winners[j] && (x[j] > maxFound)) {
      winner = j;
      maxFound = x[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 the correct number of winners is determined, the values must be adjusted 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.

```
// adjust weights for winners and non-winners
for (int i = start; i < start + size; i++) {
  if (winners[i]) {
    x[i] = x[i] / sumWinners;
    } else {
      x[i] = 0.0;
    }
}
```

This sort of an activation function can be used with competitive, learning neural networks such as the self-organizing map. This activation function has no derivative, so it cannot be used with propagation training.

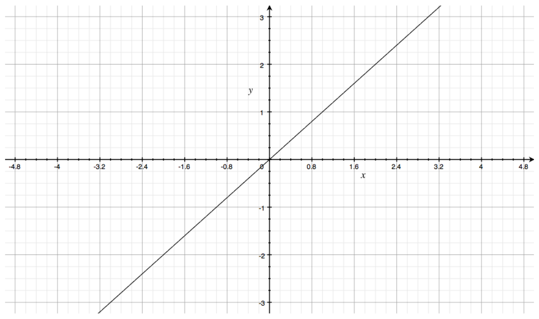
3.3.3 ActivationLinear

The **ActivationLinear** function is really no activation function at all. It simply implements the linear function. The linear function can be seen in Equation 3.1.

$$f(x) = x \tag{3.1}$$

The graph of the linear function is a simple line, as seen in Figure 3.1.

Figure 3.1: 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 final void activationFunction(final double[] x, final int start ,
final int size) {
}
```

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

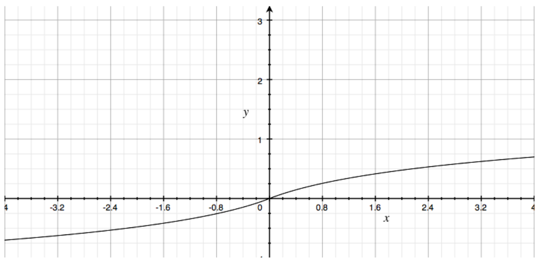
3.3.4 ActivationLOG

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

$$f(x) = \begin{cases} \log(1+x) & , x >= 0 \\ \log(1-x) & , \text{otherwise} \end{cases} \tag{3.2}$$

This produces a curve similar to the hyperbolic tangent activation function, which will be discussed later in this chapter. The graph for the logarithmic activation function is shown in Figure 3.2.

Figure 3.2: 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 3.2, 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.

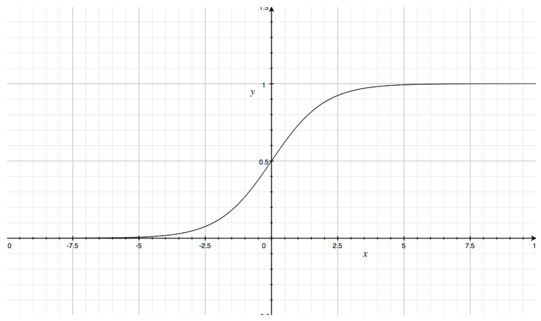
3.3.5 ActivationSigmoid

The **ActivationSigmoid** activation function should only be used when positive number output is expected because the **ActivationSigmoid** function will only produce positive output. The equation for the **ActivationSigmoid** function can be seen in Equation 3.2.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

The **ActivationSigmoid** function will move negative numbers into the positive range. This can be seen in Figure 3.3, which shows the graph of the sigmoid function.

Figure 3.3: Graph of the ActivationSigmoid Function



The **ActivationSigmoid** function is a very common choice for feedforward and simple recurrent neural networks. However, it is imperative that the training data does not expect negative output numbers. If negative numbers are required, the hyperbolic tangent activation function may be a better solution.

3.3.6 ActivationSoftMax

The **ActivationSoftMax** activation function will scale all of the input values so that the 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 = start; i < start + size; i++) {
  x[i] = x[i] / sum;
}
```

The **ActivationSoftMax** is typically used in the output layer of a neural network for classification.

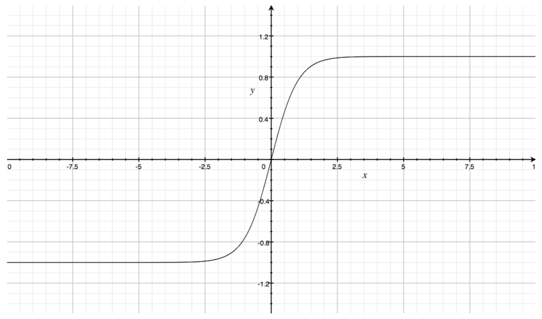
3.3.7 ActivationTANH

The **ActivationTANH** activation function 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 3.3.

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.4}$$

The fact that the hyperbolic tangent activation function accepts both positive and negative numbers can be seen in Figure 3.4, which shows the graph of the hyperbolic tangent function.

Figure 3.4: Graph of the Hyperbolic Tangent Activation Function



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.

3.4 Encog Persistence

It can take considerable time to train a neural network and it is important to take measures to guarantee your work is saved once the network has been trained. Encog provides several means for this data to be saved, with two primary ways to store Encog data objects. Encog offers file-based persistence or Java's own persistence.

Java provides its own means to serialize objects and is called Java serialization. Java serialization allows many different object types to be written to a stream, such as a disk file. Java serialization for Encog works the same way as with any Java object using Java serialization. Every important Encog object that should support serialization implements the **Serializable** interface.

Java serialization is a quick way to store an Encog object. However, it has some important limitations. The files created with Java serialization can only be used by Encog for Java; they will be incompatible with Encog for .Net or Encog for Silverlight. Further, Java serialization is directly tied to the underlying objects. As a result, future versions of Encog may not be compatible with your serialized files.

To create universal files that will work with all Encog platforms, consider the Encog EG format. The EG format stores neural networks as flat text files ending in the extension .EG.

This chapter will introduce both methods of Encog persistence, beginning with Encog EG persistence. The chapter will end by exploring how a neural network is saved in an Encog persistence file.

3.5 Using Encog EG Persistence

Encog EG persistence files are the native file format for Encog and are stored with the extension .EG. The Encog Workbench uses the Encog EG to process files. This format can be exchanged over different operating systems and Encog platforms, making it the choice format choice for an Encog application.

This section begins by looking at an XOR example that makes use of Encog's EG files. Later, this same example will be used for Java serialization. We will begin with the Encog EG persistence example.

3.5.1 Using Encog EG Persistence

Encog EG persistence is very easy to use. The **EncogDirectoryPersistence** class is used to load and save objects from an Encog EG file. The following is a good example of Encog EG persistence:

```
org.encog.examples.neural.persist.EncogPersistence
```

This example is made up of two primary methods. The first method, **trainAndSave**, trains a neural network and then saves it to an Encog EG file. The second method, **loadAndEvaluate,** loads the Encog EG file and evaluates it. This proves that the Encog EG file was saved correctly. The **main** method simply calls these two in sequence. We will begin by examining the **trainAndSave** method.

```
public void trainAndSave() {
  System.out.println(
    "Training XOR network to under 1% error rate.");
```

This method begins by creating a basic neural network to be trained with the XOR operator. It is a simple three-layer feedforward neural network.

```
  BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(2));
  network.addLayer(new BasicLayer(6));
  network.addLayer(new BasicLayer(1));
  network.getStructure().finalizeStructure();
  network.reset();
```

A training set is created that contains the expected outputs and inputs for the XOR operator.

```
  MLDataSet trainingSet =
    new BasicMLDataSet(XOR_INPUT, XOR_IDEAL);
```

This neural network will be trained using resilient propagation (RPROP).

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

RPROP iterations are performed until the error rate is very small. Training will be covered in the next chapter. For now, training is a means to verify that the error remains the same after a network reload.

```
    do {
      train.iteration();
    } while(train.getError() > 0.009);
```

Once the network has been trained, display the final error rate. The neural network can now be saved.

```
    double e = network.calculateError(trainingSet);
    System.out.println("Network traiined to error: "
    + e);
    System.out.println("Saving network");
```

The network can now be saved to a file. Only one Encog object is saved per file. This is done using the **saveObject** method of the **EncogDirectoryPersistence** class.

```
System.out.println("Saving network");
EncogDirectoryPersistence.saveObject(new File(FILENAME), network);
```

Now that the Encog EG file has been created, load the neural network back from the file to ensure it still performs well using the **loadAndEvaluate** method.

```
public void loadAndEvaluate()
```

```
{
  System.out.println("Loading network");
  BasicNetwork network =
   (BasicNetwork)EncogDirectoryPersistence.loadObject(
   new File(FILENAME));
```

Now that the collection has been constructed, load the network named **network** that was saved earlier. It is important to evaluate the neural network to prove that it is still trained. To do this, create a training set for the XOR operator.

```
  MLDataSet trainingSet =
    new BasicMLDataSet(XOR_INPUT, XOR_IDEAL);
```

Calculate the error for the given training data.

```
  double e = network.calculateError(trainingSet);
  System.out.println(
    "Loaded network's error is(should be same as above): " + e);
}
```

This error is displayed and should be the same as before the network was saved.

3.6 Using Java Serialization

It is also possible to use standard Java serialization with Encog neural networks and training sets. Encog EG persistence is much more flexible than Java serialization. However, there are cases a neural network can simply be saved to a platform-dependant binary file. This example shows how to use Java serialization with Encog. The example begins by calling the **trainAndSave** method.

```
public void trainAndSave() throws IOException {
  System.out.println(
    "Training XOR network to under 1% error rate.");
```

This method begins by creating a basic neural network to be trained with the XOR operator. It is a simple, three-layer feedforward neural network.

```
  BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(2));
  network.addLayer(new BasicLayer(6));
  network.addLayer(new BasicLayer(1));
  network.getStructure().finalizeStructure();
  network.reset();
  MLDataSet trainingSet =
    new BasicMLDataSet(XOR_INPUT, XOR_IDEAL);
```

We will train this neural network using resilient propagation (RPROP).

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

The following code loops through training iterations until the error rate is below one percent (<0.01).

```
  do {
    train.iteration();
  } while(train.getError() > 0.01);
```

The final error for the neural network is displayed.

```
  double e = network.calculateError(trainingSet);
  System.out.println("Network traiined to error: " + e);
  System.out.println("Saving network");
```

Regular Java Serialization code can be used to save the network or the **SerializeObject** class can be used. This utility class provides a **save** method that will write any single serializable object to a binary file. Here the **save** method is used to save the neural network.

```
  SerializeObject.save(FILENAME, network);
}
```

Now that the binary serialization file is created, load the neural network back from the file to see if it still performs well. This is performed by the **loadAndEvaluate** method.

```
public void loadAndEvaluate()
  throws IOException, ClassNotFoundException {
  System.out.println("Loading network");
```

The **SerializeObject** class also provides a **load** method that will read an object back from a binary serialization file.

```
  BasicNetwork network =
    (BasicNetwork) SerializeObject.load(FILENAME);
  MLDataSet trainingSet =
    new BasicMLDataSet(XOR_INPUT, XOR_IDEAL);
```

Now that the network is loaded, the error level is reported.

```
  double e = network.calculateError(trainingSet);
  System.out.println(
    "Loaded network's error is(should be same as above): " + e);
}
```

This error level should match the error level at the time the network was originally trained.

3.7 Summary

Feedforward and Simple Recurrent Neural Networks are created using the **BasicNetwork** and **BasicLayer** classes. Using these objects, neural networks can be created. Layers can also be connected using context links, just as simple recurrent neural networks, such as the Elman neural network, are created.

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. If the activation function is to be used with propagation training, the activation function must be able to calculate for its derivative.

The **ActivationBiPolar** activation function class is used when a network only accepts bipolar numbers. The **ActivationCompetitive** activation function class is used for competitive neural networks such as the self-organizing map. The **ActivationLinear** activation function class is used when no activation function is desired. The **ActivationLOG** activation function class works similarly to the **ActivationTANH** activation function class except it does not always 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 **ActivationSoftMax** activation function class scales the output so that the sum is one.

This chapter illustrated how to persist Encog objects using two methods. Objects may be persisted by using either the Encog EG format or by Java serialization.

The Encog EG format is the preferred means for saving Encog neural networks. These objects are accessed using their resource name. The EG file can be interchanged between any platform that Encog supports.

Encog also allows Java serialization to store objects to disk or stream. Java serialization is more restrictive than Encog EG files. Because the binary files are automatically stored directly from the objects, even the smallest change to an Encog object can result in incompatible files. Additionally, other platforms will be unable to use the file.

In the next chapter the concept of neural network training is introduced. Training is the process where the weights of a neural network are modified to produce the desired output. There are several ways neural networks can be trained. The next chapter will introduce propagation training.

Chapter 4

Propagation Training

- How Propagation Training Works
- Propagation Training Types
- Training and Method Factories
- Multithreaded Training

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

Encog also supports unsupervised training. With unsupervised training, the neural network is not provided with the expected output. Rather, the neural network learns and makes insights into the data with limited direction. Unsupervised training is discussed in other books about Encog. For more information, visit:

`http://www.heatonresearch.com/book/programming-neural-networks-encog3-java.html`

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

- Backpropagation Training
- Quick Propagation Training (QPROP)
- Manhattan Update Rule
- Resilient Propagation Training (RPROP)
- Scaled Conjugate Gradient (SCG)
- Levenberg Marquardt (LMA)

All six of these methods work somewhat similarly. However, there are some important differences. The next section will explore propagation training in general.

4.1 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 that will most likely improve the neural network's error rate 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 will be calculated. These changes will be applied in batches using Encog's batch training. Therefore, Encog updates the weight matrix values at the end of an 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, i.e. the difference between the actual and ideal outputs. 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 and actual outputs from each of the neurons. The gradient of this error is then calculated. To calculate this gradient, the neural network's actual output is applied to the derivative of the activation function used for this level. This value is then multiplied by the error.

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

4.1.1 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 which is essentially a percent that determines how directly the gradient descent should be applied to the weight matrix. The gradient is multiplied by the learning rate and then added to the weight matrix. This slowly optimizes 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 may not be a global minimum. The second parameter provided to the backpropagation algorithm helps the backpropagation out of local minima. The second parameter is called momentum. Momentum specifies to what degree the previous iteration weight changes 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 50% of the previous training iteration's changes to be applied to the weights for the current weight matrix.

The following code will setup a backpropagation trainer, given a training set and neural network.

```
Backpropagation train = new Backpropagation(network, trainingSet, 0.7, 0.3);
```

The above code would create a backpropagation trainer with a learning rate of 0.7 and a momentum of 0.3. Once setup the training object is ready for iteration training. For an example of Encog iteration training see:

```
org.encog.examples.neural.xor.HelloWorld
```

The above example can easily be modified to use backpropagation training by replacing the resilient propagation training line with the above training line.

4.1.2 Understanding the Manhattan Update Rule

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

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

The following code will setup a Manhattan update trainer given a training set and neural network.

```
final ManhattanPropagation train =
  new ManhattanPropagation(network, trainingSet, 0.00001);
```

The above code would create a Manhattan Update Rule trainer with a learning rate of 0.00001. Manhattan propagation generally requires a small learning rate. Once setup is complete, the training object is ready for iteration training. For an example of Encog iteration training see:

```
org.encog.examples.neural.xor.HelloWorld
```

The above example can easily be modified to use Manhattan propagation training by replacing the resilient propagation training line with the above training line.

4.1.3 Understanding Quick Propagation Training

Quick propagation (QPROP) is another variant of propagation training. Quick propagation is based on Newton's Method, which is a means of finding a function's roots. This can be adapted to the task of minimizing the error of a neural network. Typically QPROP performs much better than backpropagation. The user must provide QPROP with a learning rate parameter. However, there is no momentum parameter as QPROP is typically more tolerant of higher learning rates. A learning rate of 2.0 is generally a good starting point.

The following code will setup a Quick Propagation trainer, given a training set and neural network.

```
QuickPropagation train =
  new QuickPropagation(network, trainingSet, 2.0);
```

The above code would create a QPROP trainer with a learning rate of 2.0. QPROP can generally take a higher learning rate. Once setup, the training object is ready for iteration training. For an example of Encog iteration training see:

```
org.encog.examples.neural.xor.HelloWorld
```

The above example can easily be modified to use QPROP training by replacing the resilient propagation training line with the above training line.

4.1.4 Understanding Resilient Propagation Training

The resilient propagation training (RPROP) algorithm is often 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 parameter setting 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 optimal learning rate.

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 weight values, a much more granular approach is used. These deltas will not remain fixed like in the Manhattan Update Rule or backpropagation algorithm. Rather, these delta values will change as training progresses.

The RPROP algorithm does not keep one global update value, or delta. Rather, individual deltas are kept for every weight matrix value. These deltas are first initialized to a very small number. Every iteration through the RPROP algorithm will update the weight 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 weight matrix value to be individually trained, an advantage not provided by either the backpropagation algorithm or the Manhattan Update Rule.

The following code will setup a Resilient Propagation trainer, given a training set and neural network.

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

The above code would create a RPROP trainer. RPROP requires no parameters to be set to begin training. This is one of the main advantages of the RPROP training algorithm. Once setup, the training object is ready for iteration training. For an example of Encog iteration training see:

```
org.encog.examples.neural.xor.HelloWorld
```

The above example already uses RPROP training.

There are four main variants of the RPROP algorithm that are supported by Encog:

- RPROP+
- RPROP-
- iRPROP+
- iPROP-

By default, Encog uses RPROP+, the most standard RPROP. Some research indicates that iRPROP+ is the most efficient RPROP algorithm. To set Encog to use iRPROP+ use the following command.

```
train.setRPROPType(RPROPType.iRPROPp);
```

4.1.5 Understanding SCG Training

Scaled Conjugate Gradient (SCG) is a fast and efficient training for Encog. SCG is based on a class of optimization algorithms called Conjugate Gradient Methods (CGM). SCG is not applicable for all data sets. When it is used within its applicability, it is quite efficient. Like RPROP, SCG is at an advantage as there are no parameters that must be set.

The following code will setup an SCG trainer, given a training set and neural network.

```
ScaledConjugateGradient train
  = new ScaledConjugateGradient(network, trainingSet);
```

The above code would create a SCG trainer. Once setup, the training object is ready for iteration training. For an example of Encog iteration training see:

```
org.encog.examples.neural.xor.HelloWorld
```

The above example can easily be modified to use SCG training by replacing the resilient propagation training line with the above training line.

4.1.6 Understanding LMA Training

The Levenberg Marquardt algorithm (LMA) is a very efficient training method for neural networks. In many cases, LMA will outperform Resilient Propagation. LMA is a hybrid algorithm based on both Newton's Method and gradient descent (backpropagation), integrating the strengths of both. Gradient descent is guaranteed to converge to a local minimum, albeit slowly. GNA is quite fast but often fails to converge. By using a damping factor to interpolate between the two, a hybrid method is created.

The following code shows how to use Levenberg-Marquardt with Encog for Java.

```
LevenbergMarquardtTraining train = new LevenbergMarquardtTraining(network,
    trainingSet);
```

The above code would create an LMA with default parameters that likely require no adjustments. Once setup, the training object is ready for iteration training. For an example of Encog iteration training see:

```
org.encog.examples.neural.xor.HelloWorld
```

The above example can easily be modified to use LMA training by replacing the resilient propagation training line with the above training line.

4.2 Encog Method & Training Factories

This chapter illustrated how to instantiate trainers for many different training methods using objects such as **Backpropagation**, **ScaledConjugateGradient** or **ResilientPropagation**. In the previous chapters, we learned to create different types of neural networks using **BasicNetwork** and **BasicLayer**. We can also create training methods and neural networks using factories.

Factories create neural networks and training methods from text strings, saving time by eliminating the need to instantiate all of the objects otherwise necessary. For an example of factory usage see:

```
org.encog.examples.neural.xor.XORFactory
```

The above example uses factories to create both neural networks and training methods. This section will show how to create both neural networks and training methods using factories.

4.2.1 Creating Neural Networks with Factories

The following code uses a factory to create a feedforward neural network:

```
MLMethodFactory methodFactory = new MLMethodFactory();
MLMethod method = methodFactory.create(
    MLMethodFactory.TYPE_FEEDFORWARD,
    "?:B->SIGMOID->4:B->SIGMOID->?",
    2,
    1);
```

The above code creates a neural network with two input neurons and one output neuron. There are four hidden neurons. Bias neurons are placed on the input and hidden layers. As is typical for neural networks, there are no bias neurons on the output layer. The sigmoid activation function is used between both the input and hidden neuron, as well between the hidden and output layer.

You may notice the two question marks in the neural network architecture string. These will be filled in by the input and output layer sizes specified in the create method and are optional. You can hard-code the input and output sizes. In this case the numbers specified in the create call will be ignored.

4.2.2 Creating Training Methods with Factories

It is also possible to create a training method using a factory. The following code creates a backpropagation trainer using a factory.

```
MLTrainFactory trainFactory = new MLTrainFactory();
MLTrain train = trainFactory.create(
  network,
  dataSet,
  MLTrainFactory.TYPE_BACKPROP,
  "LR=0.7,MOM=0.3"
  );
```

The above code creates a backpropagation trainer using a learning rate of 0.7 and a momentum of 0.3.

4.3 How Multithreaded Training Works

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

This chapter has already introduced three propagation training techniques, all of which work similarly. Whether it is backpropagation, resilient propagation or the Manhattan Update Rule, the technique is similar. There are 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.
```

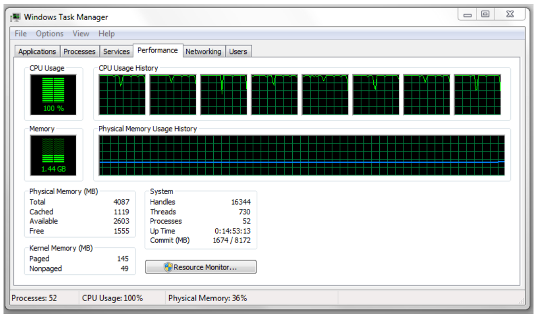
First, a regular feed forward pass is performed. The output from each level is kept so the error for each level can be evaluated independently. Second, the errors are calculated at each level and the derivatives of each activation function are used to calculate gradient descents. These gradients show 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 scales the gradient descents by a learning rate. The scaled gradient descents are then directly applied to the weights. The Manhattan Update Rule only uses the gradient sign 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 only uses the sign of the gradient descent to increase or decrease the delta amounts. The delta amounts are then applied to the weights.

The multithreaded algorithm uses threads to perform Steps 1 and 2. The training data is broken into packets that are distributed among the threads. At the beginning of each iteration, threads are started to handle each of these packets. Once all threads have completed, a single thread aggregates all of the results and applies them to the neural network. At the end of the iteration, there is a very brief amount of time where only one thread is executing. This can be seen from Figure 4.1.

Figure 4.1: Encog Training on a Hyperthreaded Quadcore



```
Physical Memory (MB)            System
Total          4087             Handles        16344
Cached         1119             Threads          730
Available      2603             Processes         52
Free           1555             Up Time     0:14:53:13
                                Commit (MB)  1674 / 8172
Kernel Memory (MB)
Paged           145
Nonpaged         49

Processes: 52    CPU Usage: 100%    Physical Memory: 36%
```

As shown in the above image, the i7 is currently running at 100%. The end of each iteration is clearly identified by 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. In attempting to overcome this, various implementations tested not forcing the threads to wait at the end of the iteration for a resynchronization. This method did not provide efficient training because the propagation training algorithms need all changes applied before the next iteration begins.

4.4 Using Multithreaded Training

To see multithreaded training really shine, a larger training set is needed. However, for now, we will look a simple benchmarking example that generates a random training set and compares multithreaded and single-threaded training times.

A simple benchmark is shown that makes use of an input layer of 40 neurons, a hidden layer of 60 neurons, and an output layer of 20 neurons. A training set of 50,000 elements is used. This example can be found at the following location.

```
org.encog.examples.neural.benchmark.MultiBench
```

Executing this program on a Quadcore i7 with Hyperthreading produced the following result:

```
Training 20 Iterations with Single-threaded
Iteration #1 Error:1.0594453784075148
Iteration #2 Error:1.0594453784075148
Iteration #3 Error:1.0059791059086385
Iteration #4 Error:0.955845375587124
Iteration #5 Error:0.934169803870454
Iteration #6 Error:0.9140418793336804
Iteration #7 Error:0.8950880473422747
Iteration #8 Error:0.8759150228219456
Iteration #9 Error:0.8596693523930371
Iteration #10 Error:0.843578483629412
Iteration #11 Error:0.8239688415389107
Iteration #12 Error:0.8076160458145523
Iteration #13 Error:0.7928442431442133
Iteration #14 Error:0.7772585699972144
Iteration #15 Error:0.7634533283610793
Iteration #16 Error:0.7500401666509937
Iteration #17 Error:0.7376158116045242
Iteration #18 Error:0.7268954113068246
Iteration #19 Error:0.7155784667628093
Iteration #20 Error:0.705537166118038
RPROP Result:35.134 seconds.
Final RPROP error: 0.6952141684716632
Training 20 Iterations with Multithreading
Iteration #1 Error:0.6952126315707992
Iteration #2 Error:0.6952126315707992
Iteration #3 Error:0.90915249248788
Iteration #4 Error:0.8797061675258835
Iteration #5 Error:0.8561169673033431
Iteration #6 Error:0.7909509694056177
Iteration #7 Error:0.7709539415065737
Iteration #8 Error:0.7541971172618358
Iteration #9 Error:0.7287094412886507
Iteration #10 Error:0.715814914438935
Iteration #11 Error:0.7037730808705016
Iteration #12 Error:0.6925902585055886
Iteration #13 Error:0.6784038181007823
Iteration #14 Error:0.6673310323078667
Iteration #15 Error:0.6585209150749294
Iteration #16 Error:0.6503710867148986
Iteration #17 Error:0.6429473784897797
Iteration #18 Error:0.6370962075614478
Iteration #19 Error:0.6314478792705961
Iteration #20 Error:0.6265724296587237
Multi-Threaded Result:8.793 seconds.
Final Multi-thread error: 0.6219704300851074
Factor improvement:4.0106783805299674
```

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

4.5 Summary

This chapter explored how to use several propagation training algorithms with Encog. Propagation training is a very common class of supervised training algorithms. Resilient propagation training is usually the best choice; however, the Manhattan Update Rule and backpropagation may be useful for certain situations. SCG and QPROP are also solid training algorithms.

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 update the weight 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.

SCG and QPROP are also very effective training methods. SCG does not work well for all sets training data, but it is very effective when it does work. QPROP works similar to RPROP. It can be an effective training method. However, QPROP requires the user to choose a learning rate.

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

This guide was intended to get you started with Encog. It is distributed as a free ebook with the Encog framework. In addition this guide I also sell additional, more in-depth, books on Encog, machine learning and neural networks. You can find more information about these books from the following URL. These books can be purchased in paperback or ebook form.

`http://www.heatonresearch.com/book`

This book is a shortened version of the full book ”Programming Neural Networks with Encog3 in Java, 2nd Edition”. If you would like to buy the full ebook(or paperback) visit the following URL.

`http://www.heatonresearch.com/book/programming-neural-networks-encog3-java.html`

Index