Encog 3.3: Development Guide

Encog 3.3: Development Guide

Jeff Heaton

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

Title	Encog 3.3: Development Guide
Author	Jeff Heaton
Published	October 01, 2014
Copyright	Copyright 2014 by Heaton Research, Inc., All Rights Reserved.
ISBN	
Price	FREE
File Created	Sat Oct 11 16:08:17 CDT 2014

Contents

Chapter 1

Building the Encog Core Files

- Acquiting the Encog Core Files
- Building Encog for Java
- Building Encog for C#

This guide explains how to get started developing and extending the Encog framework. This is useful if you want to extend the Encog framework, or perhaps fix a bug. If you simply want to make use of Encog in a project, this is not the appropriate guide for you. To learn to use Encog, refer to either the Encog Quick Start or Users Guide located at the following URL:

`http://www.encog.org`

The first step is to learn to acquire and compile the Encog core.

1.1 Acquiring the Encog Core Files

The Encog project makes use of the **git** version control system. There several repositories that makes up the complete Encog project. These repositories are all contained at the following GitHub URL:

`https://github.com/encog`

The Encog core repositories are located at the following two URLs:

- Java: https://github.com/encog/encog-java-core
- C#: https://github.com/encog/encog-dotnet-core

If you would like to contribute code back to the Encog project, it is suggested that you fork one of the above projects. You can then issue a pull request for your forked copy of Encog once you complete your changes. If your changes are accepted, they will be merged into the main Encog project. For more information on forking a GitHub project refer to the following URL:

`https://help.github.com/articles/fork-a-repo/`

If you would just like to pull the latest Encog core source code into a project you can simply clone the Encog core repository. No fork is needed. To clone the Encog Java repository the following command can be used:

```
git clone https://github.com/encog/encog-java-core.git
```

If you would like to clone the C# version of Encog, you can use the following command:

```
git clone https://github.com/encog/encog-sample-csharp.git
```

The above two commands assume that you are working from the command line, and have **git** installed for your operating system. For more information on this procedure, refer to the following URL:

`https://help.github.com/articles/duplicating-a-repository/`

Now that you have the Encog files, you are ready to compile the core.

1.2 Compiling Encog Core for Java

Encog for Java is compiled using the Maven build system. For information on installing Maven, refer to the following URL:

`http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html`

You can build Encog by using the following Maven command.

```
mvn package
```

This will build the Encog project and create the Encog JAR file. You should not use the deploy directive with Maven, as you will be prompted for a "GPG Passphrase". This is used to deploy Encog to the Maven central repository. For security purposes, this activity is restricted to Jeff Heaton, the lead maintainer of Encog.

1.3 Compiling Encog Core for C#

It is very easy to compile Encog for C#. Simply launch Visual Studio with the "encog-core-cs.sln" solution file. Choose a release build and rebuilt all. The Encog DLL wil be placed in the following directory.

```
..\encog-dotnet-core\encog-core-cs\bin\Release
```

Chapter 2

Overall Structure

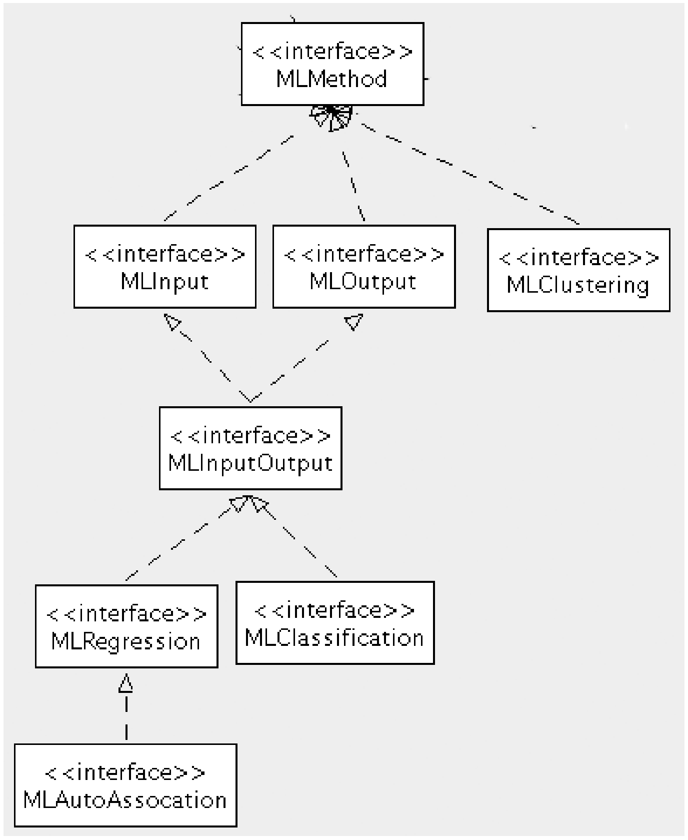
- Machine Learning Methods
- Encog Datasets
- Propagation Training

This chapter discusses the overall structure of the Encog framework. Encog contains many Java/C# interfaces that allow you to create compatible classes that can be used in conjunction with Encog. Each section in this chapter describes one major section of Encog that is commonly a target for enhancement.

2.1 Machine Learning Methods

Each model type in Encog is represented as a machine learning method. These machine-learning methods all ultimately descend from the **MLMethod** (Java) or **IMLMethod** (C#) interface. Figure 2.1 shows the **MLMethod** interface and all of its immediate sub-class interfaces.

Figure 2.1: Encog Machine Learning Method Interfaces



The three most important interfaces are:

- **MLRegression/IMLRegression** - Used to define regression models, or those that produce a numeric output.
- **MLClassification/ IMLClassification** - Used to define classification models, or those that use the output to place the input as into a class.
- **MLClustering/IMLClustering** - Used to define clustering algorithms that take input data and place them into a several clusters.

The next sections detail what methods you must add to create one of these three types of classes.

2.1.1 Creating an Encog Regression Class

The following code shows a template Encog regression class. This shows you every method that you would need to implement from the **MLRegression** interface.

```
public class MyEncogRegression implements MLRegression {
@Override
public int getInputCount() {
return 0;
}
@Override
public int getOutputCount() {
return 0;
}
@Override
public MLData compute(MLData input) {
return null;
}
}
```

The equivalent C# class is very similar.

```
class MyEncogRegression: IMLRegression
{
    public IMLData Compute(IMLData input)
    {
        throw new NotImplementedException();
    }
    public int InputCount
    {
          get { throw new NotImplementedException(); }
    }
    public int OutputCount
    {
        get { throw new NotImplementedException(); }
    }
}
```

You must provide **InputCount** and **OutputCount** properties to report how many inputs and outputs your model expects. Additionally, you must provide a **Compute** method that will accept an input vector to the model and produce an output vector. You can easily create a class that supports both classification and regression by implementing both the **MLRegression** and **MLClassification** interfaces.

2.1.2 Creating an Encog Classification Class

The following code shows a template Encog classification class. This shows you every method that you would need to implement from the **MLClassification** interface.

```
public class MyEncogClassification implements MLClassification {
@Override
public int getInputCount() {
// TODO Auto-generated method stub
return 0;
}
@Override
public int getOutputCount() {
// TODO Auto-generated method stub
return 0;
}
@Override
public int classify(MLData input) {
// TODO Auto-generated method stub
return 0;
}
}
```

The equivalent C# class is very similar.

```
class MyEncogClassification: IMLClassification
{
    public int Classify(Encog.ML.Data.IMLData input)
    {
        throw new NotImplementedException();
    }
    public int InputCount
    {
        get { throw new NotImplementedException(); }
    }
    public int OutputCount
    {
        get { throw new NotImplementedException(); }
    }
}
```

You must provide **InputCount** and **OutputCount** properties to report how many inputs and outputs your model expects. Additionally, you must provide a **Classify** method that will accept an input vector to the model and produce an output class number. You can easily create a class that supports both classification and regression by implementing both the **MLRegression** and **MLClassification** interfaces.

2.1.3 Creating an Encog Clustering Class

The following code shows a template Encog classification class. This shows you every method that you would need to implement from the **MLClustering** interface.

```
public class MyEncogCluster implements MLClustering {
@Override
public void iteration() {
}
@Override
public void iteration(int count) {
}
@Override
public MLCluster[] getClusters() {
return null;
}
@Override
public int numClusters() {
return 0;
}
}
```

The equivalent C# class is very similar.

```
class MyEncogClustering: IMLClustering
{
    public IMLCluster[] Clusters
    {
        get { throw new NotImplementedException(); }
    }
    public int Count
    {
        get { throw new NotImplementedException(); }
    }
    public void Iteration(int count)
    {
        throw new NotImplementedException();
    }
    public void Iteration()
    {
        throw new NotImplementedException();
    }
}
```

You must provide a **Clusters** property to return a list of the final clusters. The **Count** property returns the number of clusters targeted. The **Iteration** method is called to perform one iteration of clustering. Some cluster algorithms will perform their entire clustering run on a single call to **Iteration**.

2.2 Encog Datasets

Encog needs data to fit the various machine learning methods. These data are accessed using a variety of data set classes. All Encog dataset handling objects work with the following three interfaces:

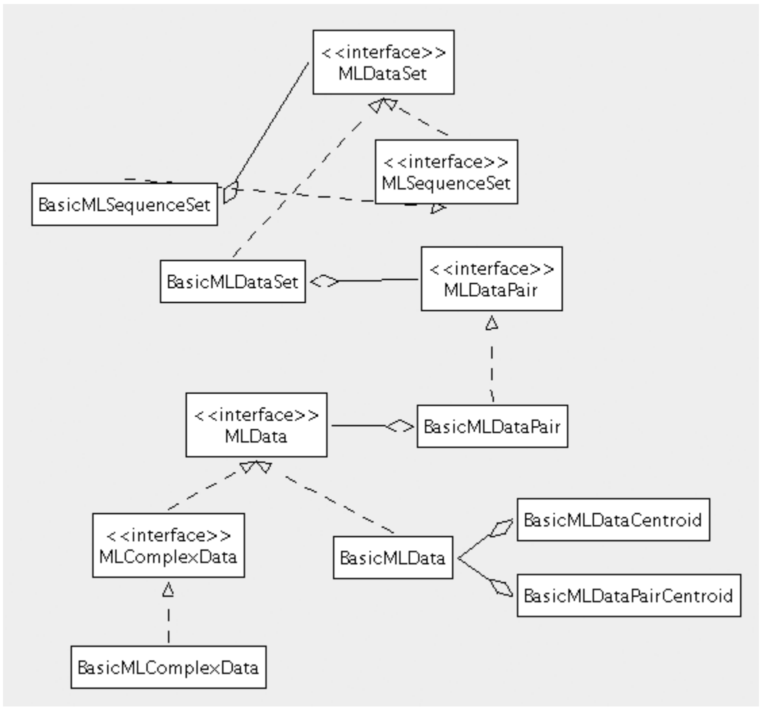
- **MLData/IMLData** - Used to hold a vector that will be the input or output either to or from a model.
- **MLDataPair/IMLDataPair** - Used to both the input and ideal **MLData** vectors for supervised training. A training set is built up of these object types.
- **MLDataSet/IMLDataSet** - Provides lists of **MLDataPair** objects to trainers.

You can create new versions of any of these three. However, you will typically create new datasets, by implementing MLDataSet. Encog provides a basic implementation of all three of the above:

- BasicMLData
- BasicMLDataPair
- BasicMLDataSet

Often the above objects will be enough to handle many tasks. Encog also provides several dataset variants for particular needs. The interfaces and basic datasets are shown in Figure 2.2.

Figure 2.2: Encog Dataset Classes



A Java template for a data set is shown here.

```
public class MyEncogDataSet implements MLDataSet {
@Override
public Iterator<MLDataPair> iterator() {
return null;
}
```

```
@Override
public int getIdealSize() {
return 0;
}
@Override
public int getInputSize() {
return 0;
}
@Override
public boolean isSupervised() {
return false;
}
@Override
public long getRecordCount() {
return 0;
}
@Override
public void getRecord(long index, MLDataPair pair) {
}
@Override
public MLDataSet openAdditional() {
return null;
}
@Override
public void add(MLData data1) {
}
@Override
public void add(MLData inputData, MLData idealData) {
}
@Override
public void add(MLDataPair inputData) {
}
@Override
public void close() {
}
@Override
public int size() {
return 0;
}
@Override
public MLDataPair get(int index) {
return null;
}
```

```
}
```

The equivalent C# class is very similar.

```
public class MyEncogDataSet : IMLDataSet
{
    public void Close()
    {
        throw new NotImplementedException();
    }
    public int Count
    {
        get { throw new NotImplementedException(); }
    }
    public IEnumerator<IMLDataPair> GetEnumerator()
    {
        throw new NotImplementedException();
    }
    public int IdealSize
    {
        get { throw new NotImplementedException(); }
    }
    public int InputSize
    {
        get { throw new NotImplementedException(); }
    }
    public IMLDataSet OpenAdditional()
    {
        throw new NotImplementedException();
    }
    public bool Supervised
    {
        get { throw new NotImplementedException(); }
    }
    public IMLDataPair this[int x]
    {
        get { throw new NotImplementedException(); }
    }
}
```

As you can see, there are a number of methods that must be defined to create a dataset. You must define properties to provide the size of the input vectors and ideal results. If the training set is for unsupervised data, then the size of the ideal results will be zero. You must also define a Java Iterator or C# Enumerator for the class. The BasicMLDataSet class is a very good example of how to create a relatively simple dataset. You can see the Java version of this class here:

`https://github.com/encog/encog-java-core/blob/master/src/main/java/org/encog/ml/data/basic/BasicMLDataSet.java`

The C# version of BasicMLDataSet is here:

`https://github.com/encog/encog-dotnet-core/blob/master/encog-core-cs/ML/Data/Basic/BasicMLDataSet.cs`

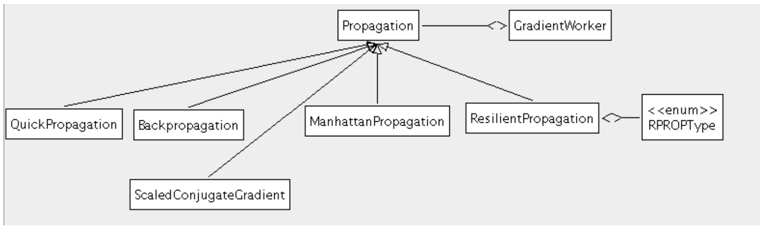
2.3 Propagation Training

Backpropagation, and related algorithms, are common a common means of training neural networks. Encog refers to these gradient-based trainers as the propagation trainers. Encog supports the following propagation trainers:

- Backpropagation
- Manhattan Propagation
- Resilient Propagation
- Scaled Conjugate Gradient

The relationships between the propagation training classes are shown in Figure 2.3.

Figure 2.3: Propagation Training

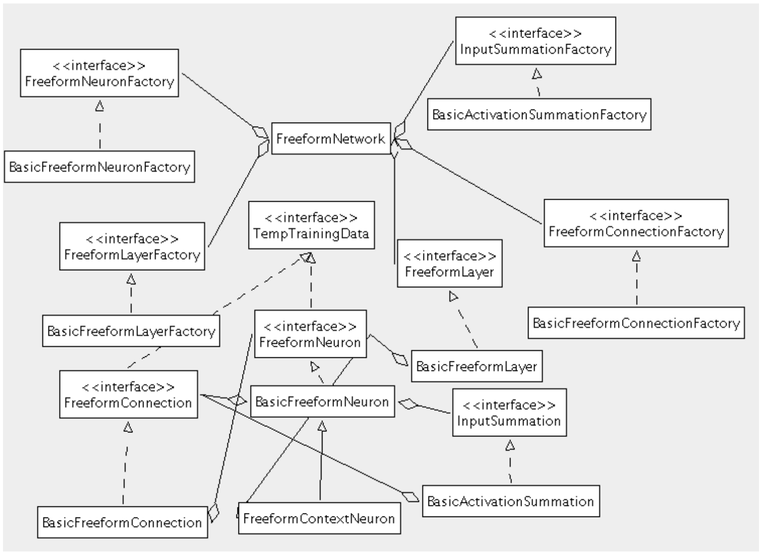


The **Propagation** base class provides the capability of calculating gradients. This is done using multi-threaded code that takes advantage of a multi-core processor. If you would like to create a training algorithm that is based on Gradients, you might consider creating a subclass of the **Propagation** class. The **Backpropagation** class provides a good example of how to do this.

2.4 Freeform Neural Networks

Freeform Neural Networks are the most flexible type of neural network offered by Encog. The other types of neural network supported Encog are flat networks and [NEAT Networks]]. Freeform networks allow neurons and layers to be configured in non-standard layout. They also allow you to use interfaces to create your own summation formulas. However, they are also somewhat slower than the more standard flat networks provided by Encog. Unless you are trying to create your own non-standard neural network topology, you should make use of the flat networks provided by Encog.

Figure 2.4: Freeform Networks


<<interface>> FreeformNeuronFactory
BasicFreeformNeuronFactory
<<interface>> InputSummationFactory
BasicActivationSummationFactory
FreeformNetwork
<<interface>> FreeformLayerFactory
<<interface>> TempTrainingData
<<interface>> FreeformLayer
<<interface>> FreeformConnectionFactory
BasicFreeformLayerFactory
<<interface>> FreeformNeuron
BasicFreeformLayer
BasicFreeformConnectionFactory
<<interface>> FreeformConnection
BasicFreeformNeuron
<<interface>> InputSummation
BasicActivationSummation
BasicFreeformConnection
FreeformContextNeuron

2.4.1 Important Interfaces for Freeform

Most of the objects used by the **FreeformNetwork** are implemented through objects. This allows you to easily create a factory and provide custom implementations of any of these objects. The primary object types (interfaces) used by the **FreeformNetwork** are listed here.

- **FreeformConnection**(created by **FreeformConnectionFactory**) - A weighted connection between two **FreeformNeurons**.
- **FreeformContextNeuron** - (created by **FreeformContextNeuronFactory**) - Descended from FreeformNeuron, represents a context neuron. Context neurons have no direct inputs and output whatever their activation is set to. Their activations are copied from the last activation of their source neuron.
- **FreeformNeuron** (created by **FreeformNeuronFactory**) - A neuron.
- **InputSummation** (created by **InputSummationFactory**) - The input summation specifies how the inputs from a neuron are combined to produce a single input to this neuron. This summation is passed through an Activation Function. Each **FreeformNeuron** has one (and only one) input summation. If there are no inbound links (i.e. context or input layer) then there will not be a summation for the specified network.

Concrete basic implementations are provided by each of the above interfaces. Default factories are also provided to create these basic implementations.

BasicActivationSummation (created by **BasicActivationSummationFactory**) - Provides a simple summation of all inputs to the neuron. This sum is passed through an activation function and then forms the activation of that neuron.

BasicFreeformConnection (created by **BasicFreeformConnectionFactory**) - Provides a basic weighted connection between two neurons.

BasicFreeformLayer (created by **BasicFreeformLayerFactory**) - Provides a layer. Layers are simply related neurons. At a minimum you must have an input and output layer. Other layers are optional.

BasicFreeformNeuron (created by **BasicFreeformNeuronFactory**) - Provides a new neuron. The factory can create both regular and context neurons.

All factories are stored on the **FreeformNetwork** class.

2.4.2 Understanding Layers

Layers are usually an important concept for neural networks. However, freeform neural networks do not require that the network follow a strict layer structure. You must have input and output layers, however, hidden nodes may be distributed without layers, if needed. If your network does fit into layers, it is advisable to use the layer helper functions to create and link your layers.

Recurrent connections are allowed, but only through context neurons. Some neural networks do make use of context-free recurrent links. NEAT Networks are a common example of a recurrent neural network that does not make use of context neurons. A context neuron receives input from its source, and then holds that input until the next time the input neuron is called. The next time output is needed from the context neuron, the exact input that the neuron received (in the previous iteration) is provided.

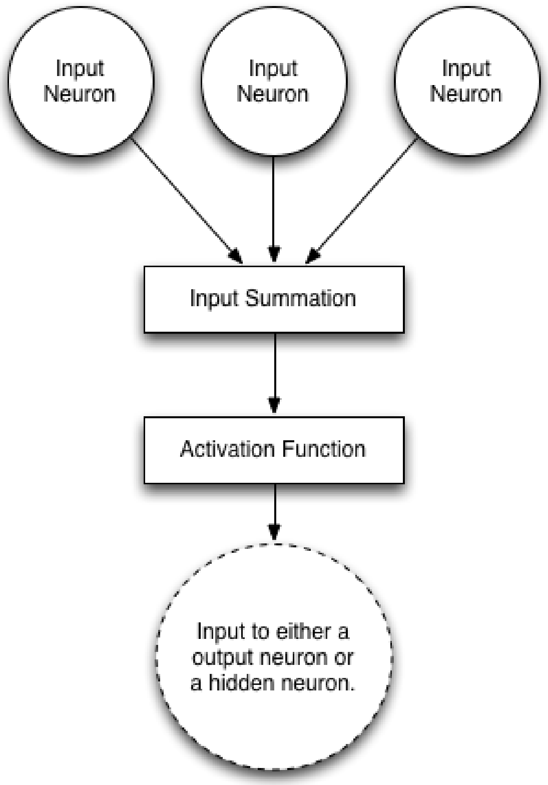
The freeform networks will likely be extended, at some point, to support context-free recurrent links.

Freeform networks cannot currently be saved as .EG files. If you wish to persist a FreeformNetwork you must use Java or C# serialization. Serialization is an effective means of persisting freeform networks as it allows user created object to be persisted as well.

2.4.3 Constructing a Freeform Network

The following diagram represents the basic structure of a freeform network. This is the basic connection in a freeform network. This diagram repeats for connections all over the freeform network. What you see here are the inputs, and single output, for a neuron, as seen in Figure 2.5.

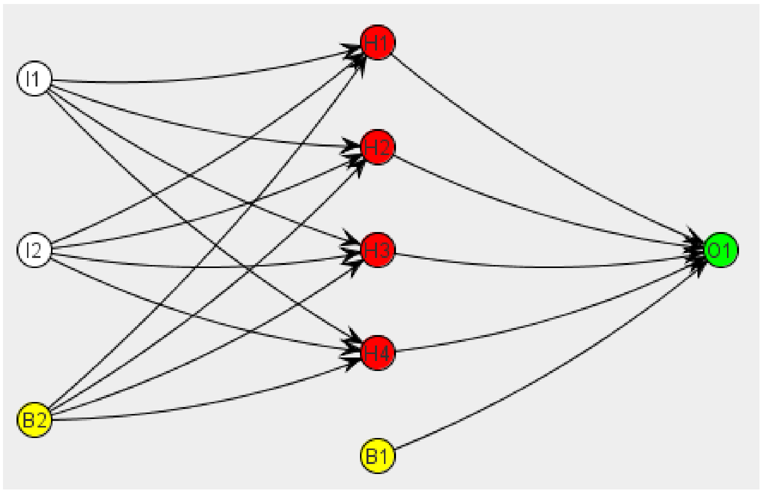
Figure 2.5: Freeform Layer Structure



The three input neurons might be actual input neurons. However, they might also be hidden neurons. Think of each of the three input neurons as taking the place of the bottom "dotted-line" neuron. This is how the network is linked together. Each neuron, that has inputs, has an input summation class. This class sums the input neurons that are connected the current neuron. This summation is then passed to an activation function. Finally, this output becomes the activation for this particular neuron. Activation functions are assigned per neuron. Additionally, activation functions process the input to a neuron, and determine the neuron's output.

The following code could be used to construct a feedforward neural network. This network would look similar to Figure 2.6.

Figure 2.6: Feedforward Feeform Network

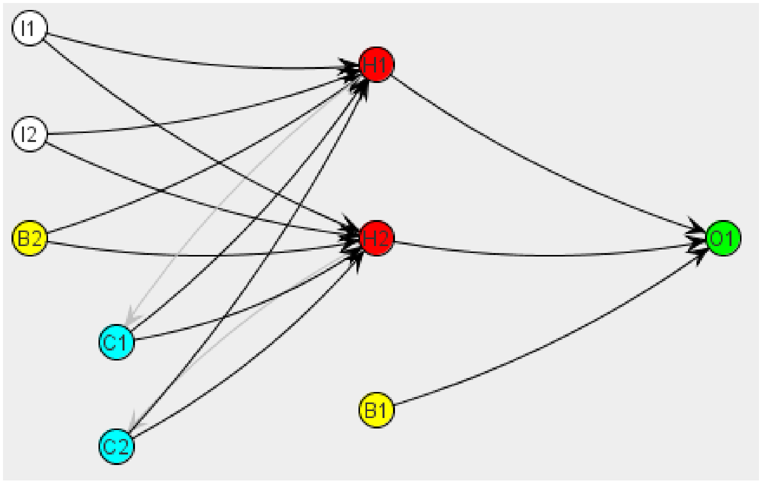


The Java code is presented here.

```
// First, create the freeform neural network
FreeformNetwork network = new FreeformNetwork();
// Create an input layer, with 2 neurons
FreeformLayer inputLayer = network.createInputLayer(2);
// Create an hidden layer, with 4 neurons
FreeformLayer hiddenLayer1 = network.createLayer(4);
// create an output layer with 1 neuron
FreeformLayer outputLayer = network.createOutputLayer(1);
// connect the input layer to the hidden layer, with a 1.0 bias
    neuron (1.0 is the usual value for a bias neuron)
network.connectLayers(inputLayer, hiddenLayer1, new
    ActivationSigmoid(), 1.0, false);
// connect the hidden layer to the output layer, with a 1.0 bias
    neuron
network.connectLayers(hiddenLayer1, outputLayer, new
    ActivationSigmoid(), 1.0, false);
// Randomize the network.
network.reset();
```

The following code could be used to construct an Elman neural network. This network would look similar to Figure 2.7.

Figure 2.7: Freeform Elman Neural Network



Chapter 3

File Formats

- Encog EGB Files
- Encog EG Files
- Encog Neural Network Files (EG)

3.1 Encog EGB Files

The Encog Binary File (EGB) is a form that normalized training data can be stored as. Encog Workbench requires that training data be stored in EGB form. Training can occur by loading the EGB file into memory, or by performing buffered reads against the EGB file for large training sets. The EGB file is stored in a binary form, yet uses the same format for Encog for Java, Encog for C# and Encog for C.

EGB files are typically created from CSV Files. This can be performed either by using Encog Workbench or using the Encog Framework. To use the framework, the EncogUtility.convertCSV2Binary method should be used in Java/C#. Other useful functions for EGB files include:

- EncogUtility.loadEGB2Memory
- EncogUtility.saveEGB

- EncogUtility.convertCSV2Binary
- Format of the EGB File

The EGB file can be thought of as one long array of double precision floating point numbers. Each of these double's is 8-bytes long. The file has a header, as follows:

- 8 bytes: ASCII "ENCOG-00" (no quotes)
- 8 Bytes: Size of input data (stored as a double, but always a whole number)
- 8 Bytes: Size of ideal data (stored as a double, but always a whole number)
- Following the header is the actual training data. Each training set element contains the following information.
- 8 x Input Size: The input data for this training set element.
- 8 x Ideal Size: The ideal data for this training set element.
- 8 Bytes: Significance. This is often just the value of 1.0, but allows the significance of a training element to be specified.

3.2 Encog EG Files

The EG File is used to hold Machine Learning Methods. Only one Machine Learning method may be placed inside of an Encog EG file. EG Files are text files, and should be readable by any text editor. The internal format of an EG file is a sort of hybrid CSV File. The format for EG files is common across all Encog platforms. Encog for Java, Encog for .Net and Encog for C all use the same format. Encog EG files are always stored in comma-decimal place form.

The type of machine learning method will determine the structure of this file. Most Encog EG files start with the following header. The only type of EG file that does not contain a header is an Encog Analyst File (EGA File). Encog Analyst files end with the extension EGA.

```
encog,BasicNetwork,java,3.1.0,1,1328887171549
```

These values represent the following.

- Column 1: Will always be ”encog”. This is the file type.
- Column 2: Is the type of EG file this is, BasicNetwork means a neural network is stored here.
- Column 3: Is the platform that created this file. This will be ”java”, ”dotnet”, ”c++”, ”silverlight” or ”cuda”.
- Column 4: Is the version of Encog that created this file.
- Column 5: Is the file version of this type of EG file.
- Column 6: Is the UNIX format time of when this file was created

EG file can be thought of as a cross between a Windows .ini File and a CSV File. Encog EG files have a hierarchy of sections and subsection. For example, the BasicNetwork file is made up of the following sections.

```
[BASIC]
[BASIC:PARAMS]
..data..
[BASIC:NETWORK]
..data..
[BASIC:ACTIVATION]
..data..
Common Parameters
```

Most Encog EG files allow parameters to be specified at the beginning of an EG file. These parameters are not used by Encog, and simply allow additional metadata to be attached to a Machine Learning Method. These parameters are simple name-value pairs. The following Neural Network has three such parameters.

```
[BASIC]
[BASIC:PARAMS]
param1=value1
```

```
param2=value1
param3=value1
[BASIC:NETWORK]
..data..
[BASIC:ACTIVATION]
..data..
```

3.2.1 Long Lines

Long lines can be problematic. EG Files for neural networks store all of the weights on one single line. This line can be megabytes long in a large network. Most parsers read an entire line at a time. A tremendously large line, from a large network, can easily cause an application to run out of memory.

```
[BASIC]
[BASIC:PARAMS]
[BASIC:NETWORK]
beginTraining=0
connectionLimit=0
contextTargetOffset=0,0,0,0
contextTargetSize=0,0,0,0
endTraining=3
hasContext=f
inputCount=248
layerCounts=1,31,101,249
layerFeedCounts=1,30,100,248
layerContextCount=0,0,0,0
layerIndex=0,1,32,133
output=0,0,0,0,0,0,0,  ...
outputCount=1
weightIndex=0,31,3061,27961
weights=##0
##double#27961
-0.661653975,-0.1598418241,-0.1579810695,0.184999618,  ...
.3469012593,-1.3712910745,0.4122231691,-0.4925843188,  ...
-0.2078151175,-0.851479728,-0.3689467346,-0.4818813258,  ...
...
0.6800220565,-1.3791134084
##end
biasActivation=0,1,1,1
[BASIC:ACTIVATION]
```

```
Notice the weights line? It has the following format.
weights=##0
##double#27961
-0.661653975,-0.1598418241,-0.1579810695,0.184999618, ...
.3469012593,-1.3712910745,0.4122231691,-0.4925843188, ...
-0.2078151175,-0.851479728,-0.3689467346,-0.4818813258, ...
...
0.6800220565,-1.3791134084
##end
```

The value after weights token specifies that this is the first (0th) long list stored in the file. The next line specifies the type, which is double. The value 27961' is the total number of weights in the file.

3.2.2 Encog Neural Network EG File

Neural Networks have a very specific form of EG File. The following shows an example of a neural network.

```
encog,BasicNetwork,java,3.1.0,1,1328887171549
[BASIC]
[BASIC:PARAMS]
param1=value1
[BASIC:NETWORK]
beginTraining=0
connectionLimit=0
contextTargetOffset=0,0,0,0
contextTargetSize=0,0,0,0
endTraining=3
hasContext=f
inputCount=248
layerCounts=1,31,101,249
layerFeedCounts=1,30,100,248
layerContextCount=0,0,0,0
layerIndex=0,1,32,133
output=0,0,0,0,0,0,0, ...
outputCount=1
weightIndex=0,31,3061,27961
weights=##0
##double#27961
-0.661653975,-0.1598418241,-0.1579810695,0.184999618, ...
.3469012593,-1.3712910745,0.4122231691,-0.4925843188, ...
```

```
−0.2078151175,−0.851479728,−0.3689467346,−0.4818813258, ...
...
0.6800220565,−1.3791134084
##end
biasActivation=0,1,1,1
[BASIC:ACTIVATION]
"ActivationTANH"
"ActivationTANH"
"ActivationTANH"
"ActivationLinear"
```

The first line is a header. This is common to all EG Files.

- BASIC:PARAMS - This section stores meta data. This information is generally not directly used by Encog. Users can add data here for various purposes. For more information, see the specification for the EG File.
- BASIC:NETWORK - The network section stores properties about the neural network. Other than Activation Functions, most neural network data is stored here. One important, and perhaps confusing, thing to remember about Encog neural networks is that the layers are stored in reverse, starting with the Output Layer moving towards the Input Layer. This is for training performance, as many training algorithms look at neural networks in this order. All indexes are zero based.
- Property beginTraining - This is the layer that training should begin at. Some neural network types are trained by multiple training algorithms. This defines the range training should be used for.
- Property connectionLimit - This is the ”connection limit”, which specifies the minimum value threshold that a weight must have for it to be considered a connection. Placing a value below this threshold is how Encog supports Sparse Networks
- Property contextTargetOffset - The context target offsets are how Encog supports Recurrent Neural Networks. Each layer in the network will have a value here. The value for each layer specifies another layer that receives a recurrent connection from this layer.

- Property contextTargetSize - This property is used in conjunction with the context target offset property. This specifies the size of the targeted layer for the recurrent connection.
- Property endTraining - This is the layer that training should begin at. Some neural network types are trained by multiple training algorithms. This defines the range training should be used for.
- Property hasContext - This value is true if this is a Recurrent Neural Network.
- Property inputCount - The number of input neurons. Bias neuron's on the Input Layer are not counted as input neurons.
- Property layerCounts
- Each layer has a value in this list. The output layer comes first. This is the total count (including Bias neurons).
- Property layerFeedCounts - Each layer has a value in this list. The output layer comes first. This is the total count (excluding Bias neurons) of neurons on this layer. Bias neurons are not ”fed” from the previous layer.
- Property layerContextCount - Each layer has a value in this list. The output layer comes first. This is the total count context neurons on this layer. Context neurons are used to create Recurrent Neural Networks.
- Property layerIndex - Each layer has a value in this list. The output layer comes first. This an index into the total neuron count for each layer. The values here are not optional, and could always be derived by Encog. They are stored here for convince, as a quick lookup.
- Property output - This array holds the last output for each neuron in the neural network. The bias neurons will be set to their bias activation.
- Property outputCount - The number of output neurons. There should be no bias on the output layer, however, if there are, these ”useless” bias neuron are not counted as input neurons.

- Property weightIndex - Each layer has a value in this list. The output layer comes first. This an index into the weights for each layer. The values here are not optional, and could always be derived by Encog. They are stored here for convince, as a quick lookup.
- Property weights - These are the weights of the neural network. Encog has a very defined flat weight ordering, defined in the weight article.
- Property bias Activation - Each layer has a value in this list. The output layer comes first. This the activation used for each layers bias neurons. If there are no bias neurons on a layer, this value is zero. This value is always generally 1 or 0.
- BASIC:ACTIVATION - These are the Activation Functions for each layer. The output layer comes first. Some activation functions have configurable parameters. These parameters are also stored on the activation function lines.

Chapter 4

Neural Network Weight Formats

- Feedforward Network Structure
- Simple Recurrent Neural Network (SRN) Structure
- Mapping to a Flat Weight Structure

Most Encog machine learning methods store their parameters as one-dimensional vectors. The parameters of a model define how a model knows what to output for a given input. The parameters are adjusted when a model is trained/fit. This allows a variety of training algorithms, such as Simulated Annealing or Nelder Mead, to train the neural network using only a loss function. Additionally, accessing this vector is typically very fast, compared to accessing a hierarchical object structure.

Most Encog neural network models store their weights, and other parameters, as a vector. Unfortunately, it can be difficult to interpret the flattened weight structure. You must understand how a neural network's weight structure is flattened to a vector.

4.1 Feedforward Network Structure

We will first look at a feed forward neural network. The example neural network has the following attributes.

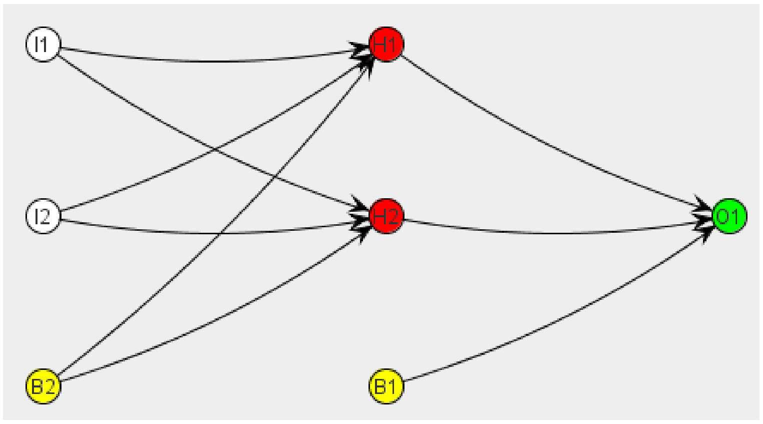
```
Input Layer: 2 neurons, 1 bias
Hidden Layer: 2 neurons, 1 bias
Output Layer: 1 neuron
This gives this network a total of 7 neurons.
```

These neurons are numbered as followed. This is the order that the **FlatNetwork.LayerOutput** property stores the network output into.

```
Neuron 0: Output 1
Neuron 1: Hidden 1
Neuron 2: Hidden 2
Neuron 3: Bias 2 (set to 1, usually)
Neuron 4: Input 1
Neuron 5: Input 2
Neuron 6: Bias 1 (set to 1, usually)
```

Figure 4.1 shows this structure graphically.

Figure 4.1: Feedforward Neural Network Structure



The flatnetwork keeps several index values, to allow it to quickly navigate the flat network. These are listed here.

```
contextTargetOffset: [0, 0, 0]
contextTargetSize: [0, 0, 0]
layerFeedCounts: [1, 2, 2]
layerCounts: [1, 3, 3]
layerIndex: [0, 1, 4]
layerOutput: [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]
weightIndex: [0, 3, 9]
```

This is the structure of the flat weights.

```
Weight 0: H1->O1
Weight 1: H2->O1
Weight 2: B2->O1
Weight 3: I1->H1
Weight 4: I2->H1
Weight 5: B1->H1
Weight 6: I1->H2
Weight 7: I2->H2
Weight 8: B1->H2
```

4.2 SRN (Elman) Structure

Next we will look at the simple recurrent neural network, in this case an Elman network. This network has the following attributes.

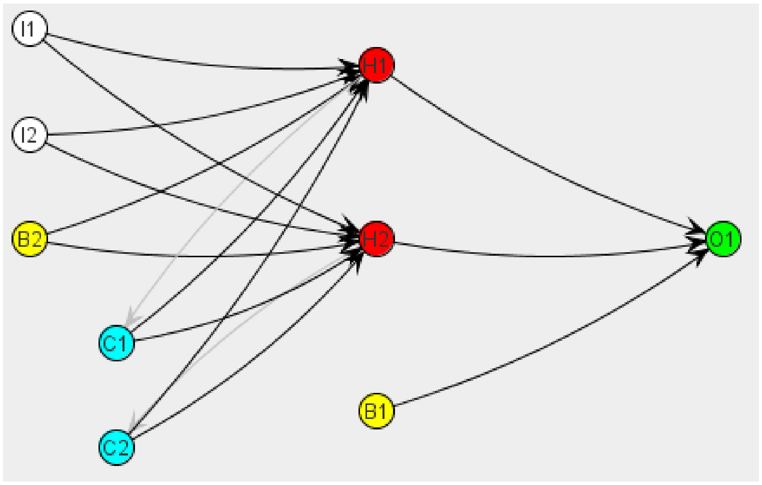
```
Input Layer: 1 neurons, 2 context, 1 bias
Hidden Layer: 2 neurons, 1 bias
Output Layer: 1 neuron
```

This gives the network a total of 8 neurons. These neurons are numbered as followed. This is the order that the FlatNetwork.LayerOutput property stores the network output into.

```
Neuron 0: Output 1
Neuron 1: Hidden 1
Neuron 2: Hidden 2
Neuron 3: Bias 2 (set to 1, usually)
Neuron 4: Input 1
Neuron 5: Bias 1 (set to 1, usually)
Neuron 6: Context 1
Neuron 7: Context 2
```

Graphically, you can see the network as follows.

Figure 4.2: Elman Neural Network Structure



The flatnetwork keeps several index values, to allow it to quickly navigate the flat network. These are listed here.

```
contextTargetOffset: [0, 6, 0]
contextTargetSize: [0, 2, 0]
layerFeedCounts: [1, 2, 1]
layerCounts: [1, 3, 4]
layerIndex: [0, 1, 4]
layerOutput: [0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0]
[0.9822812812031464, 0.5592674561520378, 0.9503503698749067, 1.0,
    0.0, 1.0, 0.9822812812031464, 0.2871920274550327]
weightIndex: [0, 3, 11]
```

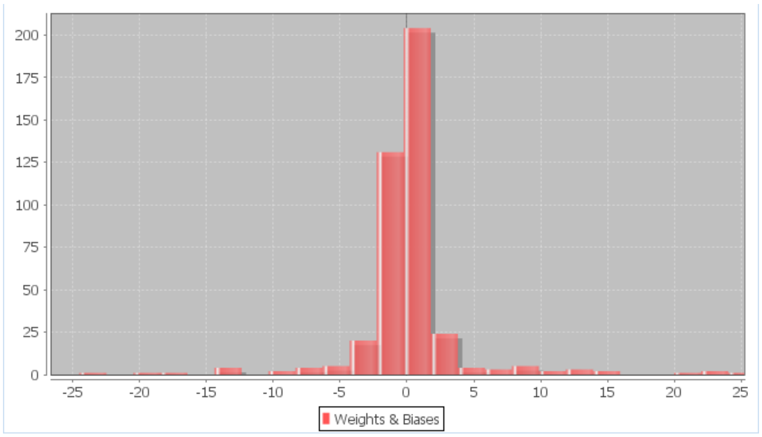
This is the structure of the flat weights.

```
Weight 0: H1->O1
Weight 1: H2->O1
Weight 2: B2->O1
Weight 3: I1->H1
Weight 4: I2->H1
Weight 5: B1->H1
Weight 6: I1->H2
Weight 7: I2->H2
Weight 8: B1->H2
```

4.3 Training

Training is the process where a neural network's weights are adjusted to produce the desired output. A trained neural network will typically have an appearance as shown here. The weights will be in a small distribution to zero. A trained neural network's weights are shown in Figure 4.3.

Figure 4.3: A Trained Network's Weights



Index