INSTITUTE FOR
ARTIFICIAL INTELLIGENCE

A Basic Guide to Encog for Neural Networks

By
Anzah H Niazi
Maulesh S Trivedi

Supervisor: Prof. Dr. Don Potter
Institute of Artificial Intelligence, University of Georgia

Introduction:

Encog is a Java framework mostly used for neural network programming. This guide is to be used in conjunction with Encog 3.2.0. It gives a basic outline to creating a simple neural network. Note that Encog's functions and classes are not limited to the ones mentioned in this guide. Though Encog contains a "Workbench" GUI structure, this guide is for building a NN using Java in Eclipse. Make sure your version of Eclipse has an IDE for Java developers.

Also note that we only use or mention a minority of the methods available in Encog. Please do look at the API documents of Encog when you are through with this guide, as they will give you more options and strategies to implement in your code. The API documents should be located in the Encog folder you downloaded. You can open them inside Eclipse or in your web browser. Look for the folder named "apidocs" in your Encog folder.

Creating a Java project in Eclipse:

- After opening Eclipse, create a project by doing the following:
  File > New > Java Project
- Type in the name of your project, e.g *MyJavaProject*. It is recommended that you check if you are running the most recent JRE (we are using JRE7).
- Click next.
- Select Libraries from the tabs above. Select Add External JARs.
- Find the folder where you have downloaded Encog 3.2.0. The .jar file you need should have the path: *….\encog-core-3.2.0\lib\encog-core-3.2.0.jar* . Select and open this. Select Finish.
- You now have a java project. Next you need to create a package and class. You can directly create both by right-clicking the *src* folder of the project and selecting *New>Class* and naming that class, e.g MyNeuralNetwork.java. A package will be created automatically. Alternately, you can individually create a package and then a class by a similar method (*New>Package* then *New>Class*).

Libraries:

We need to import Encog libraries/classes into our java class to use them. The libraries imported depend upon the way we mean to

- create our network
- import our input files
- the activation functions we mean to use
- the method used to train our network

and others. Recent versions of Eclipse will aid you in adding libraries when you use functions that they contain. Ideally you should individually import the classes you are using in your code. This might mean spending a lot of time looking up specific libraries. Usually, Eclipse will throw out an error and suggest you add the specific library to your java file, which is the easiest fix for your problem. However, if the method required is misspelled, misused or even not capitalized in the correct places, this solution will not present itself. Ultimately, the best way is to go through the API documents to identify which libraries you require.

For this guide, if you use the methods correctly as described, Eclipse should provide you with the option of adding the library, so pay attention to the way these methods are used to avoid further issues.

Other java classes we need to import for this guide are:

- *import java.io.FileNotFoundException;*
- *import java.io.PrintWriter;   //To create the output file*
- *import java.io.UnsupportedEncodingException;*

Creating the network:
Neural Networks in Encog are fairly simple to implement. The following shows a short and straightforward way of creating a simple network, although a more complex and customizable network is also possible.
First we create a BasicNetwork:

BasicNetwork myNetwork = new BasicNetwork();

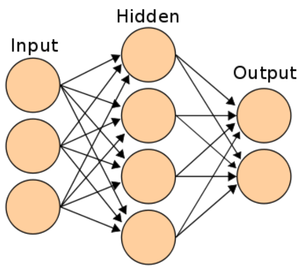
Next, we need to add layers to the network. A network may have any number of layers. The sequence of the creation of layers determines which layers are input, output or hidden. Hence, the first layer you create will be the input layer and the last one will, obviously, be the output layer. All layers created in between are hidden layers. The method of creating a layer is therefore the same for all kinds of layers.

BasicLayer(int neuronCount);

Here is an example of creating a network using BasicLayer:

```
myNetwork.addLayer(new BasicLayer(3));          // 3 nodes created for the input layer.
myNetwork.addLayer(new BasicLayer(4));          // 4 nodes created for the hidden layer.
myNetwork.addLayer(new BasicLayer(2));          // 2 nodes created for the output layer.
```

This would create a network with 3 input nodes, a single layer of 4 hidden nodes and 2 output nodes. Adding layers this way automatically creates weighted synapses between all nodes, as shown in the figure below. A synapse is an edge between two nodes of adjacent layers. As can be seen, all nodes between adjacent layers are fully connected.



A BasicLayer, by default, uses biased neurons and the hyperbolic tangent function. The hyperbolic function is used if our output can be both positive and negative. If we want different settings, we may use an overloaded constructor:

BasicLayer(activationFunction, hasBias, neuronCount)
e.g.
myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 2);

In the above example, we created 2 nodes and used the sigmoid function as the activation function, a commonly used activation function. The sigmoid function requires the output to always be positive.
We chose to use biased neurons by setting the second parameter as *true.* These are neurons with a constant output of one (though we can specify this value). If we don't wish to do this, we would change the second parameter to *false.* To learn more about the need for bias neurons, visit: http://heatonresearch.com/wiki/Bias.
Other types of layers can also be used such as the ContextLayer and the RadialBasisFunctionLayer each of which have their own features. Also note that we can use a number of activation functions in Encog such as Sigmoid, Gaussian, Bipolar, etc. Consult the API documents for more information.

The following method should be called once our network has been completely constructed:
myNetwork.getStructure().finalizeStructure();

Finally we initialize our network by randomizing the weights using the following line:
myNetwork.reset();
Note that this method can be called at any time we wish to randomize the weights of our network.

The code so far:
Your code may not look exactly like this, e.g you might have your own configuration of the basic network. However, it should have all the listed methods.

```
package MyJavaPackage;

//Libraries
import org.encog.neural.networks,BasicNetwork;
...
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;

public class MyNeuralNetwork
```

```
{
        public static void main (String[] args) throws FileNotFoundException,      UnsupportedEncoding Exception
        {
                BasicNetwork myNetwork = new BasicNetwork();        //creating the network
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 3)); //input layer
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 4)); //hidden layer
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 2)); //output layer
                myNetwork.getStructure().finalizeStructure();             // done with creating network
                myNetwork.reset();           //randomizes initial values for weights
        }
}
```

A more complex network:
If we wish to individually create our synapses, this is possible, but time consuming and more complex. However the following gives simple directions of how this can be done:

- Create a BasicNetwork:
  BasicNetwork myNetwork = new BasicNetwork();
- Create each layer such that each is identifiable:
  Layer inputLayer = new BasicLayer(3);
  Layer hiddenLayer = new BasicLayer(4);
  Layer outputLayer = new BasicLayer(2);
- To create a synapse between two layers:
  Synapse synapseInputToHidden = new WeightedSynapse(inputLayer, hiddenLayer);
  Synapse synapseHiddenToOutput = new WeightedSynapse(hiddenLayer, outputLayer);
- Add this synapse to the layer they originate from:
  inputLayer.getNext().add(synapseInputToHidden);
  hiddenLayer.getNext().add(synapsesynapseHiddenToOutput);
- Finally, we identify the input and output layers to the network:
  myNetwork.tagLayer(BasicNetwork.TAG_INPUT, inputLayer);
  myNetwork.tagLayer(BasicNetwork.TAG_OUTPUT, outputLayer);

The network we created is identical to the one we created using the simpler approach. However, this method allows the use of other kinds of synapses to be used such as Weightless, OneToOne and Direct.

Importing the Training Dataset:
Datasets can be imported in many ways. Our method creates a Machine Learning DataSet (MLDataSet) from a CSV file (as this is how our dataset is available).

MLDataSet myDataSet = TrainingSetUtil.loadCSVTOMemory(CSVFormat.ENGLISH, String path-to-CSV-File, Boolean headers, Int inputSize, Int outputSize);

Simply add the path to your input CSV file in parameter *path-to-CSV-File*. Also, make sure you use forward slashes (/) instead of backward slashes (\) in the path. For example, *“C://MyFolder/MyCSVFile.csv”.*
The *headers* parameter is Boolean. Send *true* if the first record of your data file contains the headers for your fields (*false* otherwise).
Specify the number of fields that are regarded as input in *inputSize* and the number of fields regarded as output (or the ideal outputs) in *outputSize.* Note that this assumes that your input and output fields are placed left to right in your data. Also, be sure the number of input nodes of your network matches *inputSize* and the number of output nodes matches *outputSize.*
MLDataSets are but one example of the datasets you can create in Encog. Consult the API documents for more information.

Training:
Encog includes multiple methods of training including the most popular such as Resilient propagation and Back propagation. Consult the API documents for more information.

Resilient propagation is generally regarded as the most efficient training algorithm in Encog. It requires no special parameters, other than your network and dataset, hence it is also easier to implement:

final Train myTrain = new ResilientPropagation(myNetwork, myDataSet);

Back propagation is another very popular propagation technique. It can be implemented using the following line of code:

final Train myTrain = new BackPropagation(myNetwork, myDataSet, theLearning_Rate, theMomentum);

The first two parameters we have already dealt with. The final two parameters are exclusive variables for back propagation.
The learning rate basically relates to how much the weights and thresholds of our network will change in each iteration of the training. A very high learning rate would drastically change weights and thus never resolve, and a very low learning rate would take too long to resolve. The optimum learning rate is dependent on our training set and our acceptable error rate. The value for the learning rate is a percentage given in the range 0 to 1.
The momentum prevents our network from getting stuck in a lower optima. Effectively, it “shakes” the network so it can see if there are other paths which produce lower errors. If our error rate is not decreasing according to our needs, we can increase the momentum. The momentum, like the learning rate, is also a percentage, however it can be set above 1 (though this is not recommended, and is not needed for most cases).
The selection of learning rate and momentum for our network is generally trial and error. Domain knowledge mayhaps be useful in this. Example codes provided by Encog use 0.7 and 0.3 as

learning rate and momentum, respectively. These can be used as good starting points and adjusted according to need.

You can further modify your training algorithm by adding a strategy. Strategies further modify your chosen training algorithm and each strategy works well with different networks and domains.

myTrain.addStrategy(strategy Strategy)
e.g
myTrain.addStrategy(new RequiredImprovementStrategy(theRequiredErrorRate, theNumberOfIterations));

The Required Improvement Strategy resets the weights if the error rate is not sufficiently lowered (below *theRequiredErrorRate)* after a specified number of iterations (*theNumberofIterations)*.

Running our neural network:
There are many ways to go about running a neural network. We could have it run for a fixed number of iterations or fulfill some particular criteria (usually an error threshold). Ultimately the stopping criteria for your neural network is up to you.
There are two methods that we require to run a basic neural network; running an iteration and getting the current error. Both of these are quite simple to implement:

myTrain.iteration();
This will run a single iteration for your network training. The first time this is run, it will use the initial random values for the weights (recall this is because of *myNetwork.reset()*). Each subsequent iteration will use weights as determined by your chosen training system.

double error = myTrain.getError();
Returns the mean-squared error (MSE) for the most recent iteration in double format. Note that the error is calculated before training for the next iteration.

Using these two methods, we can easily construct an algorithm using a minimum stopping criteria using a do-while loop.

```
int i = 1;
do { myTrain.iteration(); }
while (myTrain.getError() > 0.001); //The maximum MSE we want is 0.001
myTrain.finishTraining();
```

The *finishTraining()* should be evoked after our training algorithm has run.

Another way is to run it for a fixed number of iterations:

```
for (int i; i<1000; i++)
{ myTrain.iteration();}
double error = myTrain.getError();
myTrain.finishTraining();
```

Cross Validation:
Cross Validation is an effective way of making your code more efficient and can be used in combination with any training technique. Cross validation divides your dataset into a specified number of subsets called “folds”. In turn, each fold is used as a validation set while the remaining folds are used in training with your chosen training technique. This ensures that the propagation does not overfit the training data. Though increasing the number of folds will generally improve results, the computational complexity will increase as well, so a balance between efficiency and agreeable results should be sought. K-fold cross validation is implemented in Encog by the following code:

```
CrossValidationKFold trainCrossValidated =
        new CrossValidationKFold (myTrain, theNumberofFolds);
```

Cross validation in Encog requires only your existing training system and the number of folds (*k)* that you wish your dataset to have. 10 folds are generally considered a safe bet for a good result in machine learning. However, if you feel your dataset is extremely large, you might want to consider starting with a smaller number.

The code so far:

```
package MyJavaPackage;

//Libraries
import org.encog.neural.networks,BasicNetwork;
...
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;

public class MyNeuralNetwork
{
        public static void main (String[] args) throws FileNotFoundException,      UnsupportedEncoding Exception
        {
                BasicNetwork myNetwork = new BasicNetwork();        //creating the network
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 3)); //input layer
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 4)); //hidden layer
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 2)); //output layer
                myNetwork.getStructure().finalizeStructure();             // done with creating network
```

```
                    myNetwork.reset();            //randomizes initial values for weights

                    MLDataSet myDataSet = TrainingSetUtil.loadCSVTOMemory(CSVFormat.ENGLISH,
"C://MyFolder/MyCSVFile.csv, false, 3, 2);     //importing your test set;

                    final Train myTrain = new ResilientPropagation(myNetwork, myDataSet);


          //training until required maximum error (0.001) is achieved
          int i = 1;
          do { myTrain.iteration(); }
          while (myTrain.getError() > 0.001);


          }
}
```

Using our trained neural network:
Now that we have our neural network trained, we might wish to use it many ways. For example, we might want to use it for prediction on unseen data. Additionally, we might want to see it's performance with this data by computing the MSE for the neural network with new datasets. We shall now look at how we can implement these using Encog.

Let us assume that *testData* is an *MLDataSet* of new observations pertaining to our domain. We wish to obtain predictions using our 3-input, 2-output neural netwrk. We can do this using *MLDataPair.*
The inner workings of *MLDataPair* are a bit complicated. What should be understood is that it consists of a single observation, divided into input values and ideal values. Recall that our training set *myDataSet* was already defined as having 3 input fields and 2 output/ideal fields. We can assume that *testData* is organized the same way. (Indeed it is imperative that your unseen *MLDataSet* have the same number of input and output fields as the network.)
We can create an *MLDataPair* using the following method for one observation:

```
MLDataPair newPair = testData.get(obsCtr);
```

We can extract input values and ideal values from the pair using the following:

```
MLData input = newPair.getInput();   //All input values
double input1 = newPair.getInput().getData(0);  //the first input value
double input2 = newPair.getInput().getData(1); //the second input value
double input3 = newPair.getInput().getData(2); //the third input value

MLData ideal = newPair.getideal();   //All ideal values
double ideal1 = newPair.getideal().getData(0);  //the first ideal value
double ideal2 = newPair.getideal().getData(1); //the second ideal value
```

…

Note that the index for the fields/values starts with 0.

We can run this pair into our neural network and obtain the output using the following:

MLData prediction = myNetwork.compute(newPair.getInput());
double output1 = prediction.getData(0);
double output2 = prediction.getData(1);

Again, the 0 and 1 indices refer to the 1st and 2nd output values obtained by our computation.

If we wish to do this for all observations in our dataset, we can do this by implementing the code above inside the following loop:

for (MLDataPair newPair: testData)
{ ….. }

Note, however, this is not considered practical or feasible as most datasets tend to be very large. We can, if we wish to save these predictions in a file (see **Saving Data** below), but this data is usually not very useful. More important, from an analytic point of view, is the error obtained on the network using unseen data, as this gives us a good idea of its potential performance with real life data. We can obtain the MSE of a dataset run through a neural network using the following:

double theMSE = myNetwork.calculateError(testData);

We can also choose to extract all the weights from our network as a comma seperated list.

String weights = myNetwork.dumpWeights();

This is also not very useful in most scenarios. However, we may use this method to keep track of how the weights are changing through the networks iterations. It is also possible we might need the values of the weights if we wish to rebuild the network in a different interface.

Saving/reloading our trained network:
Assuming our network has been sufficiently trained, we should save our network for further use. The network should be saved with the encog extension *.EG* so it can later be reloaded.

EncogDirectoryPersistence.saveObject(newFile(String destinationFile), networkToSave);
e.g.,
EncogDirectoryPersistence.saveObject(newFile("C:\\MyFolder\MySavedNetwork.EG"), myNetwork);

To reload our network into a java file, use the following method:

BasicNetwork loadedNetwork = EncogDirectoryPersistence.loadObject(new File("C:\\MyFolder\MySavedNetwork.EG"));

The code:

```
package MyJavaPackage;

//Libraries
import org.encog.neural.networks,BasicNetwork;
...
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;

public class MyNeuralNetwork
{
        public static void main (String[] args) throws FileNotFoundException,    UnsupportedEncoding Exception
        {
                BasicNetwork myNetwork = new BasicNetwork();        //creating the network
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 3)); //input layer
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 4)); //hidden layer
                myNetwork.addLayer(new BasicLayer(new ActivationSigmoid(), true, 2)); //output layer
                myNetwork.getStructure().finalizeStructure();            // done with creating network
                myNetwork.reset();          //randomizes initial values for weights

                MLDataSet myDataSet = TrainingSetUtil.loadCSVTOMemory(CSVFormat.ENGLISH,
"C://MyFolder/MyCSVFile.csv, false, 3, 2);    //importing your test set;

                //setting up training
                final Train myTrain = new ResilientPropagation(myNetwork, myDataSet);

        //training until required maximum error (0.001) is achieved
        int i = 1;
        do { myTrain.iteration(); }
        while (myTrain.getError() > 0.001);

        //importing test dataset
        MLDataSet        testData        =        TrainingSetUtil.loadCSVTOMemory(CSVFormat.ENGLISH,
"C://MyFolder/MyTestFile.csv, false, 3, 2);

        //calculating error  using test set
        double theMSE = myNetwork.calculateError(testData);

        // saving our trained network
        EncogDirectoryPersistence.saveObject(newFile("C:\\MyFolder\MySavedNetwork.EG"), myNetwork);
        }
}
```

Conclusion

This guide is intended to provide a basic overview of Encog from an outsider’s perspective. We recommend the API Docs for Encog 3.2.0 for further reference and a detailed insight into this machine learning framework.