Name: Wimal Perera
Admission No: 09/10008

Neuroscience and Neuro Computing – Assignment

Programming Task and Reference Used

My programming task was to implement the ***backpropagation algorithm*** used in ***feed forward neural networks*** using the ***sigmoid function*** as the activation function for all perceptrons.

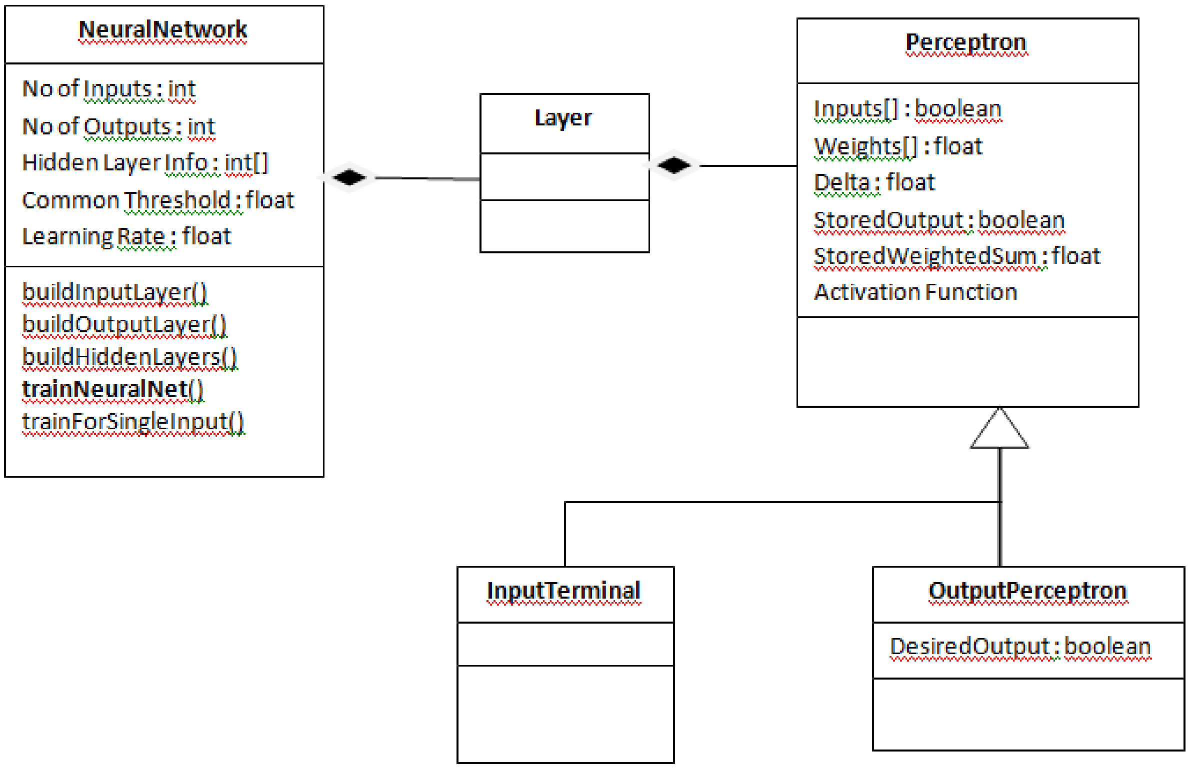
I used the following sources to implement the backpropagation algorithm.

According to Wikipedia the backpropagation algorithm for a neural network with a single hidden layer is described as follows (found at http://en.wikipedia.org/wiki/Backpropagation)

```
Initialize the weights in the network (often randomly)
  Do
         For each example e in the training set
              O = neural-net-output(network, e) ; forward pass
              T = teacher output for e
              Calculate error (T - O) at the output units
              Compute delta_wh for all weights from hidden layer to output
layer ; backward pass
              Compute delta_wi for all weights from input layer to hidden
layer ; backward pass continued
              Update the weights in the network
  Until all examples classified correctly or stopping criterion satisfied
  Return the network
```

Further I could find a graphical explanation for the backpropagation algorithm at http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html. Please note that I've attached a saved copy of the web page found in this link in the email.

Design – Class Diagram



Our **NeuralNetwork** is an array of **Layer**s (input layer, output layer and hidden layers). A Layer is an array of **Perceptron**s. We use **InputTerminal**s to represent input layer and **OutputPerceptron**s to represent output layer. Hidden layers are represented by regular Perceptrons.

Note that for the convenience in implementation, InputTerminal (used in the input layer) is considered as a special case of perceptron having a single input with weight 1 and no activation function, thus giving the same value given to input as the output.

About the Implementation

I implemented a generic neural network which can support any number of inputs, any number of hidden layers and any number of outputs (further I tested this generic neural network against the simple XOR network and the neural network with 6 inputs, 6 hidden layers and 7 outputs which I'm told to implement during this assignment).

Please note that all source code is provided in the appendix.

The initial setup of the neural network based on inputs, hidden layers and outputs is implemented in the `NeuralNetwork` class under methods `buildInputLayer` , `buildHiddenLayers` and `buildOutputLayer`.

The core backpropagation algorithm is implemented in the `NeuralNetwork` class under the method `trainForSingleInput`.

The training process is triggered in the class `NeuralNetwork` under the method `trainNeuralNet`.

How to use the Implementation – Constructing XOR

```
//parameters for creating the neural network
int inputTerminals = 2;
int outputTerminals = 1;
float learningRate = 0.1f;
float commonThreshold = 0.5f;
int[] hiddenLayerInfo = new int[] {2};

// the neural network for XOR has 2 input terminals
// in the input layer, 1 output terminal at the output
// layer and a single hidden layer with 2 perceptrons
// note that we use a common threshold of 0.5 for
// each perceptron and the learning rate is 0.1.
// further for each input in each perceptron weights
// are randomly assigned as a value between -1.0f to
// 1.0f when the neural network is initially created.
NeuralNetwork neuralNet = new NeuralNetwork(
            inputTerminals, outputTerminals,
            learningRate, commonThreshold, hiddenLayerInfo);

//example training data for the neural network
boolean[][] inputVectors = new boolean[][] {
            {false, false},
            {false, true},
            {true, false},
            {true, true}
};
boolean[][] desiredOutputVectors = new boolean[][] {
            {false},
            {true},
            {true},
            {false}
};
int iterations = 50;
String outputFile = "xor_output.txt";

//we train our neural network for 50 iterations
//the resulting weights of each perceptron in
//each layer is appended to the output file
//xor_output.txt
neuralNet.trainNeuralNet(
            outputFile, iterations,
                  inputVectors, desiredOutputVectors);
```

Sample Output for XOR

```
============================
2 inputs, 1 outputs, 1 hidden layers
Iteration 6

Layer 1 with 2 perceptrons

Perceptron 0 with 2 inputs.
Current input weights are : 0.67647254 -0.6130594
Perceptron 1 with 2 inputs.
Current input weights are : 0.75545496 -0.68519574


Layer 2 with 1 perceptrons

Perceptron 0 with 2 inputs.
Current input weights are : 0.25466794 0.27075142
```

Appendix

Test Class (Main.java)

```
import neurons.NeuralNetwork;

/**
 * This is the test class indicating how to
 * use our neural network for training
 * using the backpropagation algorithm
 * @author wimal perera (09/10008)
 *
 */
public class Main {

	public static void main(String args[]) throws Exception {

		// this is our main test-bed
		Main main = new Main();
		main.testForXOR();
		main.testForMyNetwork();
	}

	/**
	 * This method is used to demonstrate that my neural network class
	 * works for simple XOR network with a single hidden layer
	 */
	public void testForXOR() {

		//parameters for creating the neural network
		int inputTerminals = 2;
		int outputTerminals = 1;
		float learningRate = 0.1f;
		float commonThreshold = 0.5f;
		int[] hiddenLayerInfo = new int[] {2};

		// the neural network for XOR has 2 input terminals
		// in the input layer, 1 output terminal at the output
		// layer and a single hidden layer with 2 perceptrons
		// note that we use a common threshold of 0.5 for
		// each perceptron and the learning rate is 0.1.
		// further for each input in each perceptron weights
		// are randomly assigned as a value between -1.0f to
		// 1.0f when the neural network is initially created.
		NeuralNetwork neuralNet = new NeuralNetwork(
				inputTerminals, outputTerminals,
				learningRate, commonThreshold, hiddenLayerInfo);

		//example training data for the neural network
		boolean[][] inputVectors = new boolean[][] {
				{false, false},
				{false, true},
				{true, false},
```

```
                    {true, true}
        };
        boolean[][] desiredOutputVectors = new boolean[][] {
                    {false},
                    {true},
                    {true},
                    {false}
        };
        int iterations = 50;
        String outputFile = "xor_output.txt";

        //we train our neural network for 50 iterations
        //the resulting weights of each perceptron in
        //each layer is appended to the output file
        //xor_output.txt
        neuralNet.trainNeuralNet(
                    outputFile, iterations,
                    inputVectors, desiredOutputVectors);
    }

    /**
     * This method is used to demonstrate that my neural network class
     * works for the neural network which was requested for me to
     * implement during the assignment.
     */
    public void testForMyNetwork() {

        //data for creating the neural network
        int inputTerminals = 6;
        int outputTerminals = 7;
        float learningRate = 0.1f;
        float commonThreshold = 0.5f;
        int[] hiddenLayerInfo = new int[] {8, 8, 8, 8, 8, 8};

        // the neural network in the assignment has 6 input terminals
        // in the input layer, 7 output terminals at the output
        // layer and 6 hidden layers (I'm using 8 perceptrons in each)
        // note that we use a common threshold of 0.5 for
        // each perceptron and the learning rate is 0.1.
        // further for each input in each perceptron weights
        // are randomly assigned as a value between -1.0f to
        // 1.0f when the neural network is initially created.
        NeuralNetwork neuralNet = new NeuralNetwork(
                    inputTerminals, outputTerminals,
                    learningRate, commonThreshold, hiddenLayerInfo);

        //example training data for the neural network
        boolean[][] inputVectors = new boolean[][] {
                    {false, false, false, false, false, true},
                    {false, true,  true,  false, true,  false},
                    {true,  false, true,  true,  false, true},
                    {true,  true,  false, false, true,  true},
                    {true,  true,  false, true,  true,  true},
                    {true,  false, false, true,  false, true},
                    {true,  false, false, true,  false, false}
```

```
		};
		boolean[][] desiredOutputVectors = new boolean[][] {
				{false, false, false, false, false, true,  false},
				{true,  false, false, true,  false, false, true},
				{true,  false, false, true,  true,  false, true},
				{true,  false, false, false, true,  false, true},
				{true,  true,  true,  false, true,  true,  false},
				{true,  false, true,  true,  false, true,  false},
				{false, false, true,  false, true,  false, false}
		};

		int iterations = 50;
		String outputFile = "mynet_output1.txt";

		//we train our neural network for 50 iterations
		//the resulting weights of each perceptron in
		//each layer is appended to the output file
		//mynet_output1.txt
		neuralNet.trainNeuralNet(
				outputFile, iterations,
				inputVectors, desiredOutputVectors);
	}
}
```

Our Neural Network (NeuralNetwork.java)

```
package neurons;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;

import maths.ActivationFunction;
import maths.MathUtils;

/**
 * This is the core class representing the feedforward
 * neural network and the backpropagation algorithm.
 * @author wimal perera (09/10008)
 *
 */
public class NeuralNetwork {

	/**
	 * The layers in this neural network;
	 * <pre>
	 * 1. Input Layer
	 * 2. Output Layer
	 * 3. Hidden Layers
	 * </pre>
	 */
	private Layer[] layers;

	/**
```

```
 * Basic settings to set up the
 * neural network
 */
private int inputTerminalCount;
private int outputTerminalCount;
private int hiddenLayerCount;
private int[] hiddenLayerSizes;

/**
 * Basic settings helpful when training
 * the neural network
 */
private float learningRate;
private float commonThreshold;

/**
 * This is the constructor to create a neural
 * network as we wish.
 * @param inputTerminalCount
 * @param outputTerminalCount
 * @param learningRate
 * @param commonThreshold
 * @param hiddenLayerSizes
 */
public NeuralNetwork(
            int inputTerminalCount,
            int outputTerminalCount,
            float learningRate,
            float commonThreshold,
            int[] hiddenLayerSizes) {

      this.hiddenLayerSizes = hiddenLayerSizes;
      this.hiddenLayerCount = hiddenLayerSizes.length;

      //the total layers for this network is
      // hidden layer count + input layer + output layer
      this.layers = new Layer[this.hiddenLayerCount + 2];

      this.inputTerminalCount = inputTerminalCount;
      this.outputTerminalCount = outputTerminalCount;

      this.learningRate = learningRate;

      // we set up our initial neural network
      // using the below 3 methods.
      this.buildInputLayer();
      this.buildHiddenLayers();
      this.buildOutputLayer();

      this.commonThreshold = commonThreshold;
}

/**
 * This is how we build the input layer
 */
```

```
      protected void buildInputLayer() {

            //we take layer at 1st position of the array as input layer
            this.layers[0] = new Layer(0, inputTerminalCount);

            //we can think of each input terminal as a perceptron with a
single
            //input and a single output with no non-linear function
            for(int i = 0; i < inputTerminalCount; i++) {
                  this.layers[0].addPerceptron(i, new InputTerminal());

            }
      }

      /**
       * This is how we build hidden layers
       */
      protected void buildHiddenLayers() {

            //build each hidden layer and output layer
            for(int i = 0; i < hiddenLayerCount; i++) {

                  // the number of the perceptrons in the current hidden
layer
                  int currLayerSize = hiddenLayerSizes[i];

                  //we have the input layer in the 0th position in layers
array
                  this.layers[i+1] = new Layer(i+1, currLayerSize);

                  //size of previous layer
                  int prevLayerSize = this.layers[i].getPerceptronCount();

                  //set up perceptrons for this hidden layer
                  for(int j = 0; j < currLayerSize; j++) {

                        // generate a perceptron with 0.2f threshold and
sigmoid
                        // activation function
                        Perceptron currentPerceptron = new
Perceptron(prevLayerSize,
                                    ActivationFunction.SIGMOID,
this.commonThreshold);
                        // assign a random weight for each of the inputs
                        for(int k = 0; k < prevLayerSize; k++) {
                              float weight = MathUtils.getBoundedRandom(-
1.0f, 1.0f);

                              currentPerceptron.setWeight(k, weight);
                        }
                        this.layers[i+1].addPerceptron(j, currentPerceptron);

                  }
            }
      }
```

```
	/**
	 * This is how we build the output layer
	 */
	protected void buildOutputLayer() {

		//obtain the size of the hidden layer just before the output
layer
		int lastHiddenLayerSize =
this.layers[this.hiddenLayerCount].getPerceptronCount();

		//we take layer at last position of  the array as output layer
		int outputLayerIndex = (this.hiddenLayerCount + 2) - 1;
		this.layers[outputLayerIndex] = new Layer(outputLayerIndex,
outputTerminalCount);

		//an output perceptron is same as a perceptron except that it has
a
		//desired output value
		for(int i = 0; i < outputTerminalCount; i++) {

			// create perceptrons for output layer
			OutputPerceptron currentPerceptron = new
OutputPerceptron(lastHiddenLayerSize,
					ActivationFunction.SIGMOID,
this.commonThreshold);
			for(int j = 0; j < lastHiddenLayerSize; j++) {
				// assign random weights for each perceptron
				float weight = MathUtils.getBoundedRandom(-1.0f,
1.0f);
				currentPerceptron.setWeight(j, weight);
			}
			this.layers[outputLayerIndex].addPerceptron(i,
currentPerceptron);
		}
	}

	/**
	 * This is the core backpropagation algorithm.
	 * How we train our neural network with respect to a single
	 * input versus its desired output.
	 * @param inputs
	 * @param desiredOutputs
	 */
	protected void trainForSingleInput(boolean[] inputs, boolean[]
desiredOutputs) {

		if (inputs.length == inputTerminalCount
				&& desiredOutputs.length == outputTerminalCount) {

			Layer inputLayer = this.layers[0];
			Layer outputLayer = this.layers[(this.hiddenLayerCount + 2)
- 1];

			// set up input for the current training datum
			for (int i = 0; i < inputs.length; i++) {
```

```
                        InputTerminal inputTerminal =
                              (InputTerminal) inputLayer.getPerceptron(i);
                        inputTerminal.setInput(inputs[i]);
                  }

                  // set up desired output for current training datum
                  for (int j = 0; j < desiredOutputs.length; j++) {
                        // output layer is found at last in the layers array
                        OutputPerceptron outputPerceptron =
                              (OutputPerceptron)
outputLayer.getPerceptron(j);
                        outputPerceptron.setDesiredOutput(desiredOutputs[j]);
                  }

                  // calculate outputs for each perceptron from the 2nd layer
onwards
                  // this is the forward pass calculating all the outputs and
                  // storing them in each perceptron
                  // based on weighted sum and activation function and
threshold
                  // for each perceptron.
                  for(int k = 1; k < this.layers.length; k++) {
                        Layer prevLayer = this.layers[k-1];
                        Layer currLayer = this.layers[k];
                        int prevLayerSize = prevLayer.getPerceptronCount();
                        int currLayerSize = currLayer.getPerceptronCount();

                        // obtain each perceptron of the current layer
                        for(int l = 0; l < currLayerSize; l++) {
                              Perceptron currLayerPerceptron =
currLayer.getPerceptron(l);

                              //set up values for inputs of the current
perceptron
                              //by obtaining stored outputs in the previous
layer
                              for(int m = 0; m < prevLayerSize; m++) {
                                    Perceptron prevLayerPerceptron =
prevLayer.getPerceptron(m);
                                    currLayerPerceptron.setInput(m,
prevLayerPerceptron.getStoredOutput());
                              }

                              //calculate output for the current perceptron
                              //in the current layer and store the output
                              //within the perceptron for latter use
                              currLayerPerceptron.calculateOutput();
                        }
                  } //we have finished calculating outputs for each
perceptrons
                  //in each layer at this point

                  //calculate the delta values for each perceptron of the
output layer
                  //based on their (desired output - actual output)
```

```
                  for(int i = 0; i < this.outputTerminalCount; i++) {
                        OutputPerceptron outputPerceptron =
                              (OutputPerceptron)outputLayer.getPerceptron(i);
                        boolean desiredOutput =
outputPerceptron.getDesiredOutput();
                        boolean calculatedOutput =
outputPerceptron.getStoredOutput();
                        float delta =
                              MathUtils.booleanToFloat(desiredOutput) -
MathUtils.booleanToFloat(calculatedOutput);
                        outputPerceptron.setDelta(delta);
                  }

                  // now is the backward pass to calculate all delta values
                  // for perceptrons other than in the output layer
                  for(int j = this.layers.length - 2; j > 0; j--) {

                        Layer currentLayer = this.layers[j];
                        Layer nextLayer = this.layers[j+1];
                        int currLayerSize =
currentLayer.getPerceptronCount();
                        int nextLayerSize = nextLayer.getPerceptronCount();

                        //so we need to find deltas for each perceptron
                        //in the current layer
                        //based on deltas of perceptrons in next layer
                        for(int k = 0; k < currLayerSize; k++) {

                              // Obtain a perceptron for the current layer
                              Perceptron currLayerPerceptron =
currentLayer.getPerceptron(k);
                              // Calculate weighted delta value based on
perceptrons in next layer
                              float delta = 0.0f;
                              for(int l = 0; l < nextLayerSize; l++) {
                                    Perceptron nextLayerPerceptron =
nextLayer.getPerceptron(l);
                                    float kthweightOflthPerceptron =
nextLayerPerceptron.getWeight(k);
                                    float deltaOflthPerceptron =
nextLayerPerceptron.getDelta();
                                    delta += kthweightOflthPerceptron *
deltaOflthPerceptron;
                              }

                              // Assign the calculated delta value
                              currLayerPerceptron.setDelta(delta);
                        }

                  } //so we have calculated delta values for all perceptrons
                  // expect input terminals
                  // (we don't need a delta value for input terminals)

                  //the last step is to update the weights in the network
based on the
```

```
			//delta values
			for(int i = 1; i < this.layers.length; i++) {

				// obtain the current layer
				Layer currLayer = this.layers[i];
				int currLayerSize = currLayer.getPerceptronCount();

				// for each perceptron update the input weights
				for(int j = 0; j < currLayerSize; j++) {

					Perceptron currLayerPerceptron =
currLayer.getPerceptron(j);
					float currDelta =
currLayerPerceptron.getDelta();
					float diffActivationFunctionValue =
currLayerPerceptron.getDifferentialOutput();

					for(int k = 0; k <
currLayerPerceptron.getInputSize(); k++) {
						float currWeight =
currLayerPerceptron.getWeight(k);
						float currInput =

	MathUtils.booleanToFloat(currLayerPerceptron.getInput(k));

						// calculate the new weight and update
the
						// new weight in the perceptron
						float newWeight = currWeight +
							this.learningRate * currDelta *
diffActivationFunctionValue * currInput;
						currLayerPerceptron.setWeight(k,
newWeight);
					}
				}
			}
			// whooo !!!
			// finally we have finished updating the weights of the
neural network
			// with respect to this specific input

		}

	}

	/**
	 * This is a convenient method that can be used to dump the output of
	 * our neural network to an output file after the completion of
	 * each iteration.
	 * @param outputFileName
	 * @param iteration
	 * @throws Exception
	 */
	public void dumpNeuralNetToFile(String outputFileName, int iteration)
throws Exception {
```

```
            File outputFile = new File(outputFileName);
            BufferedWriter writer = new BufferedWriter(new
FileWriter(outputFile, true));

            writer.write("\r\n\r\n============================\r\n");
            writer.write(Integer.toString(this.inputTerminalCount) + "
inputs, ");
            writer.write(Integer.toString(this.outputTerminalCount) + "
outputs, ");
            writer.write(Integer.toString(this.hiddenLayerCount) + " hidden
layers \r\n");
            writer.write("Iteration " + Integer.toString(iteration) +
"\r\n\r\n");

            for(int i = 1; i < this.layers.length; i++) {

                  Layer currLayer = this.layers[i];
                  int currLayerSize = currLayer.getPerceptronCount();

                  writer.write("Layer " + Integer.toString(i) + " with " +
currLayerSize + " perceptrons\r\n\r\n");

                  for(int j = 0; j < currLayerSize; j++) {
                        Perceptron currPerceptron =
currLayer.getPerceptron(j);
                        writer.write("Perceptron " + Integer.toString(j) + "
with " + currPerceptron.getInputSize() + " inputs. \r\n");
                        writer.write("Current input weights are : ");
                        for(int k = 0; k < currPerceptron.getInputSize();
k++) {

      writer.write(Float.toString(currPerceptron.getWeight(k)) + " ");
                        }
                        writer.write("\r\n");
                  }

                  writer.write("\r\n\r\n");
            }

            writer.close();
      }

      /**
       * This is the method we should execute to trigger the training
process.
       * @param outputFile
       * @param iterations
       * @param inputVectors
       * @param desiredOutputVectors
       */
      public void trainNeuralNet(String outputFile,
                  int iterations,
                  boolean[][] inputVectors,
                  boolean[][] desiredOutputVectors) {
```

```
            try {
                  for(int i = 0; i < iterations; i++) {
                        this.trainForSingleInput(
                                    inputVectors[i % inputVectors.length],
                                    desiredOutputVectors[i %
desiredOutputVectors.length]);

                        this.dumpNeuralNetToFile(outputFile, i);
                  }
            }
            catch(Exception ex) {
                  ex.printStackTrace();
            }
      }
}
```

Single Layer in the Neural Network (Layer.java)

```
package neurons;

/**
 * This represents a single layer in the neural network
 * A neural network is an array of layers;
 * <pre>
 * 1. the input layer
 * 2. the output layer
 * 3. hidden layers
 * </pre>
 * @author wimal perera (09/10008)
 *
 */
public class Layer {

      /**
       * Our layer has an array of perceptrons
       */
      private Perceptron[] perceptrons;

      /**
       * The index of the layer with respect to the
       * neural network
       */
      private int index;

      private int perceptronCount;

      public Layer(int index, int perceptronCount) {
            this.index = index;
            this.perceptrons = new Perceptron[perceptronCount];
            this.perceptronCount = perceptronCount;
      }

      /**
```

```
	 * Method used to retrieve a perceptron from this layer
	 * @param index
	 * @return
	 */
	public Perceptron getPerceptron(int index) {
		if(index < perceptronCount)
			return this.perceptrons[index];
		else
			throw new RuntimeException(index + " is out of Range,
perceptron count for layer " + this.index + " is : " + perceptronCount);
	}

	/**
	 * Method used to add a perceptron to this layer
	 * @param index
	 * @param perceptron
	 */
	public void addPerceptron(int index, Perceptron perceptron) {
		if(index < perceptronCount)
			this.perceptrons[index] = perceptron;
		else
			throw new RuntimeException(index + " is out of Range,
perceptron count for layer " + this.index + " is : " + perceptronCount);
	}

	/**
	 * Method used to obtain the number of perceptrons (size)
	 * in this layer.
	 * @return
	 */
	public int getPerceptronCount() {
		return this.perceptronCount;
	}
}
```

A perceptron in a particular layer (Perceptron.java)

```
package neurons;

import maths.ActivationFunction;
import maths.MathUtils;

/**
 * This class represents the
 * perceptrons found in our neural network.
 * Note that for implementation purposes
 * our input terminals (i.e. input layer) are also
 * considered as a special kind of perceptrons
 * (please refer to InputTerminal class).
 * Further the perceptrons in output layer are
 * a bit different than others (please refer to the
 * OutputPerceptron class).
 * @author wimal perera (09/10008)
 *
```

```
 */
public class Perceptron {

      /**
       * Inputs and their corresponding
       * weights
       */
      private boolean[] inputs;
      private float[] weights;
      private int inputSize;

      /**
       * When we take the output we store
       * it in the perceptron itself since we
       * need it later when adjusting weights
       */
      private boolean storedOutput;
      private float storedWeightedSum;

      /**
       * This is the delta value which we calculate during
       * the backward pass of the algorithm.
       */
      private float delta;

      /**
       * Threshold, bias and activation function
       */
      private float threshold;
      private float bias;
      private ActivationFunction activationFunction =
ActivationFunction.LINEAR;

      public Perceptron(int inputSize) {
            this(inputSize, 0.0f, ActivationFunction.LINEAR, 0.0f);
      }

      public Perceptron(int inputSize,
                  ActivationFunction activationFunction,
                  float threshold) {
            this(inputSize, 0.0f, activationFunction, threshold);
      }

      public Perceptron(int inputSize,
                  float bias,
                  ActivationFunction activationFunction,
                  float threshold) {

            this.inputSize = inputSize;
            this.inputs = new boolean[inputSize];
            this.weights = new float[inputSize];

            this.bias = bias;
            this.threshold = threshold;
            this.activationFunction = activationFunction;
```

```
	}

	public void setWeight(int index, float weight) {
		if(index < inputSize)
			this.weights[index] = weight;
		else
			throw new RuntimeException(index + " is out of Input Range, 
input size is : " + inputSize);
	}

	public void setInput(int index, boolean input) {
		if(index < inputSize)
			this.inputs[index] = input;
		else
			throw new RuntimeException(index + " is out of Input Range, 
input size is : " + inputSize);
	}

	public boolean getInput(int index) {
		if(index < inputSize)
			return this.inputs[index];
		else
			throw new RuntimeException(index + " is out of Input Range, 
input size is : " + inputSize);
	}

	public float getWeight(int index) {
		if(index < inputSize)
			return this.weights[index];
		else
			throw new RuntimeException(index + " is out of Input Range, 
input size is : " + inputSize);
	}

	public float getBias() {
		return this.bias;
	}

	public void calculateOutput() {

		//obtain the weighted sum
		float sum = 0.0f;
		for(int i = 0; i < inputSize; i++) {
			sum += weights[i] * MathUtils.booleanToFloat(inputs[i]);
		}

		//we store the weighted sum in the perceptron itself
		this.storedWeightedSum = sum;

		//we have the only sigmoid activation function for now
		//except for the input terminals
		//we store the output in the perceptron itself
		if(activationFunction == ActivationFunction.SIGMOID) {
			float sigmoid = MathUtils.sigmoid(sum);
			if(sigmoid < threshold) {
```

```
                        this.storedOutput = false;
                  }
                  else {
                        this.storedOutput = true;
                  }
            }
      }

      /**
       * This method is required when re-adjusting weights
       * @return
       */
      public float getDifferentialOutput() {
            if(this.activationFunction == ActivationFunction.SIGMOID) {
                  return MathUtils.diffSigmoid(this.storedWeightedSum);
            }
            else
                  return 0;
      }

      public void setDelta(float delta) {
            this.delta = delta;
      }

      public float getDelta() {
            return delta;
      }

      protected void setStoredOutput(boolean storedOutput) {
            this.storedOutput = storedOutput;
      }

      public boolean getStoredOutput() {
            return this.storedOutput;
      }

      public int getInputSize() {
            return this.inputSize;
      }
}
```

An Input Terminal in the Input Layer (InputTerminal.java)

```
package neurons;

import maths.ActivationFunction;

/**
 * This class is used to represent the input terminals
 * for our neural network.
 *
 * Note that for convenience in implementation, I've
 * used the input terminals, as a special form of
 * perceptrons where you have one input terminal and
```

```
 * one output terminal and you always get the input
 * value as the output
 *
 * @author wimal perera (09/10008)
 *
 */
public class InputTerminal extends Perceptron {

	public InputTerminal() {
		super(1, 0.0f, ActivationFunction.LINEAR, 0.0f);
		super.setWeight(0, 1.0f);
	}

	/**
	 * We set up the input here.
	 * The stored output is the same as the input.
	 * @param input
	 */
	public void setInput(boolean input) {
		super.setStoredOutput(input);
		super.setInput(0, input);
	}

	@Override
	public void calculateOutput() {
		//we don't use this method for input terminals
	}
}
```

An Output Perceptron in the output layer (OutputPerceptron.java)

```
package neurons;

import maths.ActivationFunction;

/**
 * These perceptrons are used to represent perceptrons
 * in the output layer.
 *
 * Output perceptrons are different from others as they
 * have a desired output.
 *
 * @author wimal perera (09/10008)
 *
 */
public class OutputPerceptron extends Perceptron {

	private boolean desiredOutput;

	public OutputPerceptron(int inputSize,
			ActivationFunction activationFunction,
			float threshold) {
		super(inputSize, 0.0f, activationFunction, threshold);
	}
```

```
	public void setDesiredOutput(boolean desiredOutput) {
		this.desiredOutput = desiredOutput;
	}

	public boolean getDesiredOutput() {
		return desiredOutput;
	}



}
```

Utility Class used for Calculations (MathUtils.java)

```
package maths;

import java.util.Random;

/**
 * This class contains the utility functions
 * @author wimal perera (09/10008)
 *
 */
public class MathUtils {

	  /**
	 * Our random number generator.
	 */
    private final static Random _random = new Random();

	  /**
	 * Return a random number within the specified range
	 *
	 * @param lower range
	 * @param upper range
	 * @return a random number within the specified range
	 */
    public static synchronized float getBoundedRandom(float lower, float
upper) {
	float range = upper - lower;
	float result = _random.nextFloat() * range + lower;
	return(result);
    }

    /**
	 * This is the sigmoid function.
	 * @param x
	 * @return
	 */
    public static float sigmoid(float x) {
	return (float) (1.0f / (1.0f + Math.exp(-1.0f * x)));
    }
```

```
    /**
     * This is the differentiated sigmoid function.
     * @param x
     * @return
     */
    public static float diffSigmoid(float x) {
      return (float) (Math.exp(-1.0f * x) / Math.pow(1 + Math.exp(-1.0 * x),
2));
    }

    /**
     * A convenient method between switching from booleans to floats
     * @param value
     * @return
     */
    public static float booleanToFloat(boolean value) {
      if(value)
            return 1.0f;
      else
            return 0.0f;
    }

    /**
     * A convenient method which can be used to write a boolean array
     * to the output
     * @param array
     * @return
     */
    public static String booleanArrayToString(boolean[] array) {

      String outputString = "";
      for(int i = 0; i < array.length; i++) {
            outputString += array[i] ? "1" : "0";
      }
      return outputString;
    }
}
```

Activation Function Enum used by Perceptron.java (ActivationFunction.java)

```
package maths;

/**
 * This eneumeration is used by the perceptron class
 * to determine the activation function it should
 * use when calculating the output.
 *
 * Note that LINEAR is used only for input terminals
 * where you get the input as the output as it is.
 * @author wimal perera (09/10008)
 *
 */
public enum ActivationFunction {
      LINEAR,
      SIGMOID;
}
```