Name: Kevin Dsouza
Student Number: 93157162

Part 1a – Backpropagation Learning

The XOR problem is solved using a Neural Network with the following parameters
Setup: 2-input, 4-hidden and 1-output configuration
Initial weight range: -0.5 to +0.5
Learning rate: 0.2
Momentum: 0
Minimum error to be reached: 0.05

a) The problem is solved using binary representation. The binary sigmoid function is used as shown.

– y_j : $$y_j = f(S_j)$$

– where $$S_j = \sum_j w_{ji} x_i$$

– and $$f(x) = \frac{1}{1+e^{-x}}$$

$$f'(S_j) = y_j(1-y_j)$$

Source: class slides

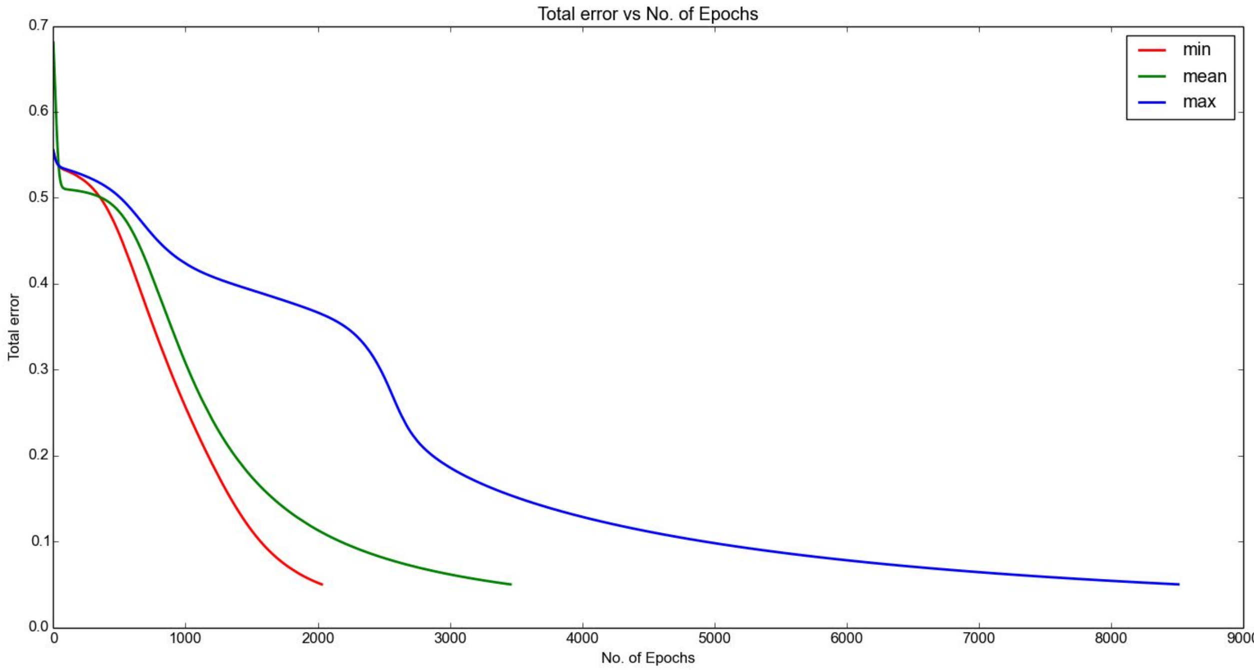
The error function used is

$$E = \sum_p E^p = \frac{1}{2} \sum_p (y^p - C^p)^2$$

Source: class slides

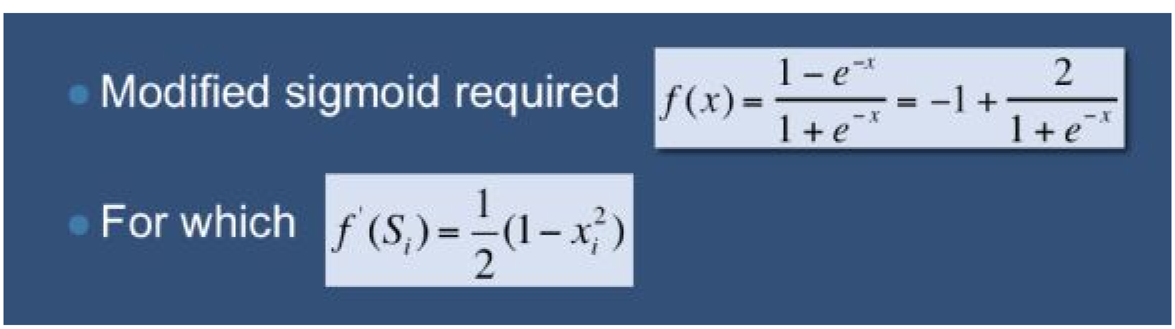
1000 independent runs are conducted. Among these the graphs for the runs of minimum, mean (closest to the mean) and maximum number of epochs are shown below.

Graphs:



Number of epochs taken:
Min - 2032
Mean - 3460.694
Max - 8512

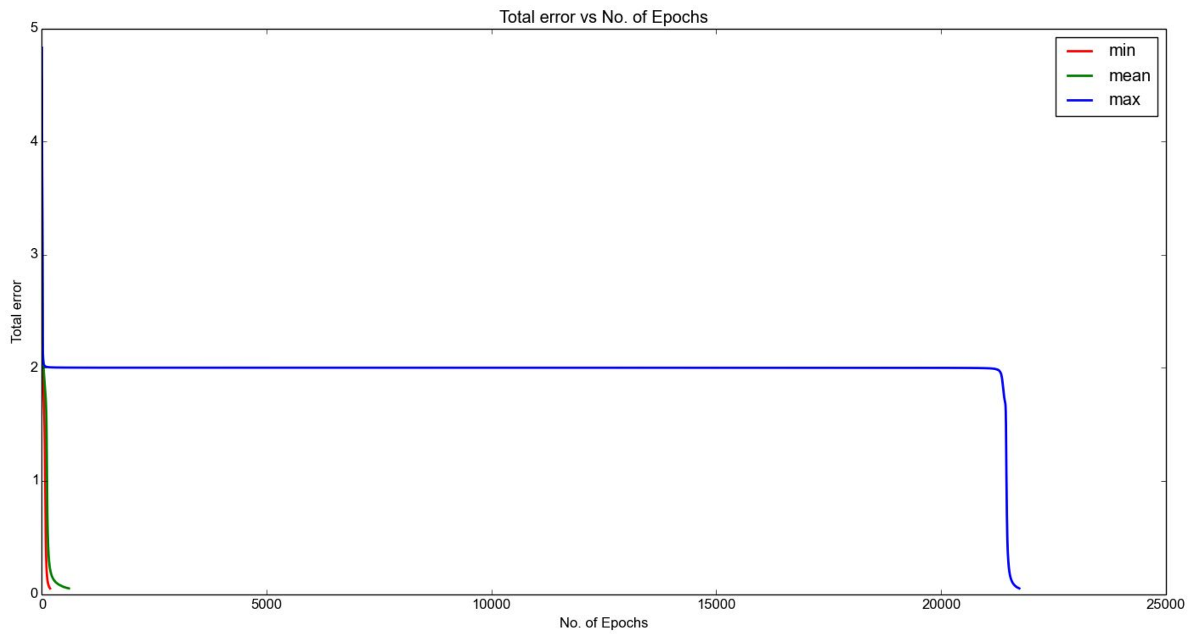
b) Now the bipolar representation of the sigmoid is used. The bipolar sigmoid function is used as shown. Error function is same as in the previous case.



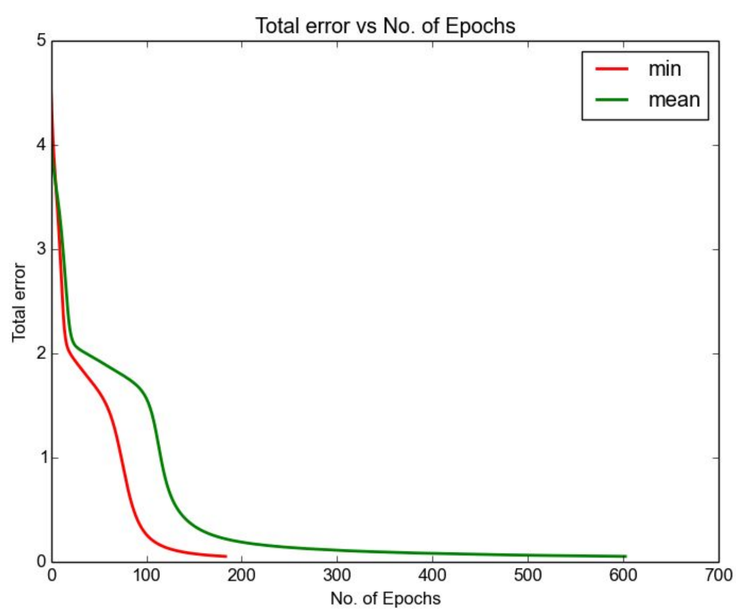
Source: class slides

Again 1000 independent runs are conducted. Because the performance of the worst case is poor when compared to the best and the average case, these runs are conducted 2 times to confirm the results. It is also noted that more than **97% of the time**, less than 1000 epochs are taken to converge.

1st time



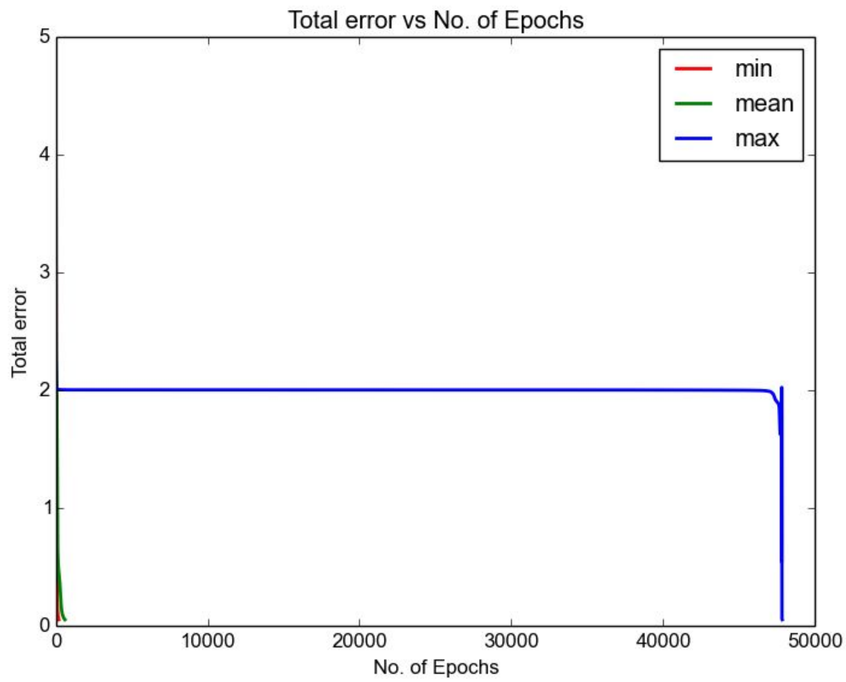
With max run included



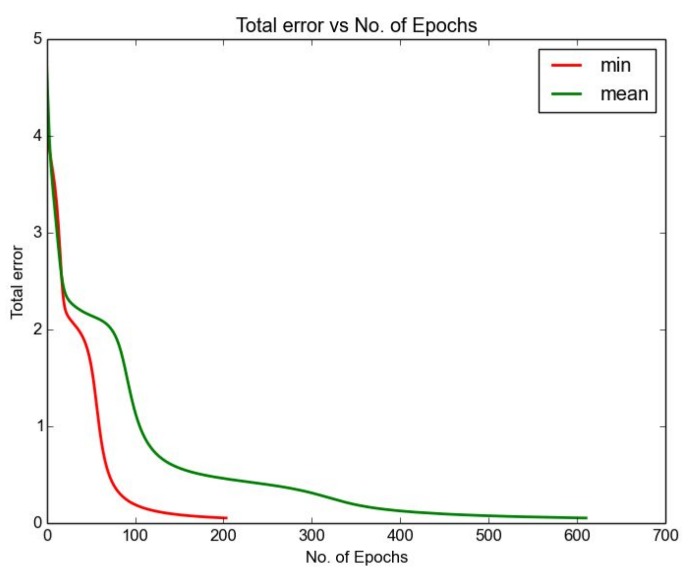
With max run excluded

Number of epochs taken:
Min - 184
Mean - 603.66
Max - 21743

2nd time



With max run included

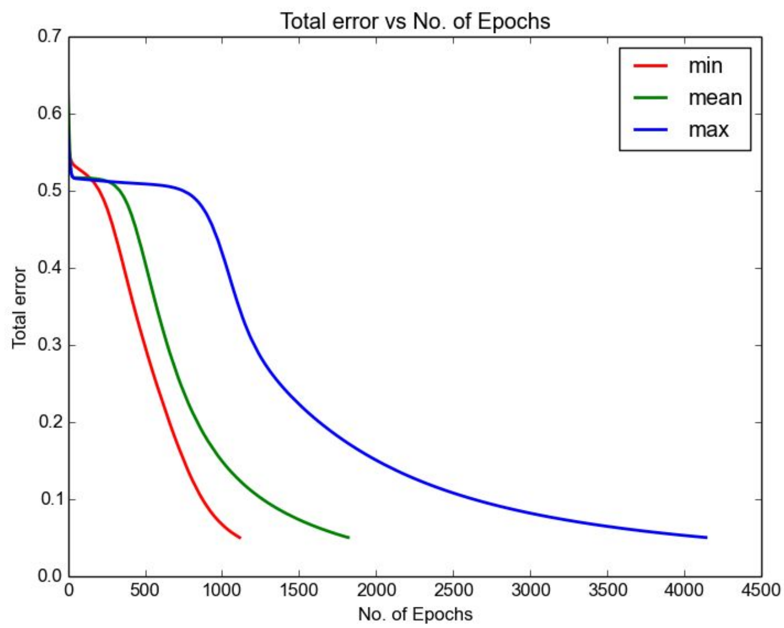


With max run excluded

Number of epochs taken:
Min - 204
Mean - 610.21
Max - 47841

c) Now the **momentum** is set to 0.9.

First the **binary representation** is used for comparison.



Number of epochs taken:
Min - 1115
Mean - 1819.242
Max - 4140

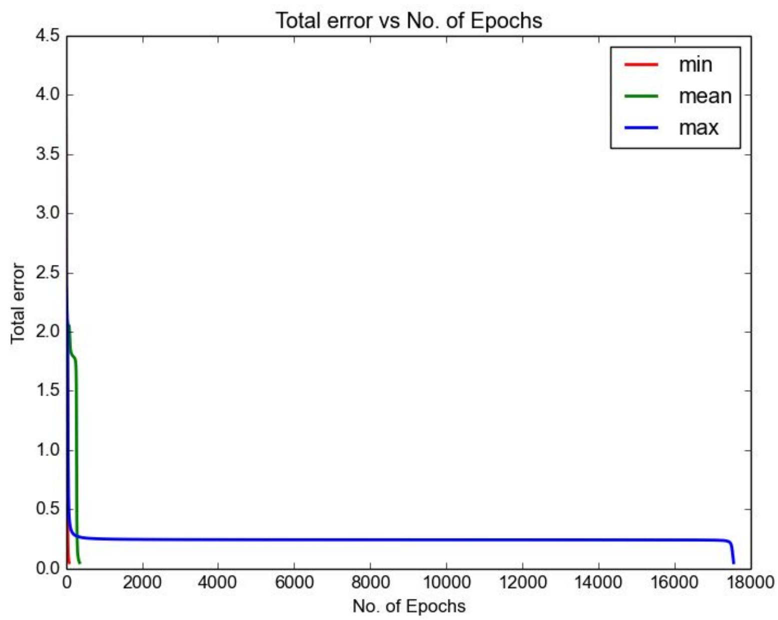
All of these values are lesser than the corresponding binary representation values for momentum equal to 0. The minimum error can be reached as fast as 1115 epochs.

Min - 2032
Mean - 3460.694
Max - 8512
Corresponding values for momentum = 0

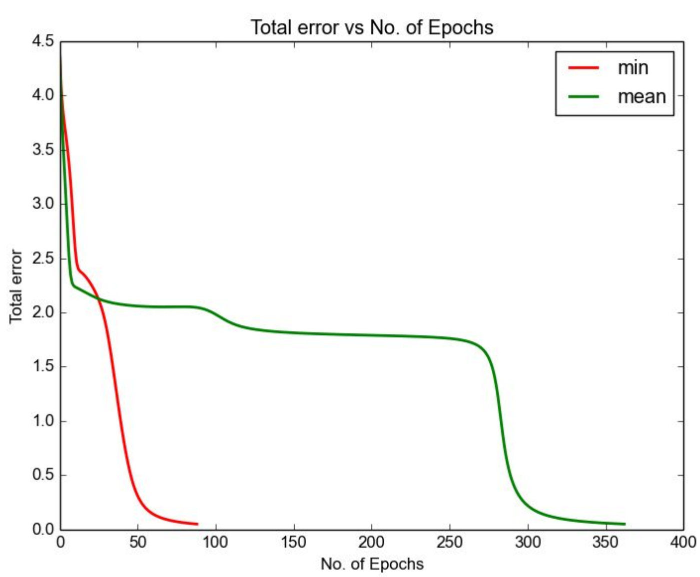
Now the **bipolar representation** is used for comparison.

Because the performance of the worst case is poor when compared to the best and the average case, these runs are conducted 2 times to confirm the results. It is also noted that more than **95% of the time**, less than 1000 epochs are taken to converge.

1st time



With max run included



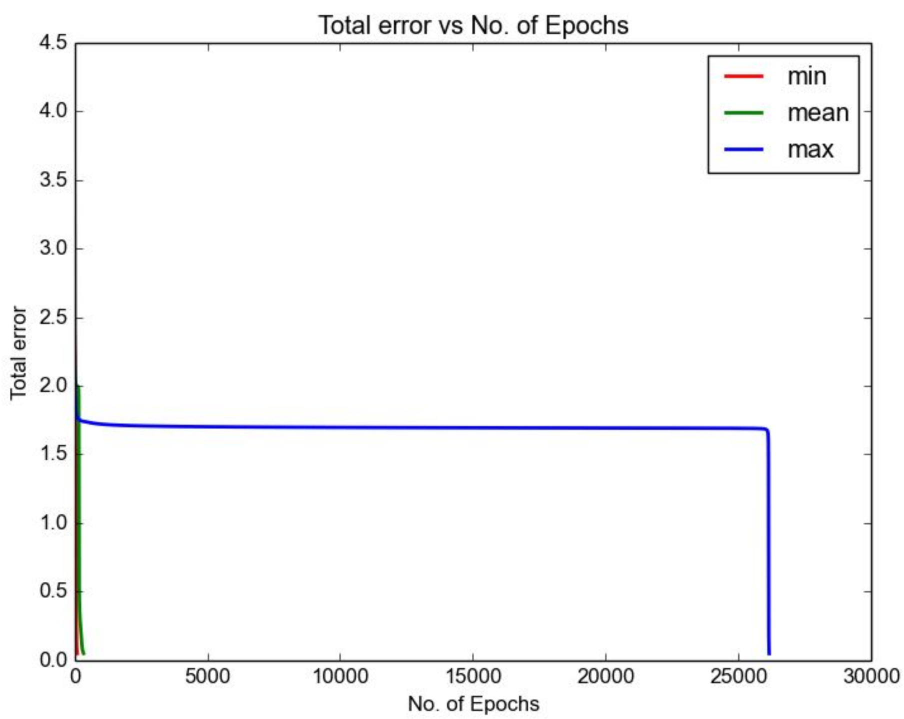
With max run excluded

Number of epochs taken:
Min - 89
Mean - 362.986
Max - 17545

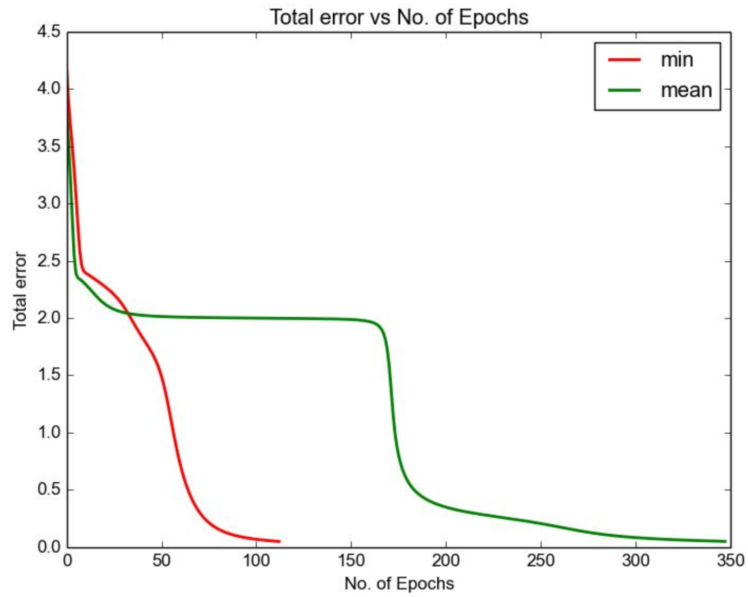
All of these values are lesser than the corresponding bipolar representation values for momentum equal to 0. The minimum error can be reached as fast as 89 epochs.

Min - 184
Mean - 603.66
Max - 21743
Corresponding values for momentum = 0

2nd time



With max run included



With max run excluded

Number of epochs taken:
Min - 113
Mean - 347.672
Max - 26155

All of these values are lesser than the corresponding bipolar representation values for momentum equal to 0. The minimum error can be reached as fast as 113 epochs.

Min - 204
Mean - 610.21
Max - 47841

Corresponding values for momentum = 0

Appendix

3 classes are implemented namely

1. NeuralNet.java
2. Neuroncell.java
3. Connect.java

The interfaces given are not used for this part of the project but will be integrated later if need be. After the error values are written to a file, Python is used to plot and compare the results (didn't find easy to use plotting libraries in java).

NeuralNet.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.text.*;
import java.util.*;

public class NeuralNet{

        //Initialize class variables
    final Random rand = new Random();
    final ArrayList<Neuroncell> inpLayer = new ArrayList<Neuroncell>();
    final ArrayList<Neuroncell> hidLayer = new ArrayList<Neuroncell>();
    final ArrayList<Neuroncell> outLayer = new ArrayList<Neuroncell>();
    final Neuroncell biasNeuron = new Neuroncell();
    final int[] numLayers;

    //Initialize Simulation parameters
    final double weightMult = 1;
    final double maxInit = 0.5f;
    final double minInit = -0.5f;
    final double learnRate = 0.2f;
    final double moment = 0.9f;
    final double posValue = 1;
    final double negValue = -1;      // set this to -1 to implement bipolar
    final boolean isBipolar = true;  // change this to true to implement bipolar

    // XOR training data
    final double inputs_XOR[][] = { { negValue, negValue }, { negValue, posValue }, { posValue,
negValue }, { posValue, posValue } };
    final double outputsIdeal[][] = { { negValue }, { posValue }, { posValue }, { negValue } };
    double outputsActual[][] = { {  }, {  }, {  }, {  } };
    double output[];

    // Main function
    public static void main(String[] args) throws IOException {
        // Change parameters before running
```

```
        DecimalFormat deciform = new DecimalFormat("#.0###");
        int nRun;
        int epochValue = 1;
        int numRuns = 2;
        int epochMax = 100000;
    double errorMIn = 0.05;

    // independent runs to average over
    for (nRun = 1; nRun <= numRuns; nRun++) {
    NeuralNet nn = new NeuralNet(2, 4, 1);
    double [] errorTrack =  nn.start(epochMax, errorMIn);

   //Get number of valid epochs
    for(int check= 0; check < errorTrack.length; check++) {
        if(errorTrack[check] == 0.0) {
                epochValue = check;
                break;
        }
    }

    // input error values to a file
    writeToFile(nRun,errorTrack,epochValue,deciform);
    }
}

//Create all neurons and connections
public NeuralNet(int input, int hidden, int output) {
    this.numLayers = new int[] { input, hidden, output };

    for (int i = 0; i < numLayers.length; i++) {
        if (i == 0) {
            for (int j = 0; j < numLayers[i]; j++) {
                Neuroncell neuron = new Neuroncell();
                inpLayer.add(neuron);
            }
        } else if (i == 1) {
            for (int j = 0; j < numLayers[i]; j++) {
                Neuroncell neuron = new Neuroncell();
                neuron.inputConnectionsAdd(inpLayer);
                neuron.biasConnectionAdd(biasNeuron);
                hidLayer.add(neuron);
            }
        } else if (i == 2) {
```

```
            for (int j = 0; j < numLayers[i]; j++) {
                Neuroncell neuron = new Neuroncell();
                neuron.inputConnectionsAdd(hidLayer);
                neuron.biasConnectionAdd(biasNeuron);
                outLayer.add(neuron);
            }
        } else {
            System.out.println("Error setting up NN, Check inputs again");
        }
    }

    // initialize random weights for all connections
    for (Neuroncell neuron : hidLayer) {
        ArrayList<Connect> connections = neuron.inputConnectionsGet();
        for (Connect conn : connections) {
            double freshWeight = randomWeightFunc();
            conn.updateWeightValue(freshWeight);
        }
    }
    for (Neuroncell neuron : outLayer) {
        ArrayList<Connect> connections = neuron.inputConnectionsGet();
        for (Connect conn : connections) {
            double freshWeight = randomWeightFunc();
            conn.updateWeightValue(freshWeight);
        }
    }

    // reinitialize the counters to 0 for next run
    Neuroncell.neurCounter = 0;
    Connect.connCounter = 0;

}

// Set inputs to input layer
public void inputSet(double inputs_XOR[]) {
    for (int i = 0; i < inpLayer.size(); i++) {
        inpLayer.get(i).setOutputValue(inputs_XOR[i]);
    }
}

// get outputs from output layer
public double[] getOutputValue() {
    double[] outputs = new double[outLayer.size()];
```

```
    for (int i = 0; i < outLayer.size(); i++)
        outputs[i] = outLayer.get(i).getOutputValue();
    return outputs;
}

// random function used to assign initial weights between [-0.5,0.5]
double randomWeightFunc() {
    return weightMult * (rand.nextDouble() * (maxInit - minInit) - minInit); // [-0.5;0.5]
}


// forward propagation function
public void forwardRipple() {
    for (Neuroncell n : hidLayer)
        n.findOutputValue(isBipolar);
    for (Neuroncell n : outLayer)
        n.findOutputValue(isBipolar);
}

// Error Back-propagation weight update function
public void errorBackpropagation(double outputExp[],boolean isBipolar) {

    int i = 0;
    for (Neuroncell n : outLayer) {
        ArrayList<Connect> connections = n.inputConnectionsGet();
        for (Connect con : connections) {
            double ak = n.getOutputValue();
            double ai = con.LNeuron.getOutputValue();
            double tempIdealOut = outputExp[i];
            double partialDescent;

            if (isBipolar) {
                    partialDescent   = - 0.5 * (1 - Math.pow(ak,2)) * ai
                        * (tempIdealOut - ak);
            } else {
            partialDescent   = -ak * (1 - ak) * ai
                    * (tempIdealOut - ak);
            }
            double delWeight = -learnRate * partialDescent;
            double freshWeight = con.getWeightValue() + delWeight;
            con.updateDeltaWeight(delWeight);
            con.updateWeightValue(freshWeight + con.getPrevDeltaWeight() * moment);
        }
```

```
        i++;
      }

      // for the hidden layer
      for (Neuroncell n : hidLayer) {
        ArrayList<Connect> connections = n.inputConnectionsGet();
        for (Connect con : connections) {
          double aj = n.getOutputValue();
          double ai = con.LNeuron.getOutputValue();
          double sumKoutputs = 0;
          int j = 0;
          for (Neuroncell out_neu : outLayer) {
            double wjk = out_neu.getConnection(n.neurId).getWeightValue(); //new updated
weight is used
            double tempIdealOut = (double) outputExp[j];
            double ak = out_neu.getOutputValue();
            j++;

            if (isBipolar) {
              sumKoutputs = sumKoutputs
                  + (-(tempIdealOut - ak) * 0.5 * (1 - Math.pow(ak,2)) * wjk);
            } else {
            sumKoutputs = sumKoutputs
                + (-(tempIdealOut - ak) * ak * (1 - ak) * wjk);
            }
          }

          double partialDescent;
          if (isBipolar) {
              partialDescent   = 0.5 * (1 - Math.pow(aj,2)) * ai * sumKoutputs;
          } else {
          partialDescent   = aj * (1 - aj) * ai * sumKoutputs;
          }
          double delWeight = -learnRate * partialDescent  ;
          double freshWeight = con.getWeightValue() + delWeight;
          con.updateDeltaWeight(delWeight);
          con.updateWeightValue(freshWeight + moment * con.getPrevDeltaWeight());
        }
      }
    }

    // Train NN until minError reached or maxSteps exceeded
    public double []  start(int maxSteps, double minError) throws IOException {
```

```
        int i;
        double error;
    error = 1;
    double [] errorTrack = new double [maxSteps];
    Arrays.fill(errorTrack, 0);
    for (i = 0; i < maxSteps && error > minError; i++) {
        error = 0;
        for (int p = 0; p < inputs_XOR.length; p++) {
            inputSet(inputs_XOR[p]);

            forwardRipple();

            output = getOutputValue();
            outputsActual[p] = output;

            for (int j = 0; j < outputsIdeal[p].length; j++) {
                double err = 0.5 * Math.pow(output[j] - outputsIdeal[p][j], 2);
                error += err;
            }

            // keep track of error
            errorTrack[i] = error;

            errorBackpropagation(outputsIdeal[p],isBipolar);
        }
    }

    nnResult();

    System.out.println("Squared sum error = " + error);
    System.out.println("EPOCH " + i+"\n");
    if (i == maxSteps)
        System.out.println("Traning taking too much time!! increase learning rate or moment");

   return errorTrack;

}


// print the result after running the epochs
void nnResult()
{
```

```
        System.out.println("NN xor training");
        for (int p = 0; p < inputs_XOR.length; p++) {
            System.out.print("XOR inputs: ");
            for (int x = 0; x < numLayers[0]; x++) {
                System.out.print(inputs_XOR[p][x] + " ");
            }

            System.out.print("| exp: ");
            for (int x = 0; x < numLayers[2]; x++) {
                System.out.print(outputsIdeal[p][x] + " ");
            }

            System.out.print("| result: ");
            for (int x = 0; x < numLayers[2]; x++) {
                System.out.print(outputsActual[p][x] + " ");
            }
            System.out.println();
        }
        System.out.println();
    }

 // write the error values to file
    public static void writeToFile(int nRun, double errorTrack[],int epochValue, DecimalFormat
deciform) throws IOException {
            int temp;

            PrintWriter writer = new PrintWriter("/media/kevin/KBD/UBC/Term 1/Learning
systems/Part 1a.2/plots/errorFileRun"+ nRun + ".txt", "UTF-8");
            for (temp=0; temp< epochValue; temp++) {
            writer.println(deciform.format(errorTrack[temp]));
            }
            writer.close();

    }
}
```

Neuroncell.java

```
import java.util.*;

public class Neuroncell {
```

```
        //initialize class variables
    static int neurCounter = 0;
    final public int neurId;
    Connect biasConnection;
    final double biasValue = -1;
    double outputValue;
    ArrayList<Connect> connectionsLeft = new ArrayList<Connect>();
    HashMap<Integer,Connect> connectHashMap = new HashMap<Integer,Connect>();

    public Neuroncell(){
        neurId = neurCounter;
        neurCounter++;
    }

    // function to add bias connection while setting up
    public void biasConnectionAdd(Neuroncell n){
        Connect con = new Connect(n,this);
        biasConnection = con;
        connectionsLeft.add(con);
    }

    // function to add connections while setting up
    public void inputConnectionsAdd(ArrayList<Neuroncell> inNeurons){
        for(Neuroncell n: inNeurons){
            Connect con = new Connect(n,this);
            connectionsLeft.add(con);
            connectHashMap.put(n.neurId, con);
        }
    }

    // function to get connection to implement backpropogation
    public Connect getConnection(int neuronIndex){
        return connectHashMap.get(neuronIndex);
    }


    //Compute Sj = Wij*Aij + w0j*bias
    public void findOutputValue(boolean isBipolar){
        double s = 0;
        for(Connect con : connectionsLeft){
            Neuroncell leftNeuron = con.getLeftNeuron();
            double wt = con.getWeightValue();
```

```
        double ac = leftNeuron.getOutputValue();

        s = s + (wt*ac);
    }
    s = s + (biasConnection.getWeightValue()*biasValue);

    if (isBipolar) {
    outputValue = bipolarSigmoid(s);
    } else {
    outputValue = sigmoid(s);
    }
}

// function to implement bipolar sigmoid
double bipolarSigmoid(double x) {
        return (1.0 - (Math.exp(-x)))/ (1.0 +  (Math.exp(-x)));
}

// function to implement regular sigmoid
double sigmoid(double x) {
    return 1.0 / (1.0 +  (Math.exp(-x)));
}

// get the ouput value from a neuron
public double getOutputValue() {
    return outputValue;
}

// set the output value of a neuron (for inputs)
public void setOutputValue(double o){
    outputValue = o;
}

// get bias value
public double getBias() {
    return biasValue;
}

// get all the connections coming from the previous layer
public ArrayList<Connect> inputConnectionsGet(){
    return connectionsLeft;
}
```

```
}
```

Connect.java

```
public class Connect {

        // initialize class variables
   double weightValue = 0;
   double delPrevWeight = 0;
   double delWeight = 0;
   final Neuroncell LNeuron;
   static int connCounter = 0;
   final public int connId;

   public Connect(Neuroncell fromN, Neuroncell toN) {
      LNeuron = fromN;
      connId = connCounter;
      connCounter++;
   }

   // update deltaweight and prevdelta weight during backprop
   public void updateDeltaWeight(double w) {
      delPrevWeight = delWeight;
      delWeight = w;
   }

   // get previous value of delta weight for momentum
   public double getPrevDeltaWeight() {
      return delPrevWeight;
   }

   // get the left side of the connection to find activation
   public Neuroncell getLeftNeuron() {
      return LNeuron;
   }

   // update weight
   public void updateWeightValue(double w) {
      weightValue = w;
   }
```

```
    // get weight to implement backprop and find activation
    public double getWeightValue() {
        return weightValue;
    }

}
```

Python code for plotting

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D


def find_nearest(array,value):
    idx = (np.abs(array-value)).argmin()
    return idx

# initialize variables and arrays
maxRun = 1000
count = 0
epochValues = []
trackErrors = []
epochs = []

# read all files to determine min, max, min of epoch values
for nRun in range(0,maxRun):
        errorfile = open("/home/kevin/Learning
systems/momentLow/Bipolar/BipolarTxt3/errorFileRun"+ str(nRun+1) +".txt", "r")
        error =  errorfile.readlines()
        epochValues.append(len(error))
        errorfile.close()

# calculate min, max, mean
epochMax = np.amax(epochValues)
index_max = np.argmax(epochValues)
epochMin = np.amin(epochValues)
index_min = np.argmin(epochValues)
epochMean = np.mean(epochValues)
```

```
index_mean = find_nearest(epochValues, epochMean)

print(epochMin)
print(epochMean)
print(epochMax)

# read these files again to plot
runNumber = [index_min, index_mean, index_max]
for i in range(0,3):
	errorfile = open("/home/kevin/Learning
systems/momentLow/Bipolar/BipolarTxt/errorFileRun"+ str(runNumber[i] + 1) +".txt", "r")
	errorTemp = errorfile.readlines()
	trackErrors.append(errorTemp)
	totalEpochs = len(trackErrors[i])
	epochs.append(np.arange(0,totalEpochs,1))

# plot the 3 curves
plt.xlabel('No. of Epochs')
plt.ylabel('Total error')
plt.title('Total error vs No. of Epochs')
lineMin, =  plt.plot(epochs[0], trackErrors[0], c='r', label="min", linewidth=2.0)
lineMean, = plt.plot(epochs[1], trackErrors[1], c='g', label="mean", linewidth=2.0)
lineMax, = plt.plot(epochs[2], trackErrors[2], c='b', label="max", linewidth=2.0)
plt.legend(handler_map={lineMin: HandlerLine2D(numpoints=4)})
plt.show()
```