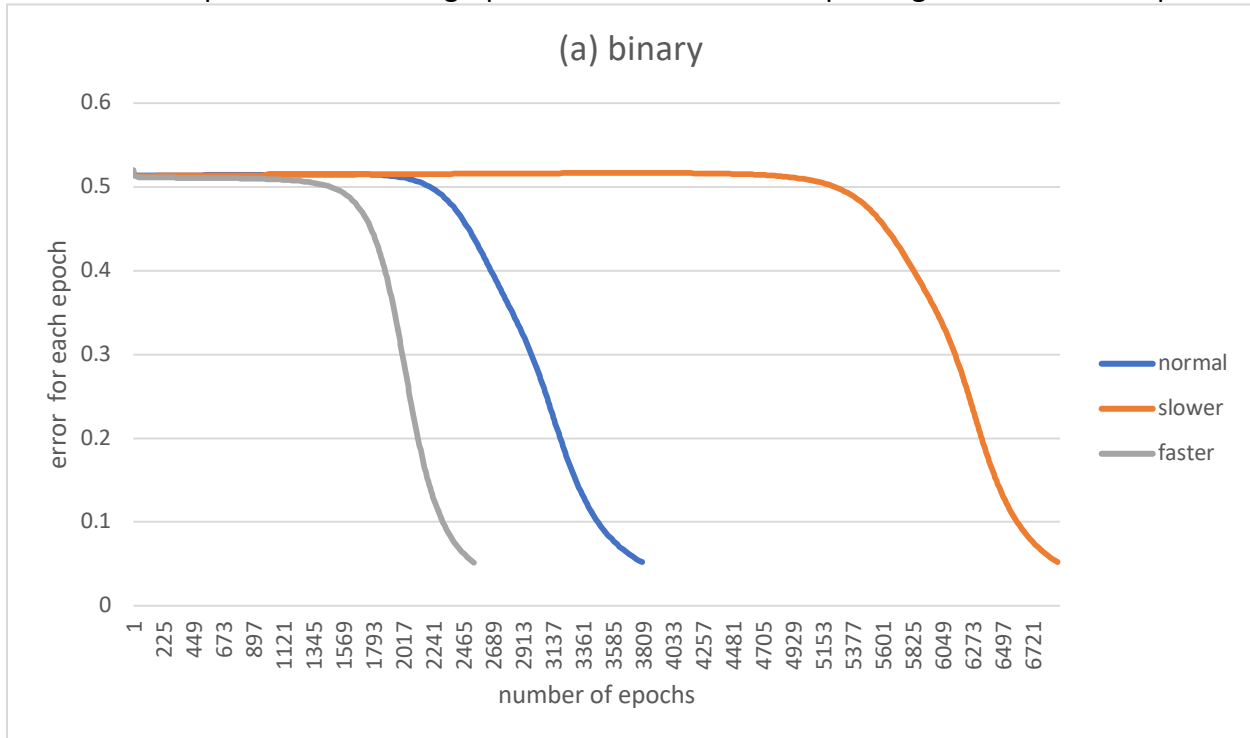


Assignment 1

(a)

I implemented the BPNetwork for binary input and output. The error goes from near 0.5 to 0.005, it changes little in the beginning, but shows a sharp decrease near the end. The normal takes 4082 to converge, while the slower one takes about 6922 epochs to converge, and the faster is 2556 epochs. Here is graph for the error for each epoch against number of epochs:



I tried 1000 trials and found the average epochs 4082. Here is the average epochs it takes to converge:

The screenshot shows the Eclipse IDE with a project named 'eclipse-workspace'. The main editor displays the code for 'XOR.java'. The code implements a neural network for XOR logic, printing epoch, error, and average epoch values. The console window at the bottom shows the output of the program, including several error values and the final state where the error is 0.0499022995103769 and the average epoch is 4082.

```

36 System.out.println("epoch="+epoch+", error="+totalError);
37 write.write(String.valueOf(totalError)+"\r\n");
38 if (totalError<ErrorBound|epoch>10000) {
39     System.out.println("error="+totalError);
40     break;
41 }
42 write.flush();
43 }
44 write.write("End trail="+trials+"\r\n");
45 avgEpoch+=epoch;
46 }
47 avgEpoch=avgEpoch/1000;
48 System.out.println("avgEpoch="+avgEpoch);
49 write.write("avgEpoch="+avgEpoch);
50 //totalError=0;
51 /*
52 if (epoch >= 10000) {
53     ++counter;
54 } else {
55     counter2 += epoch;
56     if (epoch < countermin) {

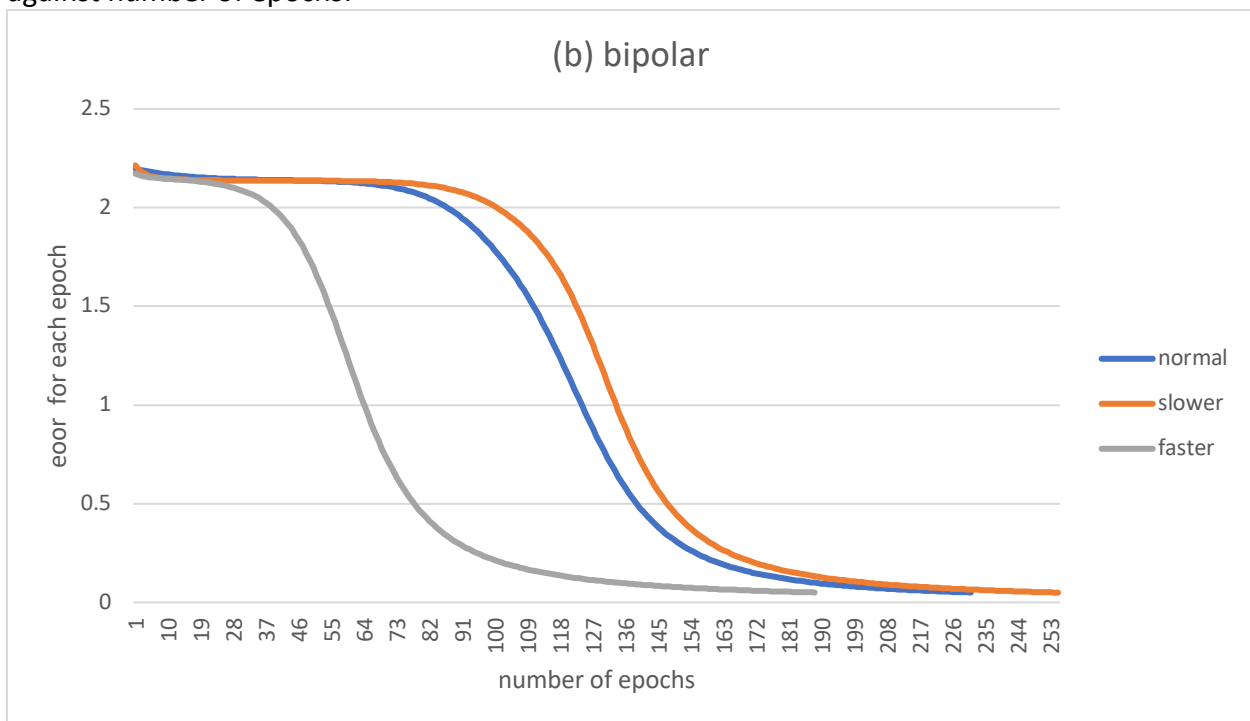
```

```

<terminated> XOR [Java Application] /Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java (Oct 25, 2018, 9:55:21 AM)
-0.0044145543726265505
-0.005370940785128924
-0.004171018609136969
-0.00399621433153015
-0.005372151438616911
epoch=4482, error=0.0499022995103769
error=0.0499022995103769
avgEpoch=4082

```

(b) The input and output are bipolar. The error goes from near 2.1 to 0.005, it has a significant decrease in the middle. The normal takes 254 to converge, while the slower takes about 255 epochs to converge, and the faster is 188 epochs. Here is graph for the error for each epoch against number of epochs:



I tried 1000 trials and found the average epochs 254. Here is the average epochs it takes to converge:

```

19  int minEpoch=10000,maxEpoch=0;
20
21  File writename=new File("output.txt");
22  boolean createNewFile =writename.createNewFile();
23  BufferedWriter write=new BufferedWriter(new FileWriter(writename));
24  int avgEpoch=0;
25  for(int trials=0;trials<1000;trials++) {
26      write.write("Start trial="+String.valueOf(trials)+"\r\n");
27      NeuralNet NN=new NeuralNet(2, 4, 0.2, 0, 1, 1);
28      NN.initializeWeights();
29      int epoch=0;
30      for(epoch=0;epoch<1000;epoch++) {
31          totalError=0;
32          for(int i=0;i<4;i++) {
33              //totalError+=NN.train(xBinary[i],yBinary[i],"binary");
34              totalError+=NN.train(xBipolar[i],yBipolar[i],"bipolar");
35          }
36          System.out.println("epoch="+epoch+", error="+totalError);
37          write.write(String.valueOf(totalError)+"\r\n");
38          if(totalError<ErrorBound||epoch>10000) {
39              System.out.println("error="+totalError);
40              break;
41          }
42      }
43  }

```

Console Output:

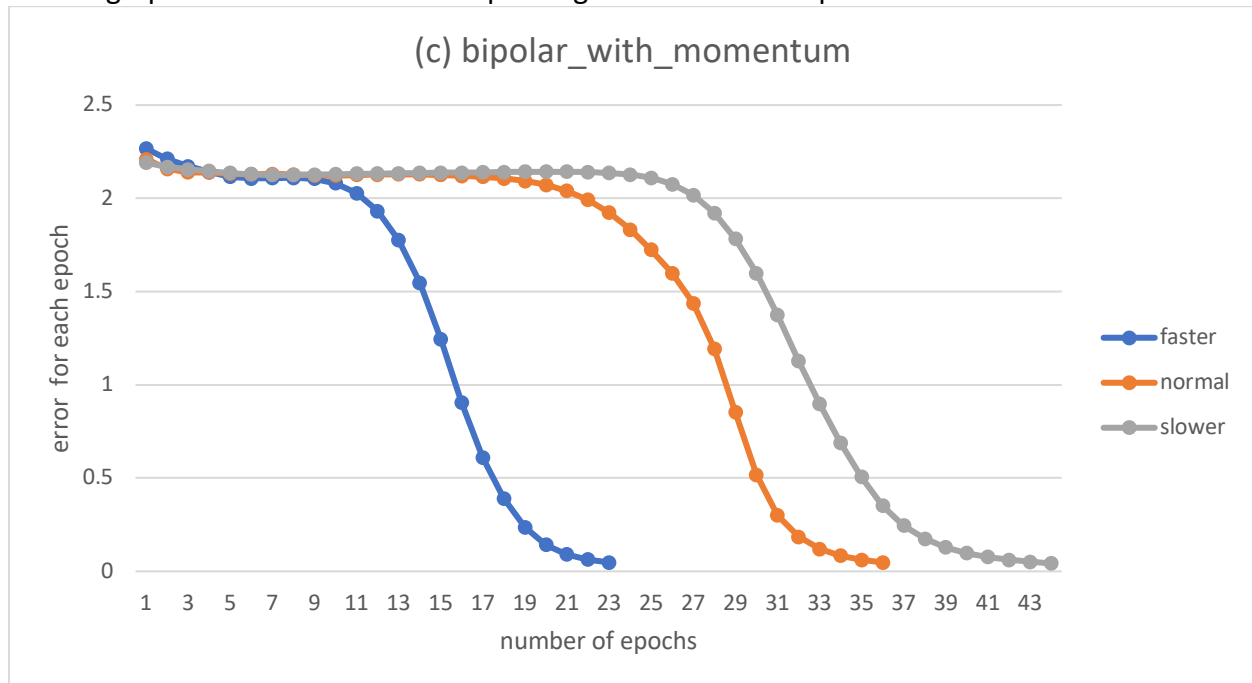
```

<terminated> XOR [Java Application] /Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java (Oct 25, 2018, 10:23:19 AM)
epoch=223, error=0.05291829159508427
epoch=224, error=0.052294683245502425
epoch=225, error=0.051684041183025276
epoch=226, error=0.05106598811229388
epoch=227, error=0.05050016066575391
epoch=228, error=0.049926208786454346
error=0.049926208786454346
avgEpoch=254

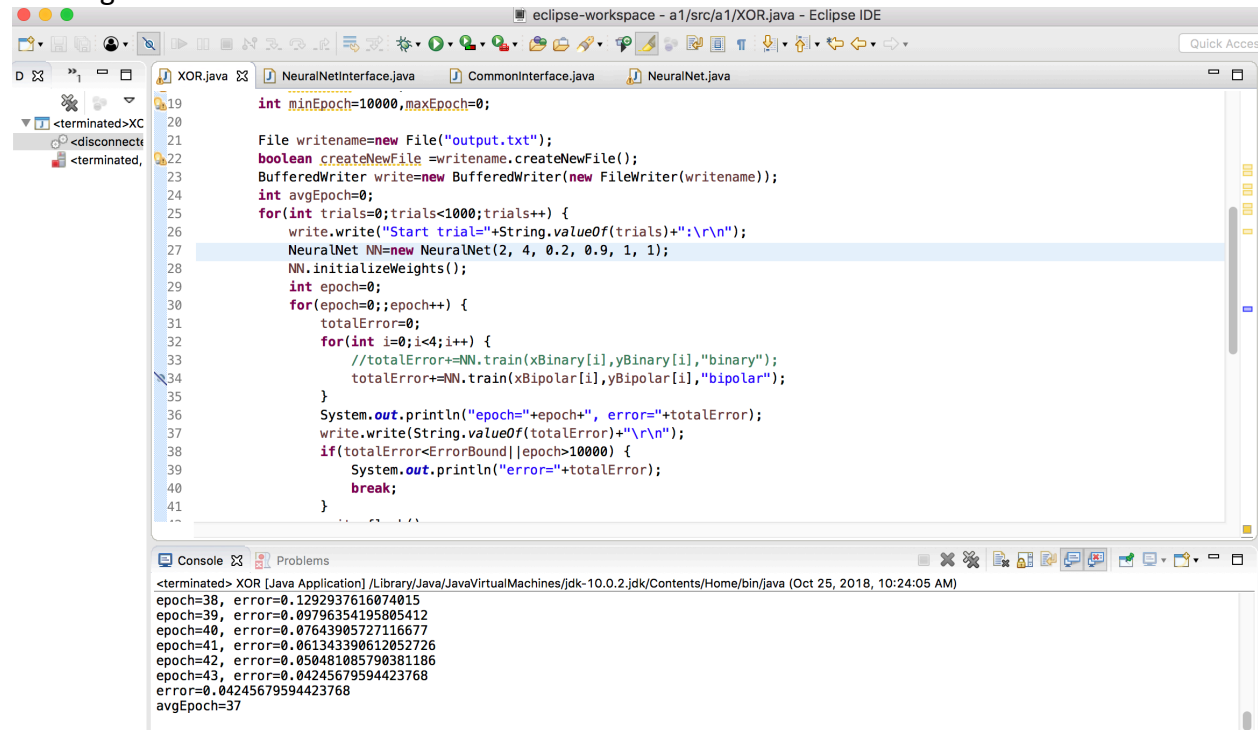
```

(c)

I added momentum into the weight update process. The error goes from near 2.2 to 0.005, it changes little in the beginning, but shows a sharp decrease near the end. The normal takes 37 to converge, while the slower takes about 44 epochs to converge, and the faster is 23 epochs. Here is graph for the error for each epoch against number of epochs:



I tried 1000 trials and found the average epochs 37. Here is the average epochs it takes to converge:



The screenshot shows the Eclipse IDE with the file `XOR.java` open. The code defines a neural network training process. The console output shows the results of 1000 trials, with the average number of epochs being 37.

```
int minEpoch=10000,maxEpoch=0;

File writename=new File("output.txt");
boolean createNewFile=writename.createNewFile();
BufferedWriter write=new BufferedWriter(new FileWriter(writename));
int avgEpoch=0;
for(int trials=0;trials<1000;trials++) {
    write.write("Start trial="+String.valueOf(trials)+"\r\n");
    NeuralNet NN=new NeuralNet(2, 4, 0.2, 0.9, 1, 1);
    NN.initializeWeights();
    int epoch=0;
    for(epoch=0;epoch<1000;epoch++) {
        totalError=0;
        for(int i=0;i<4;i++) {
            //totalError+=NN.train(xBinary[i],yBinary[i],"binary");
            totalError+=NN.train(xBipolar[i],yBipolar[i],"bipolar");
        }
        System.out.println("epoch="+epoch+", error="+totalError);
        write.write(String.valueOf(totalError)+"\r\n");
        if(totalError<ErrorBound||epoch>10000) {
            System.out.println("error="+totalError);
            break;
        }
    }
}
```

Console Output:

```
<terminated> XOR [Java Application] /Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java (Oct 25, 2018, 10:24:05 AM)
epoch=38, error=0.1292937616074015
epoch=39, error=0.09796354195805412
epoch=40, error=0.07643905727116677
epoch=41, error=0.061343390612052726
epoch=42, error=0.050481085790381186
epoch=43, error=0.04245679594423768
error=0.04245679594423768
avgEpoch=37
```

APPENDIX:

I implemented the interface given by the assignment, and I will paste the interface in the very end.

Bellow are my code, there are two classes, NeuralNet.java and XOR.java:

NeuralNet.java:

```
package a1;

import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.Random;

public class NeuralNet implements NeuralNetInterface{
    int argNumInputs;
    int argNumHidden;
    double argLearningRate;
    double argMomentumTerm;
    double argA;
    double argB;
    double[][][] weight;
    double[][][] weightChange;
    public NeuralNet (
        int argNumInputs,
        int argNumHidden,
        double argLearningRate,
        double argMomentumTerm,
```

```

double argA,
double argB ) {
this.argNumInputs=argNumInputs;
this.argNumHidden=argNumHidden;
this.argLearningRate=argLearningRate;
this.argMomentumTerm=argMomentumTerm;
this.argA=argA;
this.argB=argB;
this.weight=new double[2][[]];
this.weight[0]=new double[argNumInputs+1][argNumHidden+1];
this.weight[1]=new double[argNumHidden+1][1];
this.weightChange=new double[2][[]];
this.weightChange[0]=new double[argNumInputs+1][argNumHidden+1];
this.weightChange[1]=new double[argNumHidden+1][1];
}

public double train(double[] X, double argValue,String type) {
double []h=new double[argNumHidden+1],delta=new double[argNumHidden+1];

double yhat=0,error=0;
double[] input = Arrays.copyOf(X, X.length+1);//数组扩容
input[input.length-1]=bias;
for(int j=0;j<this.argNumHidden;j++) {
    for(int i=0;i<this.argNumInputs+1;i++) {
        h[j]+=input[i]*weight[0][i][j];
    }
    if(type=="binary")
        h[j]=sigmoid(h[j]);
    else
        h[j]=customSigmoid(h[j]);//customSigmoid
}
h[argNumHidden]=bias;
double sum=0;
for(int i=0;i<argNumHidden+1;i++) {
    sum+=h[i]*weight[1][i][0];
}
if(type=="binary") {
    yhat=sigmoid(sum);
    for(int i=0;i<argNumHidden+1;i++) {
        delta[i]=yhat*(1-yhat)*(argValue-yhat);

weight[1][i][0]+=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChange[1][i][0];

weightChange[1][i][0]=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChange[1][i][0];
        System.out.println(weightChange[1][i][0]);
    }
    for(int i=0;i<argNumInputs+1;i++) {
        for(int j=0;j<argNumHidden+1;j++) {
            weight[0][i][j]+=argLearningRate*h[j]*(1-
h[j])*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];
            weightChange[0][i][j]=argLearningRate*h[j]*(1-
h[j])*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];
        }
    }
}
else {
    yhat=customSigmoid(sum);//customSigmoid
    for(int i=0;i<argNumHidden+1;i++) {

```

```

        delta[i]=0.5*(1-Math.pow(yhat,2))*(argValue-yhat);

        weight[1][i][0]+=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChange[1][i][0];

        weightChange[1][i][0]=argLearningRate*delta[i]*h[i]+argMomentumTerm*weightChange[1][i][0];
    }
    for(int i=0;i<argNumInputs+1;i++) {
        for(int j=0;j<argNumHidden;j++) {
            weight[0][i][j]+=argLearningRate*0.5*(1-
Math.pow(h[j],2))*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];
            weightChange[0][i][j]=argLearningRate*0.5*(1-
Math.pow(h[j],2))*delta[j]*weight[1][j][0]*input[i]+argMomentumTerm*weightChange[0][i][j];
        }
    }
    }
    return 0.5*Math.pow(argValue-yhat, 2);
}

public double sigmoidBipolar(double x) {
    return 2 / (1 + Math.exp(-x))-1;
}
public double sigmoid(double x) {
    return 1 / (1 + Math.exp(-x));
}

public double customSigmoid(double x) {
    return (argB+argA) / (1 + Math.exp(-x))-argA;
}

public void initializeWeights() {
    for(int i=0;i<=this.argNumInputs;i++) {
        for(int j=0;j<=this.argNumHidden;j++) {
            weight[0][i][j]=Math.random()-0.5;
        }
    }
    for(int i=0;i<=this.argNumHidden;i++) {
        weight[1][i][0]=Math.random()-0.5;
    }
}
}

```

XOR. java:

```

package a1;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
//import org.jfree.chart.JFreeChart;

public class XOR {
    public static void main(String[] args) throws IOException{
        double ErrorBound=0.05,totalError;
        int table[][]=new int [4][2];
    }
}

```

```

int[] yBinary= {0,1,1,0};
int[] yBipolar= {-1,1,1,-1};
double []saveTotalError = null;
double[][] xBinary= {{0,0},{0,1},{1,0},{1,1}};
double[][] xBipolar= {{-1,-1},{-1,1},{1,-1},{1,1}};
int epochmin=10000;
int minEpoch=10000,maxEpoch=0;

File writename=new File("output.txt");
boolean createNewFile =writename.createNewFile();
BufferedWriter write=new BufferedWriter(new FileWriter(writename));
int avgEpoch=0;
for(int trials=0;trials<1000;trials++) {
    write.write("Start trial="+String.valueOf(trials)+"\r\n");
    NeuralNet NN=new NeuralNet(2, 4, 0.2, 0.9, 1, 1);
    NN.initializeWeights();
    int epoch=0;
    for(epoch=0;;epoch++) {
        totalError=0;
        for(int i=0;i<4;i++) {
            //totalError+=NN.train(xBinary[i],yBinary[i],"binary");

            totalError+=NN.train(xBipolar[i],yBipolar[i],"bipolar");
        }
        System.out.println("epoch="+epoch+", error="+totalError);
        write.write(String.valueOf(totalError)+"\r\n");
        if(totalError<ErrorBound || epoch>10000) {
            System.out.println("error="+totalError);

            break;
        }
        write.flush();
    }
    write.write("End trail="+trials+"\r\n");
    avgEpoch+=epoch;
}
avgEpoch=avgEpoch/1000;
System.out.println("avgEpoch="+avgEpoch);
write.write("avgEpoch="+avgEpoch);
write.flush();
write.close();
}
}

```

CommonInterface.java:

```

package a1;

import java.io.File;
/**
 * This interface is common to both the Neural Net and LUT interfaces.
 * The idea is that you should be able to easily switch the LUT
 * for the Neural Net since the interfaces are identical.
 * @date 20 June 2012
 * @author sarbjit *
 */
public interface CommonInterface {

```

```

/**
 * @param X The input vector. An array of doubles.
 * @return The value returned by the LUT or NN for this input vector */
public double outputFor(double [] X);
/**
 * This method will tell the NN or the LUT the output
 * value that should be mapped to the given input vector. I.e.
 * the desired correct output value for an input.
 * @param X The input vector
 * @param argValue The new value to learn
 * @return The error in the output for that input vector
 */
public double train(double [] X, double argValue, String type);
/**
 * A method to write either a LUT or weights of a neural net to a file.
 * @param argFile of type File.
 */
public void save(File argFile);
/**
 * Loads the LUT or neural net weights from file. The load must of course
 * have knowledge of how the data was written out by the save method.
 * You should raise an error in the case that an attempt is being
 * made to load data into an LUT or neural net whose structure does not match * the data in the file. (e.g. wrong
number of hidden neurons).
 * @throws IOException
 */
//public void load(String argFileName) throws IOException;
}

```

NeuralNetInterface.java:

```

package a1;

public interface NeuralNetInterface extends CommonInterface{
    final double bias = 1.0; // The input for each neurons bias weight
    /**
     * Constructor. (Cannot be declared in an interface, but your implementation will need one)
     * @param argNumInputs The number of inputs in your input vector
     * @param argNumHidden The number of hidden neurons in your hidden layer. Only a single hidden layer is
supported
     * @param argLearningRate The learning rate coefficient
     * @param argMomentumTerm The momentum coefficient
     * @param argA Integer lower bound of sigmoid used by the output neuron only.
     * @param argB Integer upper bound of sigmoid used by the output neuron only.
    public abstract NeuralNet (
        int argNumInputs,
        int argNumHidden,
        double argLearningRate,
        double argMomentumTerm,
        double argA,
        double argB );

    /**
     * Return a bipolar sigmoid of the input X
     * @param x The input
     * @return  $f(x) = 2 / (1 + e^{-x}) - 1$ 
     */
    public double sigmoid(double x);
    /**
     * This method implements a general sigmoid with asymptotes bounded by (a,b)

```



```

    * @param x The input
    * @return  $f(x) = \frac{b\_minus\_a}{1 + e^{-x}} - minus\_a$ 
    */
    public double customSigmoid(double x);

/**
 * Initialize the weights to random values.
 * For say 2 inputs, the input vector is [0] & [1]. We add [2] for the bias.
 * Like wise for hidden units. For say 2 hidden units which are stored in an array.
 * [0] & [1] are the hidden & [2] the bias.
 * We also initialise the last weight change arrays. This is to implement the alpha term.
 */
    public void initializeWeights();
    /**
     * Initialize the weights to 0.
     */
    public void zeroWeights();
} // End of public interface NeuralNetInterface

```