Course: CS3642 Artificial Intelligence Section W01
Student name: Aaron Cummings
Student ID: 000857610
Assignment #: 3
Due Date: May 1, 2021
Signature: 
Score:

Note: the ANN structure I used in my final implementation is 5,2,2

Architecture of ANN:

```
public class ANN {

    //outputs from the output layer [layer][neuron]
    private double[][] output;
    //the weights for each connection between layers [layer][nueon][nueon in
last layer]
    private double [][][] weights;
    //the bias for each layer (Only for the first layer, hence 4,1,2 becomes
5-1-2) [layer][neuron]
    private double [][] bias;

    //
    private double[][] errorArray;
    //contains Output(1-output) for each layer to condense algorithms.
Calculated in forward movement.
    private double[][] outputDif;

    //size of each layer in the ANN
    public final int[] NetworkLayerSizes;
    //number of inputs for the ANN
    public final int InputSize;
    //number of outputs for the ANN
    public final int OutputSize;
    //number of layers
    public final int NetworkSize;

    //Constructor
    public ANN(int... networkLayerSizes) {
        //array where the index is equivalent to the index of the layer.
Contains number of Neurons in that layer
        NetworkLayerSizes = networkLayerSizes;
        //Size of the first layer
        InputSize = networkLayerSizes[0];
        //Total Number of Layers in the network is equivalent to the length
of the array of layer sizes.
        NetworkSize = networkLayerSizes.length;
        //Size of the last Layer
        OutputSize = networkLayerSizes[NetworkSize - 1];

        output = new double[NetworkSize][];
        weights = new double[NetworkSize][][];
        bias = new double [NetworkSize][];
```

```
        errorArray = new double [NetworkSize][];
        outputDif = new double [NetworkSize][];


        //initializing the arrays used to represent the Neurons and the data
describing them
        for(int i = 0; i < NetworkSize; i++){
            output[i] = new double[NetworkLayerSizes[i]];
            errorArray[i] = new double[NetworkLayerSizes[i]];
            outputDif[i] = new double[NetworkLayerSizes[i]];
            //For this ANN there is one Bias in the Input layer, the others
are set to 0
            //i.e they don't impact the data unless this implementation is
changed
            if(i == 1) {
                bias[i] = createRandomizedWeights(NetworkLayerSizes[i], 0,
0.5);
            }else{
                bias[i] = createZerodWeights(NetworkLayerSizes[i]);
            }

            //initializing the weights of each layer
            if(i>0){
                weights[i] = createRandomizedWeights(NetworkLayerSizes[i],
                        NetworkLayerSizes[i-1], 0, 0.5);
            }
        }
    }
```

Design of Algorithm (back propagation):

```
//backpropogation algorithm to calculate the error for each layer
public void errorBackPropagation(double[] target){
    //OUTPUT LAYER
    for(int currentNeuron = 0; currentNeuron < NetworkLayerSizes[NetworkSize-
1]; currentNeuron++){
        errorArray[NetworkSize-1][currentNeuron] = (output[NetworkSize-
1][currentNeuron] - target[currentNeuron])
                * outputDif[NetworkSize-1][currentNeuron];
    }
    //ALL OTHER LAYERS EXCEPT INPUT LAYER BECAUSE IT HAS NO WEIGHTS BEHIND
IT; ie moving backward
    for(int currentLayer = NetworkSize - 2; currentLayer > 0; currentLayer--
){
        for (int currentNeuron = 0; currentNeuron <
NetworkLayerSizes[currentLayer]; currentNeuron++){
            double sum = 0;
            //going through the errors from the neurons in front of the layer
we are looking at
            for(int nextNeuron = 0; nextNeuron <
NetworkLayerSizes[currentLayer+1]; nextNeuron++){
                sum+= (weights[currentLayer+1][nextNeuron][currentNeuron] *
errorArray[currentLayer+1][nextNeuron]);
            }
            errorArray[currentLayer][currentNeuron] = sum *
outputDif[currentLayer][currentNeuron];
```

```
            //System.out.println(errorArray[currentLayer][currentNeuron]);
        }
    }
}
```

Sample output:

Testing: [-1.0, -1.0, -1.0, -1.0]
Expecting Dark
is dark

Testing: [-1.0, -1.0, -1.0, 1.0]
Expecting Dark
is dark

Testing: [-1.0, -1.0, 1.0, -1.0]
Expecting Dark
is dark

Testing: [-1.0, 1.0, -1.0, -1.0]
Expecting Dark
is dark

Testing: [1.0, -1.0, -1.0, -1.0]
Expecting Dark
is dark

Testing: [1.0, 1.0, -1.0, -1.0]
Expecting Bright
is bright

Testing: [1.0, -1.0, 1.0, -1.0]
Expecting Dark
is bright

Testing: [1.0, -1.0, -1.0, 1.0]
Expecting Bright
is bright

Testing: [-1.0, 1.0, 1.0, -1.0]

Expecting Bright
is bright

Testing: [-1.0, 1.0, -1.0, 1.0]
Expecting Bright
is bright

Testing: [-1.0, -1.0, 1.0, 1.0]
Expecting Bright
is bright

Testing: [1.0, 1.0, 1.0, 1.0]
Expecting Bright
is bright

Testing: [1.0, 1.0, 1.0, -1.0]
Expecting Bright
is bright

Testing: [1.0, 1.0, -1.0, 1.0]
Expecting Bright
is bright

Testing: [1.0, -1.0, 1.0, 1.0]
Expecting Bright
is bright

Testing: [-1.0, 1.0, 1.0, 1.0]
Expecting Bright
is bright

Completed 260 cycles of the training data.
Train again? y/n

Source Code:

```
//Aaron Cummings
//Artificial Intelligence
//ANN for Assignment 3
//NOTE: 1 is BRIGHT, -1 is DARK in the array of inputs (i.e. 4 pixels)
```

```
import java.util.*;

public class ANN {

    //outputs from the output layer [layer][neuron]
    private double[][] output;
    //the weights for each connection between layers [layer][nueon][nueon in
last layer]
    private double [][][] weights;
    //the bias for each layer (Only for the first layer, hence 4,1,2 becomes
5-1-2) [layer][neuron]
    private double [][] bias;

    //
    private double[][] errorArray;
    //contains Output(1-output) for each layer to condense algorithms.
Calculated in forward movement.
    private double[][] outputDif;

    //size of each layer in the ANN
    public final int[] NetworkLayerSizes;
    //number of inputs for the ANN
    public final int InputSize;
    //number of outputs for the ANN
    public final int OutputSize;
    //number of layers
    public final int NetworkSize;

    //Constructor
    public ANN(int... networkLayerSizes) {
        //array where the index is equivalent to the index of the layer.
Contains number of Neurons in that layer
        NetworkLayerSizes = networkLayerSizes;
        //Size of the first layer
        InputSize = networkLayerSizes[0];
        //Total Number of Layers in the network is equivalent to the length
of the array of layer sizes.
        NetworkSize = networkLayerSizes.length;
        //Size of the last Layer
        OutputSize = networkLayerSizes[NetworkSize - 1];

        output = new double[NetworkSize][];
        weights = new double[NetworkSize][][];
        bias = new double [NetworkSize][];

        errorArray = new double [NetworkSize][];
        outputDif = new double [NetworkSize][];


        //initializing the arrays used to represent the Neurons and the data
describing them
        for(int i = 0; i < NetworkSize; i++){
            output[i] = new double[NetworkLayerSizes[i]];
            errorArray[i] = new double[NetworkLayerSizes[i]];
            outputDif[i] = new double[NetworkLayerSizes[i]];
            //For this ANN there is one Bias in the Input layer, the others
are set to 0
```

```
            //i.e they don't impact the data unless this implementation is
changed
            if(i == 1) {
                bias[i] = createRandomizedWeights(NetworkLayerSizes[i], 0,
0.5);
            }else{
                bias[i] = createZerodWeights(NetworkLayerSizes[i]);
            }

            //initializing the weights of each layer
            if(i>0){
                weights[i] = createRandomizedWeights(NetworkLayerSizes[i],
                        NetworkLayerSizes[i-1], 0, 0.5);
            }
        }
    }

    //Processes input through the ANN
    public double[] processForward(double... input){

        if(input.length != InputSize) {
            return null;
        }
        output[0] = input;
        for (int currentLayer = 1; currentLayer < NetworkSize;
currentLayer++){
            for(int currentNeuron = 0; currentNeuron <
NetworkLayerSizes[currentLayer]; currentNeuron++){
                double sum;
                if (currentLayer == 1) {
                    sum = bias[currentLayer][currentNeuron];
                }else {
                    sum = 0;
                }
                for (int previousNeuron = 0; previousNeuron <
NetworkLayerSizes[currentLayer-1]; previousNeuron++){
                    sum += output[currentLayer-
1][previousNeuron]*weights[currentLayer][currentNeuron][previousNeuron];
                }
                //System.out.println("Sum "+sum);
                output[currentLayer][currentNeuron] = signmoidFunction(sum);
                //derivative of the sigmoid function
                outputDif[currentLayer][currentNeuron] =
output[currentLayer][currentNeuron]
                        *(1 - output[currentLayer][currentNeuron]);
                //System.out.println("output dif
"+outputDif[currentLayer][currentNeuron]);
            }
        }
        return output[NetworkSize - 1];
    }

    //Taking Input, processes it through the ANN
    //Calculates error through backpropagation
    //moves forward through the Network updating weights with the errors
    public void backPropTraining(double[] input, double[] target, double
learningRate){
```

```
        if(input.length != InputSize || target.length != OutputSize){
            return;
        }
        processForward(input);
        errorBackPropagation(target);
        learningFunction(learningRate);
    }

    //backpropogation algorithm to calculate the error for each layer
    public void errorBackPropagation(double[] target){
        //OUTPUT LAYER
        for(int currentNeuron = 0; currentNeuron <
NetworkLayerSizes[NetworkSize-1]; currentNeuron++){
            errorArray[NetworkSize-1][currentNeuron] = (output[NetworkSize-
1][currentNeuron] - target[currentNeuron])
                    * outputDif[NetworkSize-1][currentNeuron];
        }
        //ALL OTHER LAYERS EXCEPT INPUT LAYER BECAUSE IT HAS NO WEIGHTS
BEHIND IT; ie moving backward
        for(int currentLayer = NetworkSize - 2; currentLayer > 0;
currentLayer-- ){
            for (int currentNeuron = 0; currentNeuron <
NetworkLayerSizes[currentLayer]; currentNeuron++){
                double sum = 0;
                //going through the errors from the neurons in front of the
layer we are looking at
                for(int nextNeuron = 0; nextNeuron <
NetworkLayerSizes[currentLayer+1]; nextNeuron++){
                    sum+= (weights[currentLayer+1][nextNeuron][currentNeuron]
* errorArray[currentLayer+1][nextNeuron]);
                }
                errorArray[currentLayer][currentNeuron] = sum *
outputDif[currentLayer][currentNeuron];

//System.out.println(errorArray[currentLayer][currentNeuron]);
            }
        }
    }

    //Starts at the first hidden layer because input has no layer behind it.
    //Note the back propogation is not done in this function, just updating
the weights
    public void learningFunction(double learningRate){
        for (int currentLayer = 1; currentLayer < NetworkSize;
currentLayer++){
            for(int currentNeuron = 0; currentNeuron <
NetworkLayerSizes[currentLayer]; currentNeuron++){
                for (int previousNeuron = 0; previousNeuron <
NetworkLayerSizes[currentLayer-1]; previousNeuron++){
                    //weights[currentLayer][currentNueron][previousNeuron]
                    double deltaWeight = -learningRate * output[currentLayer-
1][previousNeuron]
                            * errorArray[currentLayer][currentNeuron];
                    weights[currentLayer][currentNeuron][previousNeuron] +=
deltaWeight;
                    //System.out.println("delta weight   "+deltaWeight);
                }
```

```
                //only bias is in first layer
                if(currentLayer == 1) {
                    double deltaBias = -learningRate *
errorArray[currentLayer][currentNeuron];
                    bias[currentLayer][currentNeuron] += deltaBias;
                    //System.out.println("deltabias  "+deltaBias);
                }
            }
        }
    }

    //Applies a sigmoid function to a value
    public double signmoidFunction(double x){
        return (1d / (1 +Math.exp(-x)));
    }

    //creates a double[] array of random values
    public static double[] createRandomizedWeights(int size, double
lowerBound, double upperBound){
        if(size < 1){
            return null;
        }
        double[] array = new double[size];
        for(int i = 0; i < size; i++){
            array[i] = randomDouble(lowerBound, upperBound);
        }
        return array;
    }

    //creates a double[] array of zero values
    public static double[] createZerodWeights(int size){
        if(size < 1){
            return null;
        }
        double[] array = new double[size];
        for(int i = 0; i < size; i++){
            array[i] = 0;
        }
        return array;
    }

    //Creates a double[][] array of random values
    public static double[][] createRandomizedWeights(int sizeX, int sizeY,
double lowerBound, double upperBound) {
        if(sizeX < 1 || sizeY < 1){
            return null;
        }
        double[][] array = new double[sizeX][sizeY];
        for(int i =0; i <sizeX; i++){
            array[i] = createRandomizedWeights(sizeY, lowerBound,
upperBound);
        }
        return array;
    }

    //returns a random double within the bounds
    public static double randomDouble(double lowerBound, double upperBound){
```

```
        return Math.random()*(upperBound-lowerBound) + lowerBound;
    }

    //Prints the sizes of the layers in the network
    public void printNetworkSizes(){
        System.out.println(NetworkLayerSizes[0]);
        System.out.println(NetworkLayerSizes[1]);
        System.out.println(NetworkLayerSizes[2]);
    }

    //Prints the Current State of the network
    public void printNetwork(){
        System.out.println("PRINTING NETWORK");
        System.out.println("Bias neuron value: "+bias[1][0]);
        for (int currentLayer = 1; currentLayer < NetworkSize;
currentLayer++){
            for(int currentNeuron = 0; currentNeuron <
NetworkLayerSizes[currentLayer]; currentNeuron++){
                for (int previousNeuron = 0; previousNeuron <
NetworkLayerSizes[currentLayer-1]; previousNeuron++){
                    System.out.println("layer "+(currentLayer-1)+" Neuron
"+previousNeuron+": "
                            +output[currentLayer-1][previousNeuron]+
                            "  weight:
"+weights[currentLayer][currentNeuron][previousNeuron]);
                }
            }
        }
    }

    //MAIN FOR TESTING ANN
    //NOTE: 1 is BRIGHT, -1 is DARK in the array of inputs (i.e. 4 pixels)
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        double learningRate = 0.3;
        //ANNs will have one bias in the input layer. hence this is 5,2,2
        ANN network = new ANN(4,2,2);
        boolean train = true;
        int sum = 0;
        while (train) {
            //Trains ANN on ALL 16 samples available
            for (int i = 0; i < 10; i++) {
                double[] input1 = new double[]{-1.0, -1.0, -1.0, -1.0};
                double[] target1 = new double[]{0.45, 0.55};

                network.backPropTraining(input1, target1, learningRate);
                //network.printNetwork();

                double[] input2 = new double[]{-1.0, -1.0, -1.0, 1.0};
                double[] target2 = new double[]{0.45, 0.55};

                network.backPropTraining(input2, target2, learningRate);
                //network.printNetwork();

                double[] input3 = new double[]{-1.0, -1.0, 1.0, -1.0};
                double[] target3 = new double[]{0.45, 0.55};
```

```
                network.backPropTraining(input3, target3, learningRate);
                //network.printNetwork();

                double[] input4 = new double[]{-1.0, 1.0, -1.0, -1.0};
                double[] target4 = new double[]{0.45, 0.55};

                network.backPropTraining(input4, target4, learningRate);
                //network.printNetwork();

                double[] input5 = new double[]{1.0, -1.0, -1.0, -1.0};
                double[] target5 = new double[]{0.45, 0.55};

                network.backPropTraining(input5, target5, learningRate);
                //network.printNetwork();

                double[] input6 = new double[]{1.0, 1.0, -1.0, -1.0};
                double[] target6 = new double[]{0.55, 0.45};

                network.backPropTraining(input6, target6, learningRate);
                //network.printNetwork();

                double[] input7 = new double[]{1.0, -1.0, 1.0, -1.0};
                double[] target7 = new double[]{0.55, 0.45};

                network.backPropTraining(input7, target7, learningRate);
                //network.printNetwork();

                double[] input8 = new double[]{1.0, -1.0, -1.0, 1.0};
                double[] target8 = new double[]{0.55, 0.45};

                network.backPropTraining(input8, target8, learningRate);
                //network.printNetwork();

                double[] input9 = new double[]{-1.0, 1.0, 1.0, -1.0};
                double[] target9 = new double[]{0.55, 0.45};

                network.backPropTraining(input9, target9, learningRate);
                //network.printNetwork();

                double[] input10 = new double[]{-1.0, 1.0, -1.0, 1.0};
                double[] target10 = new double[]{0.55, 0.45};

                network.backPropTraining(input10, target10, learningRate);
                //network.printNetwork();

                double[] input11 = new double[]{-1.0, -1.0, 1.0, 1.0};
                double[] target11 = new double[]{0.55, 0.45};

                network.backPropTraining(input11, target11, learningRate);
                //network.printNetwork();

                double[] input12 = new double[]{1.0, 1.0, 1.0, 1.0};
                double[] target12 = new double[]{0.55, 0.45};

                network.backPropTraining(input12, target12, learningRate);
                //network.printNetwork();
```

```
                double[] input13 = new double[]{1.0, 1.0, 1.0, -1.0};
                double[] target13 = new double[]{0.55, 0.45};

                network.backPropTraining(input13, target13, learningRate);
                //network.printNetwork();

                double[] input14 = new double[]{1.0, 1.0, -1.0, 1.0};
                double[] target14 = new double[]{0.55, 0.45};

                network.backPropTraining(input14, target14, learningRate);
                //network.printNetwork();

                double[] input15 = new double[]{1.0, -1.0, 1.0, 1.0};
                double[] target15 = new double[]{0.55, 0.45};

                network.backPropTraining(input15, target15, learningRate);
                //network.printNetwork();

                double[] input16 = new double[]{-1.0, 1.0, 1.0, 1.0};
                double[] target16 = new double[]{0.55, 0.45};

                network.backPropTraining(input16, target16, learningRate);
                //network.printNetwork();

                System.out.println("Finished cycle " + (i+1)+"\n");
                sum++;
            }

            double[] output = new double[2];
            double[] data = new double[4];

            data = new double[]{-1.0, -1.0, -1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Dark");
            output = network.processForward(data);
            isBright(output);

            data= new double[]{-1.0, -1.0, -1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Dark");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{-1.0, -1.0, 1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Dark");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{-1.0, 1.0, -1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Dark");
            output = network.processForward(data);
            isBright(output);


            data = new double[]{1.0, -1.0, -1.0, -1.0};
```

```
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Dark");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{1.0, 1.0, -1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{1.0, -1.0, 1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Dark");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{1.0, -1.0, -1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{-1.0, 1.0, 1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{-1.0, 1.0, -1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{-1.0, -1.0, 1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{1.0, 1.0, 1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{1.0, 1.0, 1.0, -1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{1.0, 1.0, -1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
```

```
            isBright(output);

            data = new double[]{1.0, -1.0, 1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            data = new double[]{-1.0, 1.0, 1.0, 1.0};
            System.out.println("Testing: " + Arrays.toString(data) +
"\nExpecting Bright");
            output = network.processForward(data);
            isBright(output);

            System.out.println("\nCompleted "+sum+" cycles of the training
data.");

            //asks if it would like to run the above loop again
            System.out.println("Train again? y/n");
            if(input.next()=="n"){
                train = false;
            }
        }
    }

    //Function for determining the results of the ANN output
    public static void isBright(double[] output){
        if(output[0] > output[1]){
            System.out.println("is bright\n\n");
        }else{
            System.out.println("is dark\n\n");
        }
    }
}
```