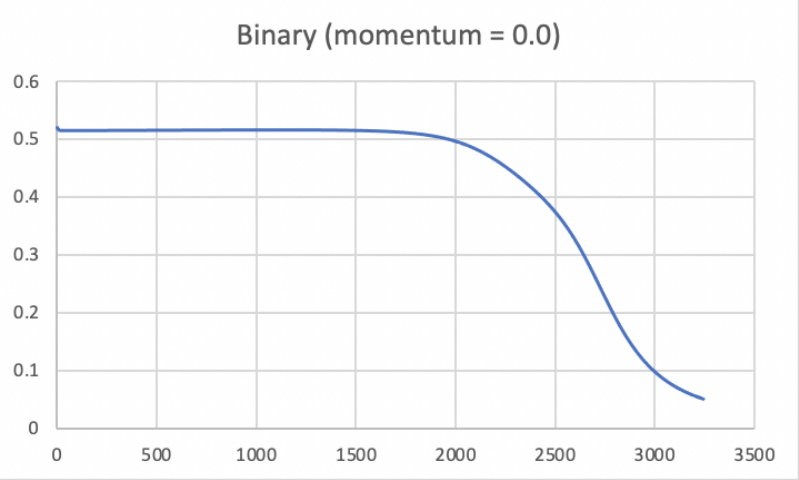
CPEN 502 Assignment 1

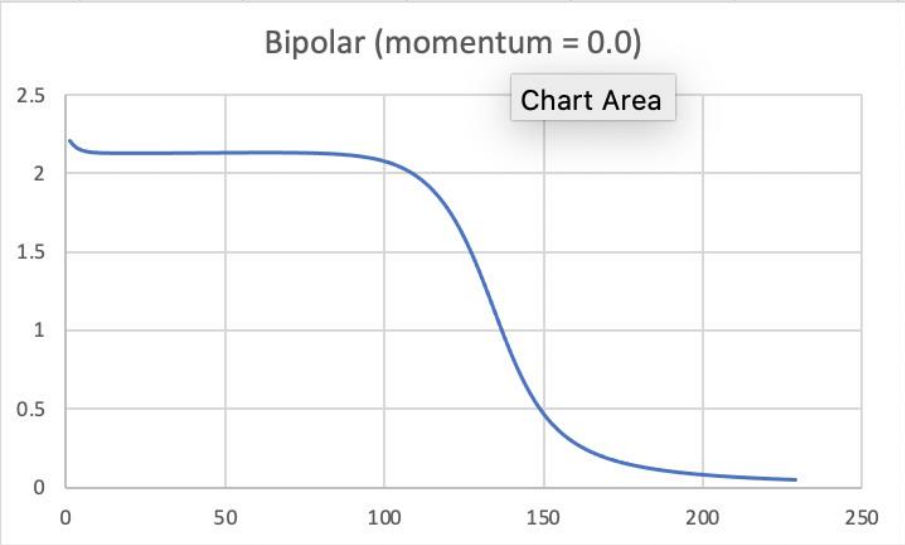
Jiaqi Zhang (**63174551**)

a) Define your XOR problem using a binary representation. Draw a graph of total error against number of epochs. On average, how many epochs does it take to reach a total error of less than 0.05? You should perform many trials to get your results, although you don't need to plot them all.



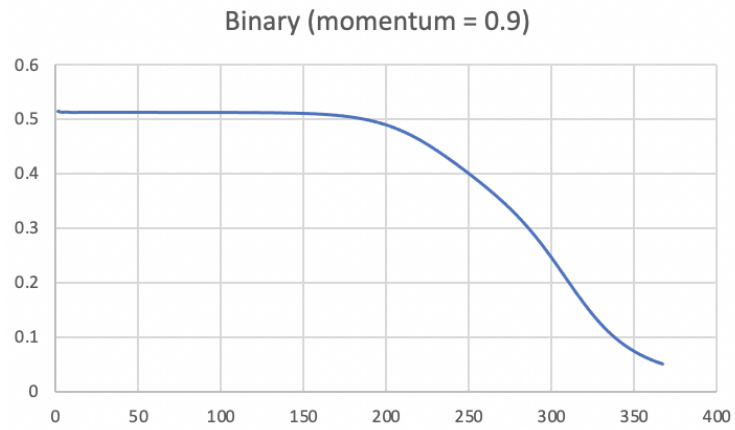
On average, it takes 3875 epochs to reach a total error of less than 0.05.

b) This time use a bipolar representation. Again, graph your results to show the total error varying against number of epochs. On average, how many epochs to reach a total error of less than 0.05?

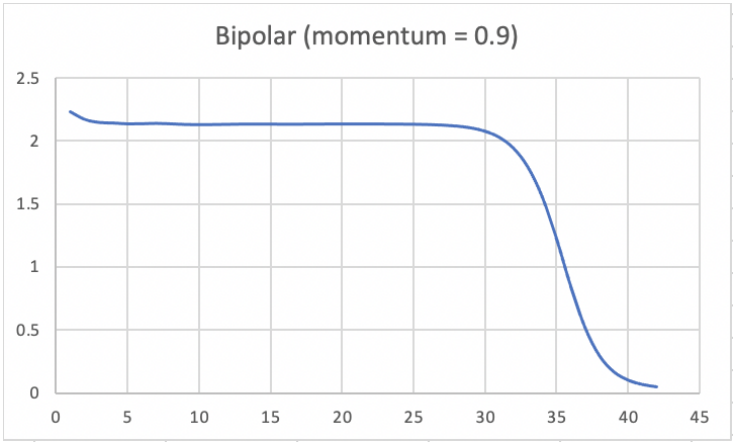


On average, it takes 258 epochs to reach a total error of less than 0.05.

c) Now set the momentum to 0.9. What does the graph look like now and how fast can 0.05 be reached?



On average, it takes 434 epochs to reach a total error of less than 0.05 when using Binary representation and momentum is 0.9.



On average, it takes 38 epochs to reach a total error of less than 0.05 when using Bipolar representation and momentum is 0.9.

Appendix: Source Code

Layer.java

```
package NeuralNet;

public class Layer {
    private boolean IS_BINARY = false;
    private static final String RANDOM = "random";
    private static final String ZERO = "zero";
    public static final String INPUT_LAYER = "input layer";
    public static final String HIDDEN_LAYER = "hidden layer";
    public static final String OUTPUT_LAYER = "output layer";

    private int size;
    private String type;
    private Layer prev;
    private Layer next;
    private double[] values;
    private double[] deltas;
    private double[][] currweights;
    // only for output layer
    private double[] expectedOutputs;
    private double[][] deltaWeights;

    public Layer(int size, String type){
        this.size = size;
        this.type = type;
        this.deltas = new double[size];
        if(type.equals(OUTPUT_LAYER)) {
            this.values = new double[size];
        } else {
            this.values = new double[size+1];
            values[size] = 1.0;    // bias neuron
        }
    }

    // set input data
    public void setInputs(double[] inputs) {
        if(!type.equals(INPUT_LAYER) || inputs.length != size) {
            return;
        }
        for(int i=0; i<size; i++) {
            values[i] = inputs[i];
        }
    }

    // set output data
    public void setOutputs(double[] outputs) {
        if(!type.equals(OUTPUT_LAYER) || outputs.length != size) {
            return;
        }
        expectedOutputs = new double[size];
        for(int i=0; i<size; i++) {
            expectedOutputs[i] = outputs[i];
        }
    }
```

```
    // connect current layer with prev layer and next layer
    public void connectLayer(Layer prevLayer, Layer nextLayer) {
        prev = prevLayer;
        next = nextLayer;
        // outputLayer
        if(type.equals(OUTPUT_LAYER)) {
            currweights = null;
            deltaWeights = null;
            expectedOutputs = new double[size];
        } else {
            currweights = new double[size+1][next.size];
            deltaWeights = new double[size+1][next.size];
            expectedOutputs = null;
        }
    }

    public double getRandom(double lowerBound, double upperBound) {
        return lowerBound + (upperBound-lowerBound)*Math.random();
    }

    public void setWeights(String setValue, double lowerBound, double
upperBound) {
        if(type.equals(OUTPUT_LAYER)) {
            return;
        }
        for (int j = 0; j < next.size; j++) {
            for (int i = 0; i < size + 1; i++) {
                if(setValue == RANDOM) {
                    currweights[i][j] = getRandom(lowerBound, upperBound);
                } else if(setValue == ZERO) {
                    currweights[i][j] = 0.0;
                }
                deltaWeights[i][j] = 0.0;
            }
        }
    }

    public void initializeWeights(double lowerBound, double upperBound) {
        setWeights(RANDOM, lowerBound, upperBound);
    }

    public void zeroWeights() {
        setWeights(ZERO, 0, 0);
    }

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

    public double customSigmoid(double x) {
        if (IS_BINARY) {
            return 1/(1+Math.exp(-x));
        }
        return sigmoid(x);
    }

    public void forwardPropagation() {
```

```
        if(type.equals(OUTPUT_LAYER)) return;
        for (int j=0; j<next.size; j++) {
            next.values[j] = 0.0;
            // include current layer bias neuron to predict next layer values
            for(int i=0; i<size+1; i++) {
                next.values[j] += currweights[i][j]*values[i];
            }
            next.values[j] = customSigmoid(next.values[j]);
        }
    }

    public double getDerivative(double value) {
        if(IS_BINARY) {
            return value*(1-value);
        }
        return (value+1)*(1-value)*0.5;
    }

    public void backwardPropagation(double learningRate, double momentum) {
        if(type.equals(INPUT_LAYER)) {
            return;
        }
        if(type.equals(OUTPUT_LAYER)) {
            for(int i=0; i<size; i++) {
                double error = expectedOutputs[i]-values[i];
                deltas[i] = getDerivative(values[i])*error;
            }
        } else if(type.equals(HIDDEN_LAYER)) {
            for(int i=0; i<size; i++) {
                deltas[i] = 0;
                for(int j=0; j<next.size; j++) {
                    deltas[i] += currweights[i][j]*next.deltas[j];
                }
                deltas[i] = getDerivative(values[i])*deltas[i];
            }
        }
        for(int j=0; j<size; j++) {
            for(int i=0; i <= prev.size; i++) {
                prev.deltaWeights[i][j] = momentum*prev.deltaWeights[i][j] +
learningRate*deltas[j]*prev.values[i];
                prev.currweights[i][j] += prev.deltaWeights[i][j];
            }
        }
    }

    public Layer getNext() {
        return next;
    }

    public Layer getPrev() {
        return prev;
    }

    public double[] getValues() {
        return values;
    }
}
```

NeuralNet.java

```
package NeuralNet;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;

public class NeuralNet implements NeuralNetInterface{
    private static final double ERROR_THRESHOLD = 0.05;
    private Layer inputLayer;
    private Layer hiddenLayer;
    private Layer outputLayer;

    private double learningRate;
    private double momentum;

    private int numInput;
    private int numHidden;
    private int numOutput;
    private double lowerBound = -0.5;
    private double upperBound = 0.5;
    private ArrayList<String> arr = new ArrayList<>();

    public NeuralNet(int numInput, int numHidden, int numOutput, double
learningRate, double momentum) {
        this.numInput = numInput;
        this.numHidden = numHidden;
        this.numOutput = numOutput;
        this.learningRate = learningRate;
        this.momentum = momentum;
        setUpNeuralNet();
    }

    public void setUpNeuralNet() {
        inputLayer = new Layer(numInput, Layer.INPUT_LAYER);
        hiddenLayer = new Layer(numHidden, Layer.HIDDEN_LAYER);
        outputLayer = new Layer(numOutput, Layer.OUTPUT_LAYER);
        inputLayer.connectLayer(null, hiddenLayer);
        hiddenLayer.connectLayer(inputLayer, outputLayer);
        outputLayer.connectLayer(hiddenLayer, null);
    }

    public int trainWholeProcess(double[][] inputs, double[][] outputs) {
        initializeWeights();
        double[] totalErrors = new double[numOutput];
        int epoch = 0;
        totalErrors[0] = ERROR_THRESHOLD;
        while(totalErrors[0]>=ERROR_THRESHOLD && epoch<10000) {
            for(int i=0; i<numOutput; i++) {
                totalErrors[i] = 0;
            }
            for (int i = 0; i < inputs.length; i++) {
                inputLayer.setInputs(inputs[i]);
                outputLayer.setOutputs(outputs[i]);
```

```
                feedForward();
                for(int j=0; j<numOutput; j++) {
                    double singleError = outputs[i][j] -
outputLayer.getValues()[j];
                    totalErrors[j] += Math.pow(singleError, 2) / 2;
                }
                backwardPropagation();
            }
            epoch++;
            System.out.println("Epoch: " + epoch + ", total error: " +
totalErrors[0]);
            arr.add("Epoch: " + epoch + ", total error: " + totalErrors[0]);
        }
        return epoch;
    }

    @Override
    public double outputFor(double[] X) {
        initializeWeights();
        inputLayer.setInputs(X);
        feedForward();
        return outputLayer.getValues()[0];
    }

    @Override
    public double train(double[] X, double argValue) {
        return argValue - outputFor(X);
    }

    @Override
    public void save(File argFile) {
        PrintStream file = null;
        try{
            file = new PrintStream(new FileOutputStream(argFile) );
            for(int i=0; i<arr.size(); i++) {
                file.println(arr.get(i));
            }
            file.flush();
            file.close();
        }
        catch(IOException e){
            System.out.println("ERROR");
        }
    }

    @Override
    public void load(String argFileName) throws IOException {

    }

    @Override
    public void initializeWeights() {
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
currLayer.getNext()) {
            currLayer.initializeWeights(lowerBound, upperBound);
        }
    }
```

```
    @Override
    public void zeroWeights() {
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
currLayer.getNext()) {
            currLayer.zeroWeights();
        }
    }

    public void feedForward() {
        for(Layer currLayer=inputLayer; currLayer != outputLayer; currLayer =
currLayer.getNext()) {
            currLayer.forwardPropagation();
        }
    }

    public void backwardPropagation() {
        for(Layer currLayer=outputLayer; currLayer != inputLayer; currLayer =
currLayer.getPrev()) {
            currLayer.backwardPropagation(learningRate, momentum);
        }
    }
}
```