CPEN 502 Part1a Report

Shanny Lu - 57267783

In this file, I use green to highlight my answer to each question. Note that for all figures below, I ran 200 trails for each type of representation, and selected the trial satisfying the condition $trials\%20 == 0$ and plotted them in the figure.

1 a) binary representation

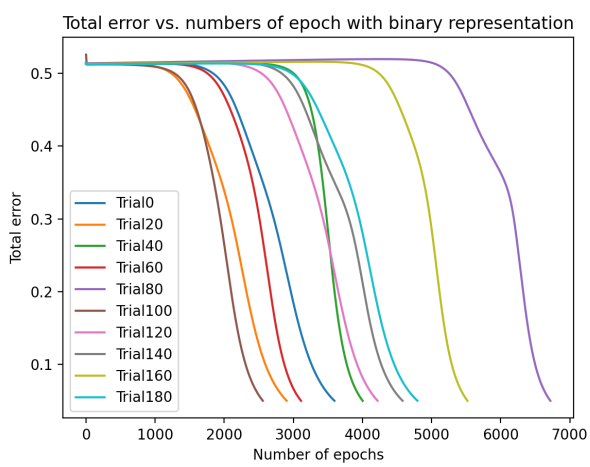
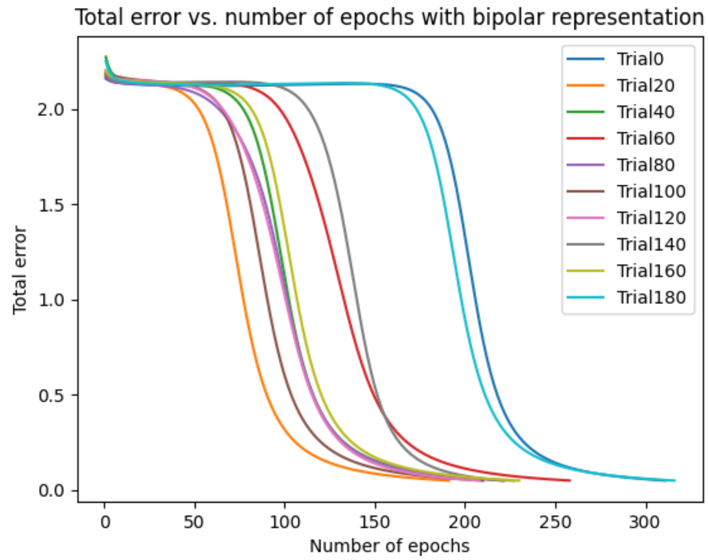


Figure 1: standard backpropagation of XOR problem using a binary representation.

Answer: On average, it takes 4093 epochs for binary representation to reach a total error of less than 0.05.

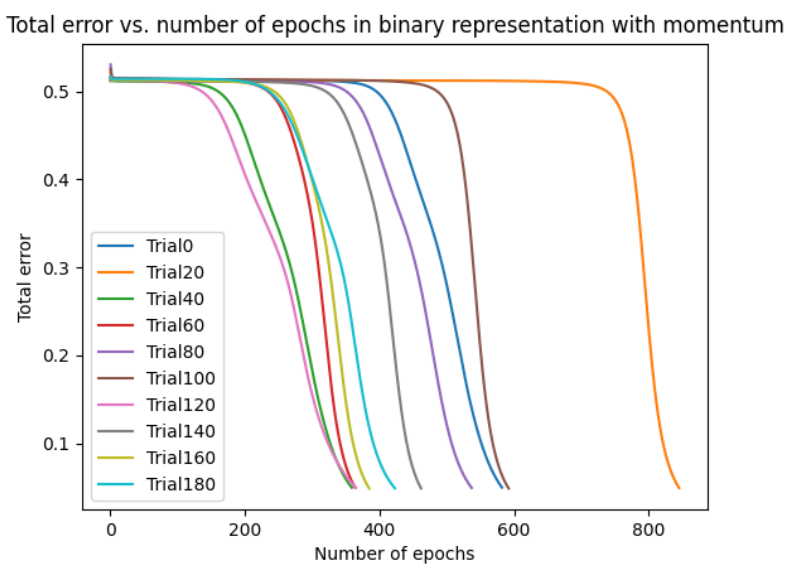
My result is quite similar to the benchmark, the only difference is that my total error is half of the benchmark. I believe it is because I'm using $\frac{1}{2}\sum_p(y_i - c_i)^2$, while the benchmark didn't consider the $\frac{1}{2}$ in front. This reasoning applies for all figures in this assignment.

2 b) bipolar representation

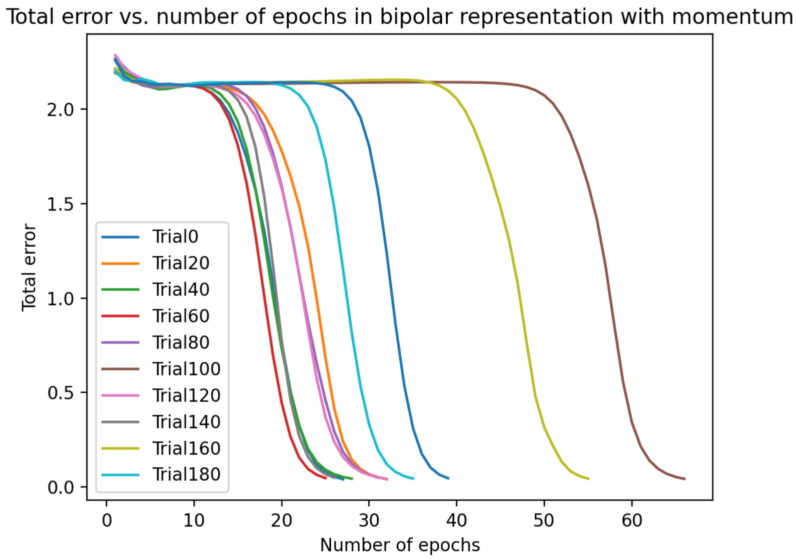


Answer: On average, it takes 260 epochs for bipolar representation to reach a total error of less than 0.05.

3 c) backpropagation with momentum = 0.9



Answer: On average, it takes 464 epochs for bipolar with momentum = 0.9 to reach a total error of less than 0.05.



Answer: On average, it takes 37 epochs for bipolar with momentum = 0.9 to reach a total error of less than 0.05.

Appendix: Source Code Implemented

A main.java

```
package main.java.ece.cpen502;

import com.github.sh0nk.matplotlib4j.Plot;
import com.github.sh0nk.matplotlib4j.PythonExecutionException;

import java.io.IOException;
import java.util.ArrayList;

public class main {

    static ArrayList<Double> errorList = new ArrayList<Double>();
    static ArrayList<Integer> epochList = new ArrayList<Integer>();

    public static void main(String[] args) throws PythonExecutionException, IOException {

        double error_threshold = 0.05;
        int interations = 200;
        double errorSum;
        int epoch;
        int totalEpoch;


        // binary representation
        double binaryInputData[][] = {{0,0},{1,0},{0,1},{1,1}};
        double binaryExpectedOutput[][] = {{0},{1},{1},{0}};

        // bipolar representation
        double bipolarInputData[][] = {{-1,-1},{1,-1},{-1,1},{1,1}};
        double bipolarExpectedOutput[][] = {{-1},{1},{1},{-1}};


        NeuralNet binary = new NeuralNet(2,4,1,0.2,0.0, true);
        NeuralNet bipolar = new NeuralNet(2,4,1,0.2,0.0,false);

        System.out.println("Binary representation with momentum = 0.0");
         // binary representation
        errorList.clear();
        epochList.clear();
        Plot plt1 = Plot.create();
        totalEpoch=0;

        // Train for 200 times(Binary representation)
        for(int j=0; j<interations;j++){
            binary.initializeWeights();
            epoch=0;
            errorList.clear();
            epochList.clear();
            do{
                errorSum = 0;
                for(int i=0; i<binaryInputData.length; i++){
                    double error = binary.train(binaryInputData[i], binaryExpectedOutput[i][0]);
```

```java
                errorSum += Math.pow(error, 2) / 2; // E = 1/2 * SUM(C-y)^2
            }
            epoch++;
            errorList.add(errorSum);
            epochList.add(epoch);
        } while (error_threshold < errorSum);
        totalEpoch +=epoch;
        if( j%20 == 0){
            plt1.plot().add(epochList, errorList).label("Trial" + j);
        }
    }
    plt1.xlabel("Number of epochs");
    plt1.ylabel("Total error");
    plt1.title("Total error vs. numbers of epoch with binary representation");
    plt1.legend();
    plt1.savefig("binary.png");
    plt1.show();
    System.out.printf("Average of epoch for binary is %d\n", totalEpoch/interations);



    System.out.println("Bipolar representation with momentum = 0.0");
    // bipolar representation
    errorList.clear();
    epochList.clear();
    Plot plt2 = Plot.create();
    totalEpoch=0;

    // Train for 200 times(Bipolar representation)
    for(int j=0; j<interations;j++){
        bipolar.initializeWeights();
        epoch=0;
        errorList.clear();
        epochList.clear();
        do{
            errorSum = 0;
            for(int i=0; i<bipolarInputData.length; i++){
                double error = bipolar.train(bipolarInputData[i],
                    bipolarExpectedOutput[i][0]);
                errorSum += Math.pow(error, 2) / 2; // E = 1/2 * SUM(C-y)^2
            }
            epoch++;
            errorList.add(errorSum);
            epochList.add(epoch);
        } while (error_threshold < errorSum);
        totalEpoch +=epoch;
        if( j%20 == 0){
            plt2.plot().add(epochList, errorList).label("Trial" + j);
        }
    }
    plt2.xlabel("Number of epochs");
    plt2.ylabel("Total error");
    plt2.title("Total error vs. number of epochs with bipolar representation");
    plt2.legend();
    plt2.savefig("bipolar.png");
    plt2.show();
    System.out.printf("Average of epoch for bipolar is %d\n", totalEpoch/interations);
```

```
NeuralNet binary_momentum = new NeuralNet(2, 4, 1, 0.2, 0.9, true);
NeuralNet bipolar_momentum = new NeuralNet(2,4,1,0.2,0.9,false);


// Binary representation with momentum = 0.9
System.out.println("Binary representation with momentum = 0.9");
// binary representation
errorList.clear();
epochList.clear();
Plot plt3 = Plot.create();
totalEpoch=0;

//Train for 200 times(Binary with momentum representation)
for(int j=0; j<interations;j++){
    binary_momentum.initializeWeights();
    epoch=0;
    errorList.clear();
    epochList.clear();
    do{
        errorSum = 0;
        for(int i=0; i<binaryInputData.length; i++){
            double error = binary_momentum.train(binaryInputData[i],
                binaryExpectedOutput[i][0]);
            errorSum += Math.pow(error, 2) / 2; // E = 1/2 * SUM(C-y)^2
        }
        epoch++;
        errorList.add(errorSum);
        epochList.add(epoch);
} while (error_threshold < errorSum);
    totalEpoch +=epoch;
    if( j%20 == 0){
        plt3.plot().add(epochList, errorList).label("Trial" + j);
    }
}
plt3.xlabel("Number of epochs");
plt3.ylabel("Total error");
plt3.title("Total error vs. number of epochs in binary representation with momentum");
plt3.legend();
plt3.savefig("binary_momentum.png");
plt3.show();
System.out.printf("Average of epoch for binary_momentum is %d\n",
    totalEpoch/interations);


// Bipolar representation with momentum = 0.9
System.out.println("Binary representation with momentum = 0.9");
// binary representation
errorList.clear();
epochList.clear();
Plot plt4 = Plot.create();
totalEpoch=0;

// Train for 200 times(bipolar with momentum representation)
for(int j=0; j<interations;j++){
    bipolar_momentum.initializeWeights();
    epoch=0;
```

```
            errorList.clear();
            epochList.clear();
        do{
            errorSum = 0;
            for(int i=0; i<bipolarInputData.length; i++){
                double error = bipolar_momentum.train(bipolarInputData[i],
                    bipolarExpectedOutput[i][0]);
                errorSum += Math.pow(error, 2) / 2; // E = 1/2 * SUM(C-y)^2
            }
            epoch++;
            errorList.add(errorSum);
            epochList.add(epoch);
        } while (error_threshold < errorSum);
            totalEpoch +=epoch;
            if( j%20 == 0){
                plt4.plot().add(epochList, errorList).label("Trial" + j);
            }
        }
        plt4.plot().add(epochList,errorList);
        plt4.xlabel("Number of epochs");
        plt4.ylabel("Total error");
        plt4.title("Total error vs. number of epochs in bipolar representation with momentum");
        plt4.legend();
        plt4.savefig("bipolar_momentum.png");
        plt4.show();
        System.out.printf("Average of epoch for bipolar_momentum is %d\n",
            totalEpoch/interations);
    }
}
```

B NeuralNet.java

```
package main.java.ece.cpen502;

import java.io.File;
import java.io.IOException;

public class NeuralNet implements NeuralNetInterface {

    private int NumInputs = 2;
    private int NumHidden = 4;
    private int NumOutputs = 1;
    private double learningRate = 0.2;
    private double momentum = 0.0;
    double error_threshold = 0.05;


    private boolean binary = true; // true for binary, false for bipolar



    //weights matrix for each layer
    double[][] w1 = new double[NumInputs+1][NumHidden+1]; // add 1 for bias
    double[][] w2 = new double[NumHidden+1][NumOutputs];
```

```java
double[][] w1Delta = new double[NumInputs+1][NumHidden+1];
double[][] w2Delta = new double[NumHidden+1][NumOutputs];
double[][] savedlastDeltaWeight1 = new double[NumInputs+1][NumHidden+1];
double[][] savedlastDeltaWeight2 = new double[NumHidden+1][NumOutputs];


double[] inputLayer = new double[NumInputs + 1];
double[] hiddenLayer = new double[NumHidden + 1];
double[] outputLayer = new double[NumOutputs];
double[] outputDelta = new double[NumOutputs]; //error signal
double[] hiddenDelta = new double[NumHidden+1];
double[] Error = new double[NumOutputs];


public NeuralNet(int argNumInputs, int argNumHidden, int argNumOutputs, double
    argLearningRate,
            double argMomentumTerm, boolean argbinary){
    this.NumInputs = argNumInputs;
    this.NumHidden = argNumHidden;
    this.NumOutputs = argNumOutputs;
    this.learningRate = argLearningRate;
    this.momentum = argMomentumTerm;
    this.binary = argbinary; //true for binary
}

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

@Override
public double customSigmoid(double x){
    //for binary representation
    return 1/(1 + Math.exp(-x));
}

@Override
public void initializeWeights(){
    // Initialize weights to random values in the range -0.5 to +0.5
    for(int i=0; i< w1.length; i++){
        for(int j=0; j< w1[0].length; j++){
            w1[i][j] = Math.random() - 0.5;
            savedlastDeltaWeight1[i][j] = 0;
            w1Delta[i][j] = 0;
        }
    }

    for(int i=0; i< w2.length; i++){
        for(int j=0; j< w2[0].length; j++){
            w2[i][j] = Math.random() - 0.5;
            savedlastDeltaWeight2[i][j] = 0;
            w2Delta[i][j] = 0;
        }
    }
}
```

```java
    @Override
    public void zeroWeights(){
//        Initialize weights to random values to 0
        for(int i=0; i< w1.length; i++){
            for(int j=0; j< w1[0].length; j++){
                w1[i][j] = 0;
            }
        }

        for(int i=0; i< w2.length; i++){
            for(int j=0; j< w2[0].length; j++){
                w2[i][j] = 0;
            }
        }
    }

    public void forwardPropagation(double[] input){
        for(int i=0; i<input.length; i++){
            inputLayer[i] = input[i];
        }
        inputLayer[input.length] = 1; // add bias term


        for(int j=0; j<NumHidden; j++){
            hiddenLayer[j] = 0;
            for(int i=0; i<NumInputs+1; i++){
                hiddenLayer[j] += w1[i][j] * inputLayer[i];
            }
            if(binary) { // binary
                hiddenLayer[j] = customSigmoid(hiddenLayer[j]);
            } else { // bipolar
                hiddenLayer[j] = sigmoid(hiddenLayer[j]);
            }

        }
        hiddenLayer[NumHidden] = 1;

        for(int j=0; j<NumOutputs; j++){
            outputLayer[j] = 0;
            for(int i=0; i<NumHidden+1; i++) {
                outputLayer[j] += w2[i][j] * hiddenLayer[i];
            }
            if (binary) { //binary
                outputLayer[j] = customSigmoid(outputLayer[j]);
            } else{ //bipolar
                outputLayer[j] = sigmoid(outputLayer[j]);
            }
        }
    }

    public void backPropagation(){
        //compute error signal when y is an output unit
        for(int i=0; i<NumOutputs; i++){
            //TODO
            outputDelta[i] = 0;
            if(binary){ //binary
```

```
            outputDelta[i] = outputLayer[i] * (1 - outputLayer[i]) * Error[i];
        } else { //bipolar : derivative is different!
            outputDelta[i] = (1 + outputLayer[i]) * (1 - outputLayer[i]) / 2 * Error[i];
        }
    }

    // Update weights for hidden to output layer first
    for (int j = 0; j < NumOutputs; j++) {
        for(int k=0; k<NumHidden+1; k++) {
//              w2Delta[k][j] = momentum * w2Delta[k][j] + learningRate * outputDelta[j] *
    hiddenLayer[k] ;
            w2Delta[k][j] = momentum * savedlastDeltaWeight2[k][j] + learningRate *
                outputDelta[j] * hiddenLayer[k];
            w2[k][j] += w2Delta[k][j];
            savedlastDeltaWeight2[k][j] = w2Delta[k][j];
        }
    }

    //compute error signal when y is a hidden unit
    for(int k = 0; k<NumHidden; k++){
        hiddenDelta[k] = 0;
        for(int j=0; j<NumOutputs; j++){
            if(binary){
                hiddenDelta[k] += hiddenLayer[k] * (1 - hiddenLayer[k]) * outputDelta[j] *
                    w2[k][j];
            } else {
                hiddenDelta[k] += (1 + hiddenLayer[k]) * (1 - hiddenLayer[k]) / 2 *
                    outputDelta[j] * w2[k][j];
            }
        }
    }

    // Update weights for input to hidden layer
    for(int k=0; k<NumHidden; k++) {
        for(int i=0; i< NumInputs+1; i++) {
//              w1Delta[i][k] = momentum * w1Delta[i][k] + learningRate * hiddenDelta[k] *
    inputLayer[i];
            w1Delta[i][k] = momentum * savedlastDeltaWeight1[i][k] + learningRate *
                hiddenDelta[k] * inputLayer[i];
            w1[i][k] += w1Delta[i][k];
            savedlastDeltaWeight1[i][k] = w1Delta[i][k];
        }
    }


}

@Override
public double outputFor(double [] X){
    // TODO: not used for part 1a
    if(Error[0] < error_threshold){
        return outputLayer[0];
    } else{
        System.out.println("The neural net is not trained well yet.\n");
        return 0.0;
    }
}
```

```java
    @Override
    // train the neutral net for one dataset
    public double train(double [] X, double argValue){
        forwardPropagation(X);
        for(int i=0; i<NumOutputs; i++){ // NumOutputs is 1 in this example
            Error[i] = argValue - outputLayer[i];
        }
        backPropagation();
        return Error[0]; // hardcode as the number of output is 1
    }


    @Override
    public void save(File argFile){
        // TODO: no need for part 1a
    }

    @Override
    public void load(String argFileName) throws IOException{
        // TODO: no need for part 1a
    }


}
```

C CommonInterface.java

```java
package main.java.ece.cpen502;

import java.io.File;
import java.io.IOException;

/**
 * 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 {
    /**d
     * @param X The input vector. An array of doubles.
     * @return The value returned by th 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);

    /**
     * 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;
}
```

D NeuralNetInterface.java

```java
package main.java.ece.cpen502;

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 arbB 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) = 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
```