UBC-CPEN-502 Assignment 1a

Back-propagation Learning

Name: Yue Shen Student #: 76006386

Graphs of Training Result

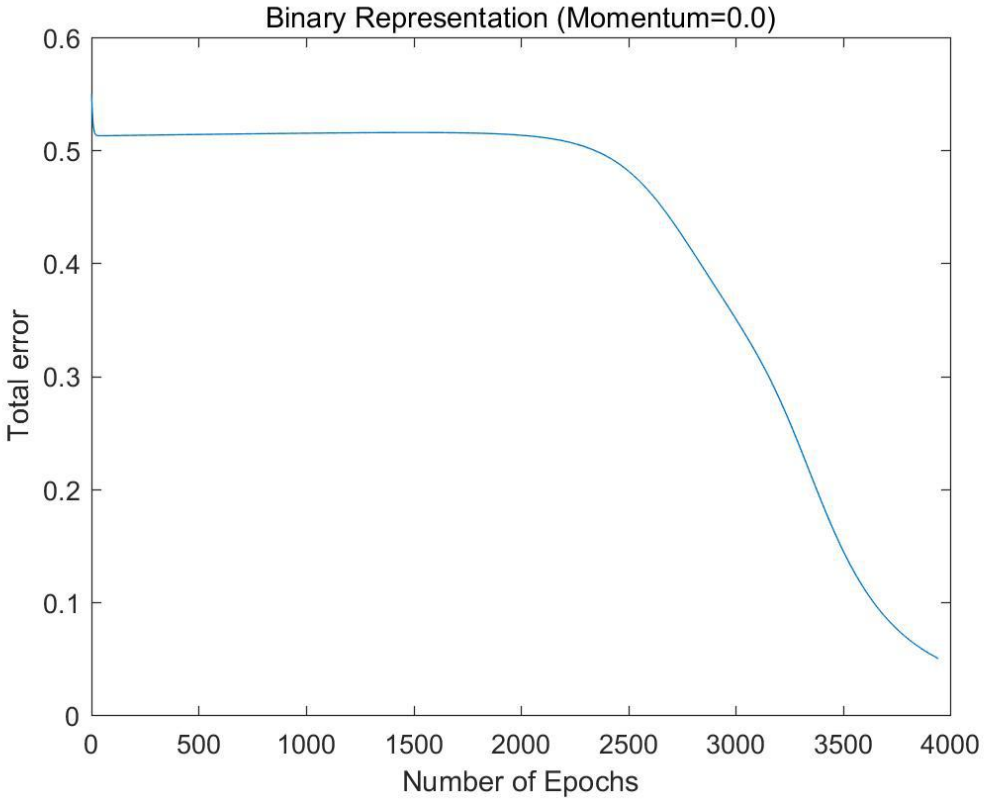
(1) Set up your network in a 2-input, 4-hidden and 1-output configuration. Apply the XOR training set. Initialize weights to random values in the range -0.5 to +0.5 and set the learning rate to 0.2 with momentum at 0.0.

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?

For binary representation, 0.0 momentum, after 2000 trials of training, the average needed epochs is 3835.

Representation Type	Learning Rate	momentum	Number of Trials	Average Epochs	Max Epochs	Min Epochs
Binary	0.2	0.0	2000	3835	17361	2411

Graph 1 below shows the 1000th trial with 3945 epochs:



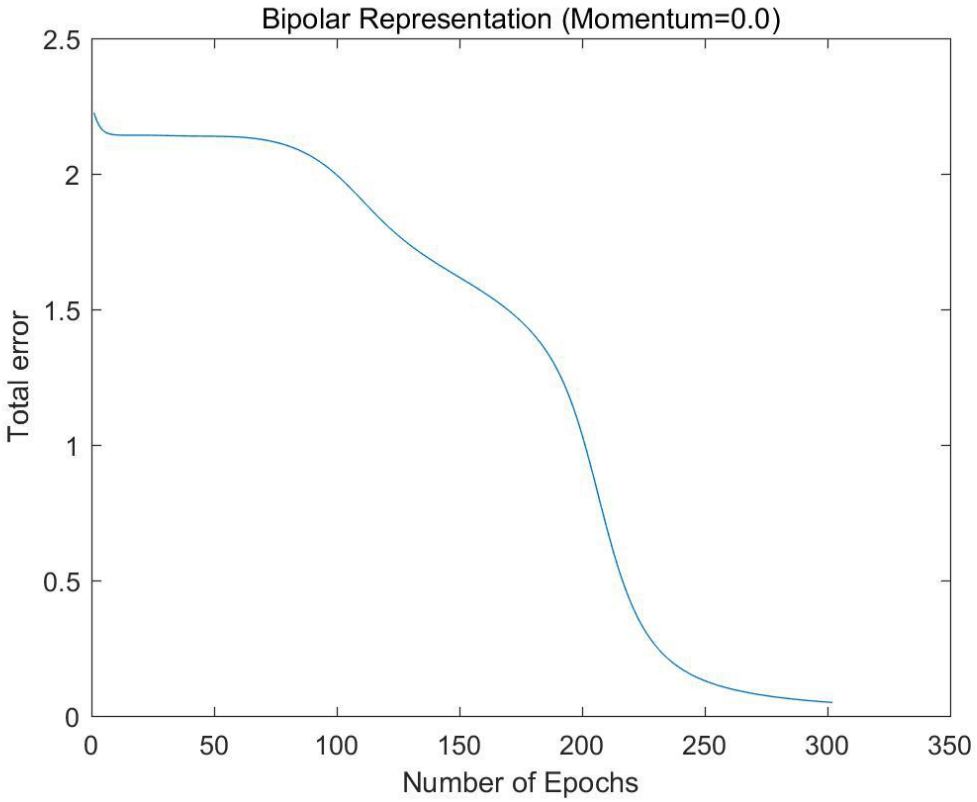
Graph 1 Binary Representation with Momentum 0.0

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?

For bipolar representation, 0.0 momentum, after 2000 trials of training, the average needed epochs is 305.

Representation Type	Learning Rate	momentum	Number of Trials	Average Epochs	Max Epochs	Min Epochs
Bipolar	0.2	0.0	2000	305	564	200

Graph 2 below shows the 1000th trial with 302 epochs:



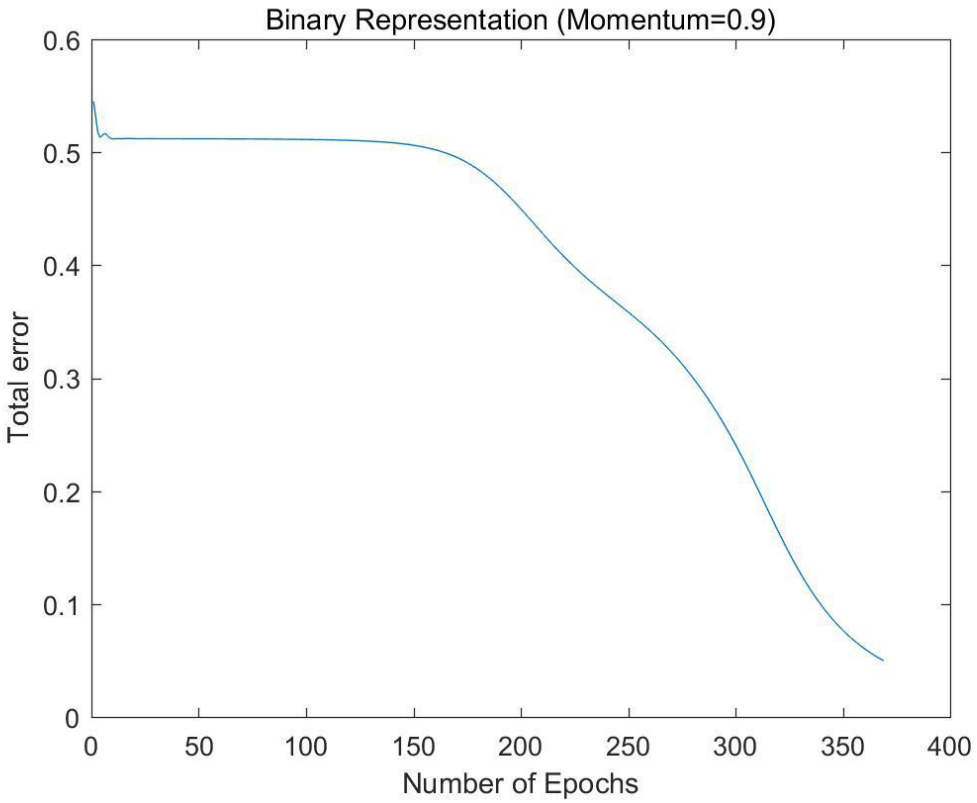
Graph 2 Bipolar Representation with Momentum 0.0

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

For binary representation, 0.9 momentum, after 2000 trials of training, the average needed epochs is 388.

Representation Type	Learning Rate	momentum	Number of Trials	Average Epochs	Max Epochs	Min Epochs
Binary	0.2	0.9	2000	388	1131	234

Graph 3 below shows the 1000th trial with 369 epochs:

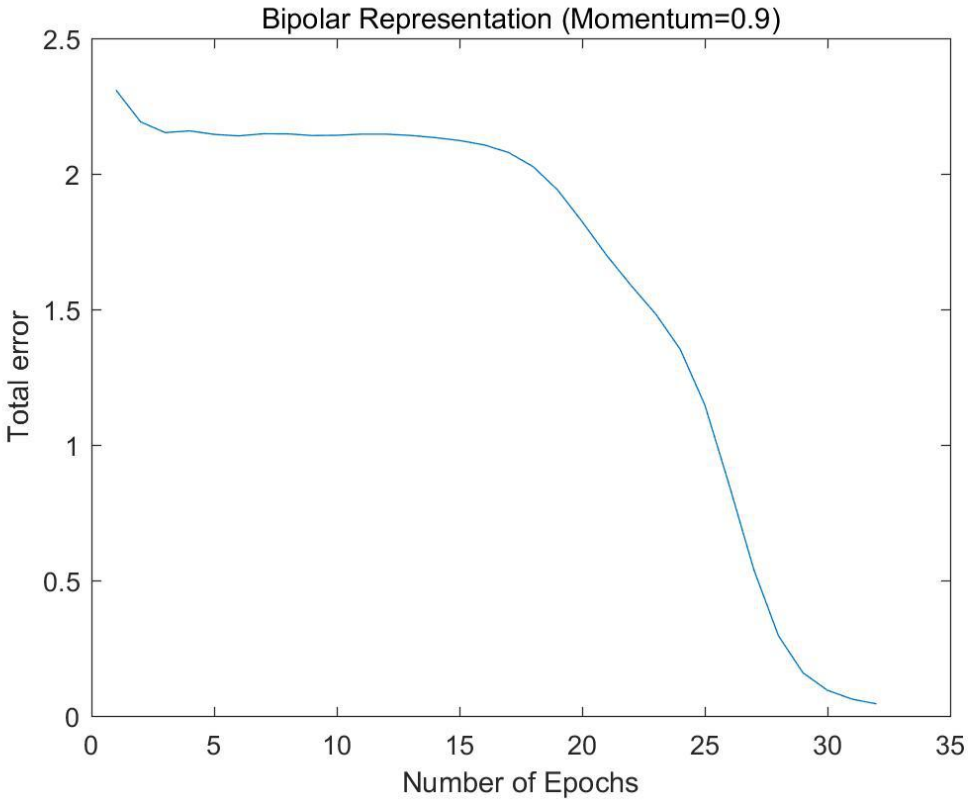


Graph 3 Binary Representation with Momentum 0.9

For bipolar representation, 0.9 momentum, after 2000 trials of training, the average needed epochs is 37.

Representation Type	Learning Rate	momentum	Number of Trials	Average Epochs	Max Epochs	Min Epochs
Bipolar	0.2	0.9	2000	37	95	21

Graph 4 below shows the 1000th trial with 32 epochs:



Graph 4 Bipolar Representation with Momentum 0.9

Appendix

1. NeuralNet.java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.util.Random;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintStream;

public class NeuralNet implements NeuralNetInterface {
    static final int MAX_INPUTS_NUM = 20;
    static final int MAX_HIDDEN_NUM = 20;
    static final int MAX_OUTPUTS_NUM = 20;

    private int inputsNum;
    private int hiddenNum;
    private int outputsNum;
    private double learningRate;
    private double momentumTerm;
    private double argA;
    private double argB;
    private String activeType;

    // Weights from input layers to hidden layers
    private double[][] weight1 = new
double[MAX_INPUTS_NUM][MAX_HIDDEN_NUM];
    // Weights form hidden layers to output layers
    private double[][] weight2 = new
double[MAX_HIDDEN_NUM][MAX_OUTPUTS_NUM];
    // Weights difference between updated weights and previous one.
    private double[][] weightChange1 = new
double[MAX_INPUTS_NUM][MAX_HIDDEN_NUM];
    private double[][] weightChange2 = new
double[MAX_HIDDEN_NUM][MAX_OUTPUTS_NUM];

    private double[] inputsNeuron = new double[MAX_INPUTS_NUM ];
    private double[] hiddenNeuron = new double[MAX_HIDDEN_NUM];
    private double[] outputsNeuron = new double[MAX_OUTPUTS_NUM];
    // Value of delta in the hidden layer
    private double[] deltaHidden = new double[MAX_HIDDEN_NUM];
```

```
    // Value of delta in the output layer
    private double[] deltaOutput = new double[MAX_OUTPUTS_NUM];
    private double error = 0;

    public NeuralNet(int inputsNum, int hiddenNum, int outputsNum,
                     double learningRate, double momentumTerm,
                      double argA, double argB, String activeType)
{
        this.inputsNum = inputsNum;
        this.hiddenNum = hiddenNum;
        this.outputsNum = outputsNum;
        this.learningRate = learningRate;
        this.momentumTerm = momentumTerm;
        this.argA = argA;
        this.argB = argB;
        this.activeType = activeType;
    }


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

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


    @Override
    public void initializeWeights() {
        for (int i = 0; i < inputsNum + 1; i++) {
            // The last index represents the bias of input
            for (int h = 0; h < hiddenNum; h++) {
                weight1[i][h] = getRandomWeight(-0.5, 0.5);
                weightChange1[i][h] = 0.0;
            }
        }

        for (int h = 0; h < hiddenNum + 1; h++) {
            for (int j = 0; j < outputsNum; j++) {
                weight2[h][j] = getRandomWeight(-0.5, 0.5);
                weightChange2[h][j] = 0.0;
```

```
            }
        }
    }


    private double getRandomWeight(double minWeight, double
maxWeight) {
        double random = new Random().nextDouble();
        return minWeight + (random * (maxWeight - minWeight));
    }


    @Override
    public void zeroWeights() {
        for (int i = 0; i < inputsNum + 1; i++) {
            for (int h = 0; h < hiddenNum; h++) {
                weight1[i][h] = 0.0;
            }
        }
        for (int h = 0; h < hiddenNum+1; h++) {
            for (int j = 0; j < outputsNum; j++) {
                weight2[h][j] = 0.0;
            }
        }
    }



    /*Construct neural net and do feed forward process, calculating
the final output*/
    @Override
    public double outputFor(double[] X) {
        //Firstly, given the input X[], set up the neural net
        for(int i = 0;i < inputsNum; i++){
            inputsNeuron[i] = X[i];
        }
        //Add bias
        inputsNeuron[inputsNum] = 1;
        hiddenNeuron[hiddenNum] = 1;

        //Compute hidden layer
        for(int h = 0; h < hiddenNum; h++){
          hiddenNeuron[h] = 0;
            for(int i = 0;i < inputsNum + 1; i++){
```

```
                hiddenNeuron[h] += weight1[i][h] *
inputsNeuron[i];
            }
            hiddenNeuron[h] = customSigmoid(hiddenNeuron[h]);
        }

        //Compute output layer
        for(int j = 0; j < outputsNum; j++){
          outputsNeuron[j] = 0;
            for(int h = 0;h < hiddenNum + 1; h++){
                outputsNeuron[j] += weight2[h][j] *
hiddenNeuron[h];
            }
            outputsNeuron[j] = customSigmoid(outputsNeuron[j]);
        }
        return outputsNeuron[0];  //Single output
    }



    /*Backward process, computing the delta for each layer and then
update weights*/
    private void updateWeight(double argValue){
        //Compute deltaOutput[] for output layer
        for(int j = 0; j < outputsNum; j++){
            if(activeType.equals("binary"))
            deltaOutput[j] = (argValue - outputsNeuron[j]) * (1 -
outputsNeuron[j]) * outputsNeuron[j];
            else if(activeType.equals("bipolar")) {
             deltaOutput[j] = (argValue - outputsNeuron[j]) * 0.5
* (1 - outputsNeuron[j]) * (1 + outputsNeuron[j]);
            }
        }
        //Update weights from output layer to hidden layer
        for(int j = 0; j < outputsNum; j++){
            for(int h = 0; h < hiddenNum + 1; h++){
                weight2[h][j] += momentumTerm * weightChange2[h][j]
+ learningRate * deltaOutput[j] * hiddenNeuron[h];
                weightChange2[h][j] = momentumTerm *
weightChange2[h][j] + learningRate * deltaOutput[j] *
hiddenNeuron[h];
            }
        }
```

```
        //Compute deltaHidden[] for hidden layer
        for(int h = 0; h < hiddenNum; h++){
            for(int j = 0; j < outputsNum; j++){
                deltaHidden[h] += deltaOutput[j] * weight2[h][j];
            }
            if(activeType.equals("binary"))
            deltaHidden[h] = (1 - hiddenNeuron[h]) * hiddenNeuron[h]
* deltaHidden[h];
            else if(activeType.equals("bipolar")) {
            deltaHidden[h] = 0.5 * (1 - hiddenNeuron[h]) * (1 +
hiddenNeuron[h]) * deltaHidden[h];
            }
        }
        //Update weights from hidden layer to input layer
        for(int h = 0; h < hiddenNum; h++){
            for(int i = 0; i < inputsNum + 1; i++){
                weight1[i][h] += momentumTerm * weightChange1[i][h]
+ learningRate * deltaHidden[h] * inputsNeuron[i];
                weightChange1[i][h] = momentumTerm *
weightChange1[i][h] + learningRate * deltaHidden[h] *
inputsNeuron[i];
            }
        }
    }


    /*Train the network and return the error of each single neuron*/
    @Override
    public double train(double[] X, double argValue) {
       double output;
       try {
          output = outputFor(X);
          error = 0.5 * (argValue - output) * (argValue - output);
          updateWeight(argValue);
       }catch(Exception e) {
          System.out.println(e);
       }
        return error;
    }



   /*Save weights of a neural net*/
    @Override
```

```
    public void save(File argFile) {
        PrintStream saveWeight = null;
        try {
          saveWeight = new PrintStream(new
FileOutputStream(argFile));
        }catch(Exception e) {
          System.out.println(e);
        }
        for(int h = 0; h < hiddenNum; h++){
          for(int i = 0;i < inputsNum + 1; i++) {
             saveWeight.println(weight1[i][h]);
          }
        }
        for(int j = 0; j < outputsNum; j ++) {
          for(int h = 0; h < hiddenNum+1; h++) {
             saveWeight.println(weight2[h][j]);
          }
        }
        saveWeight.close();
     }


    /*Load weights of a neural net from a file*/
    @Override
    public void load(String argFileName) throws IOException {
        FileInputStream weightFile = new
FileInputStream(argFileName);
         BufferedReader weightReader = new BufferedReader(new
InputStreamReader(weightFile));
         for(int h = 0; h < hiddenNum; h++){
          for(int i = 0; i < inputsNum + 1; i++) {
             weight1[i][h] =
Double.valueOf(weightReader.readLine());
          }
         }
         for(int j = 0; j < outputsNum; j++) {
          for(int h = 0; h< hiddenNum + 1; h++) {
             weight2[h][j] =
Double.valueOf(weightReader.readLine());
          }
         }
         weightReader.close();
    }
}
```

2. NeuralMain.java

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import java.util.LinkedList;

public class NeuralMain {

    public static void main(String[] args) {
       /*initialize arguments*/
       int inputsNum = 2;
       int hiddenNum = 4;
       int outputsNum = 1;
       double learningRate = 0.2;
       double momentumTerm = 0.9;
       double argA = -1;   // For binary input, argA = 0
       double argB = 1;
       String activeType = new String("bipolar");

        /*Binary test*/
//      double inputs[][] = {{0, 0}, {0, 1},{ 1, 0}, {1, 1}};
//      double targets[] = {0, 1, 1, 0};

        /*Bipolar test*/
        double inputs[][] = {{1, 1}, {1, -1},{ -1, 1}, {-1, -1}};
        double targets[] = {-1, 1, 1, -1};

        double acceptError = 0.05;
      //String list to store error of each epoch
        List<String> errorSave = new LinkedList<>();
      //Set training times
        int trialsNum = 2000;
      //Average needed epochs after a number of trials
        int epochsAvg = 0;

        NeuralNet myNNet= new
NeuralNet(inputsNum,hiddenNum,outputsNum,
                                    learningRate,momentumTerm,
                                    argA,argB,activeType);

        int epochMin = 10000;
        int epochMax = 0;
        for(int trial = 1;trial <= trialsNum; trial++) {
```

```
            //Initialize epochs for every trial
            int epochs = 0;
            //Set error bigger than acceptError
            double error = 1.05;
            myNNet.initializeWeights();
            while(error > acceptError) {
            error = 0;
            for (int i=0; i < hiddenNum; i++) {
               double[] inputNeuron = inputs[i];
               double argValue = targets[i];
                error = error + myNNet.train(inputNeuron,argValue);

            }
            epochs++;
            if(trial == trialsNum / 2) {
                 System.out.println("epochs: "+epochs+" | "+"error:
"+error);
                 errorSave.add(Double.toString(error));
               }
             }
            //Find min number of epochs
            if(epochs > epochMax) epochMax = epochs;
            //Find max number of epochs
            if(epochs < epochMin) epochMin = epochs;
            epochsAvg += epochs;
             }

        /*Save error data of the first trial to a text file*/
        System.out.println("Above shows one example trial");
        try {
           Files.write(Paths.get("./errorSave.txt"), errorSave);
           }catch(Exception e) {
              System.out.println(e);
           }

        /*Calculate Average needed epochs*/
        epochsAvg /= trialsNum;
        System.out.println("After "+ trialsNum + " trials, the
average needed epochs is " + epochsAvg);
        System.out.println("The max number of epochs:"+ epochMax);
        System.out.println("The min number of epochs:"+ epochMin);
    }
}
```