

CPEN 502

Assignment 1 Report

Table of Contents

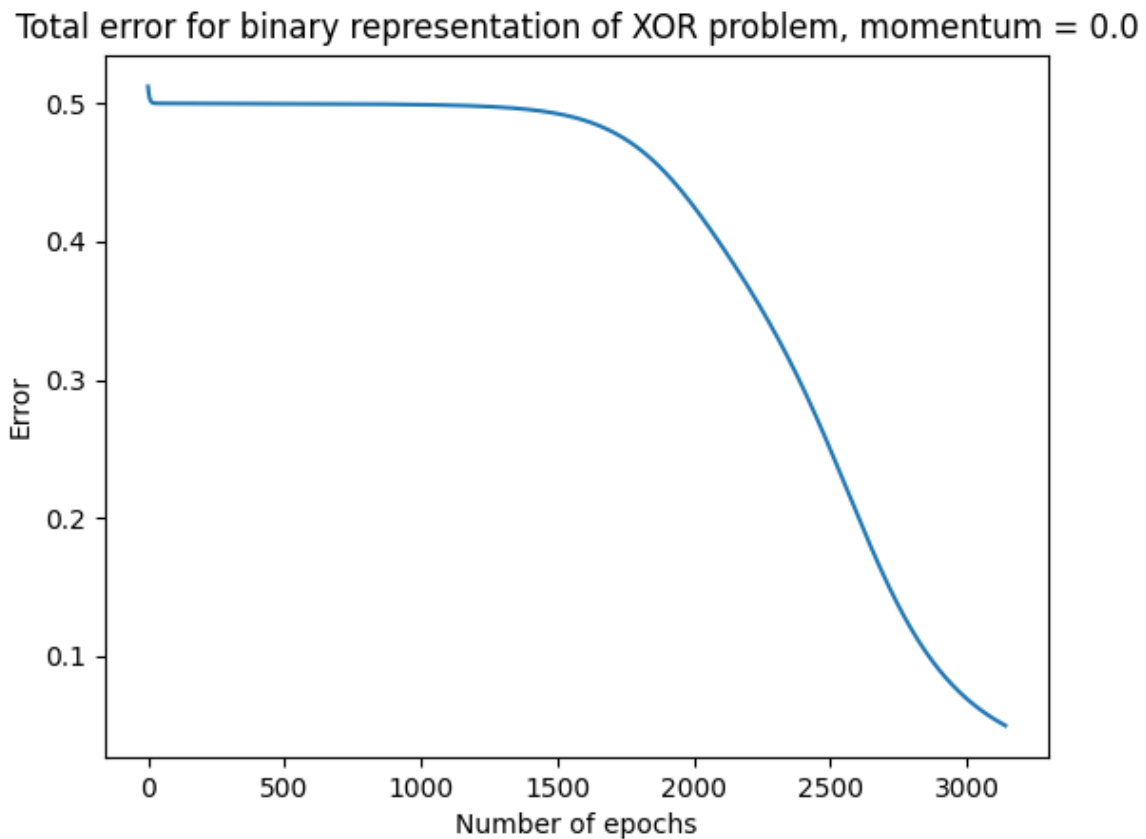
Question 1a.....	1
Question 1b.....	3
Question 1c.....	6
Appendix for source code.....	8

Question 1a

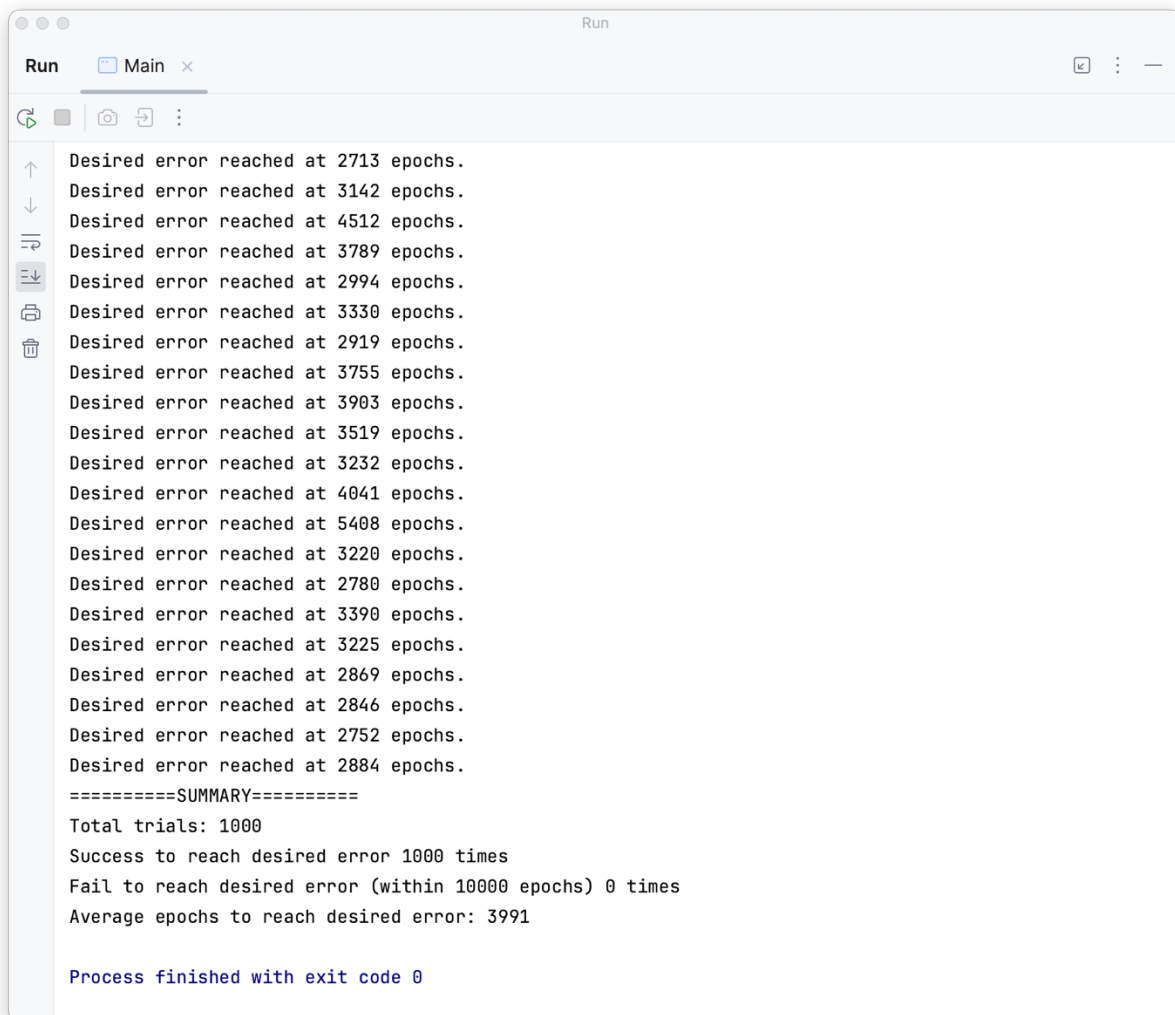
(1) Set up your network in a 2-input, 4-hidden and 1-output configuration. Apply the XOR training set. Initialize weights to random values in the range -0.5 to +0.5 and set the learning rate to 0.2 with momentum at 0.0.

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? You should perform many trials to get your results, although you don't need to plot them all.

This is a graph of total error against number of epochs in one trial, which reaches the desired error less than 0.05 at 3141 epochs.



I performed 1000 trials, and on average 3991 epochs it needs to take to reach a total error of less than 0.05. The printed result is as follows:



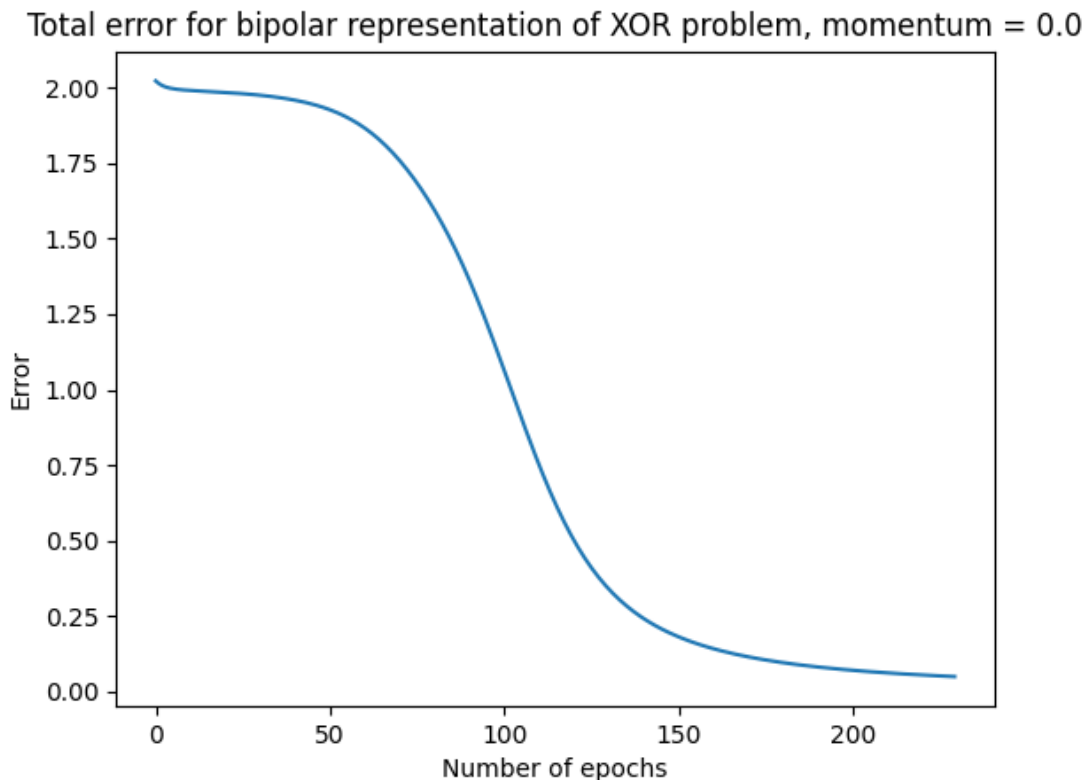
```
Desired error reached at 2713 epochs.
Desired error reached at 3142 epochs.
Desired error reached at 4512 epochs.
Desired error reached at 3789 epochs.
Desired error reached at 2994 epochs.
Desired error reached at 3330 epochs.
Desired error reached at 2919 epochs.
Desired error reached at 3755 epochs.
Desired error reached at 3903 epochs.
Desired error reached at 3519 epochs.
Desired error reached at 3232 epochs.
Desired error reached at 4041 epochs.
Desired error reached at 5408 epochs.
Desired error reached at 3220 epochs.
Desired error reached at 2780 epochs.
Desired error reached at 3390 epochs.
Desired error reached at 3225 epochs.
Desired error reached at 2869 epochs.
Desired error reached at 2846 epochs.
Desired error reached at 2752 epochs.
Desired error reached at 2884 epochs.
=====SUMMARY=====
Total trials: 1000
Success to reach desired error 1000 times
Fail to reach desired error (within 10000 epochs) 0 times
Average epochs to reach desired error: 3991

Process finished with exit code 0
```

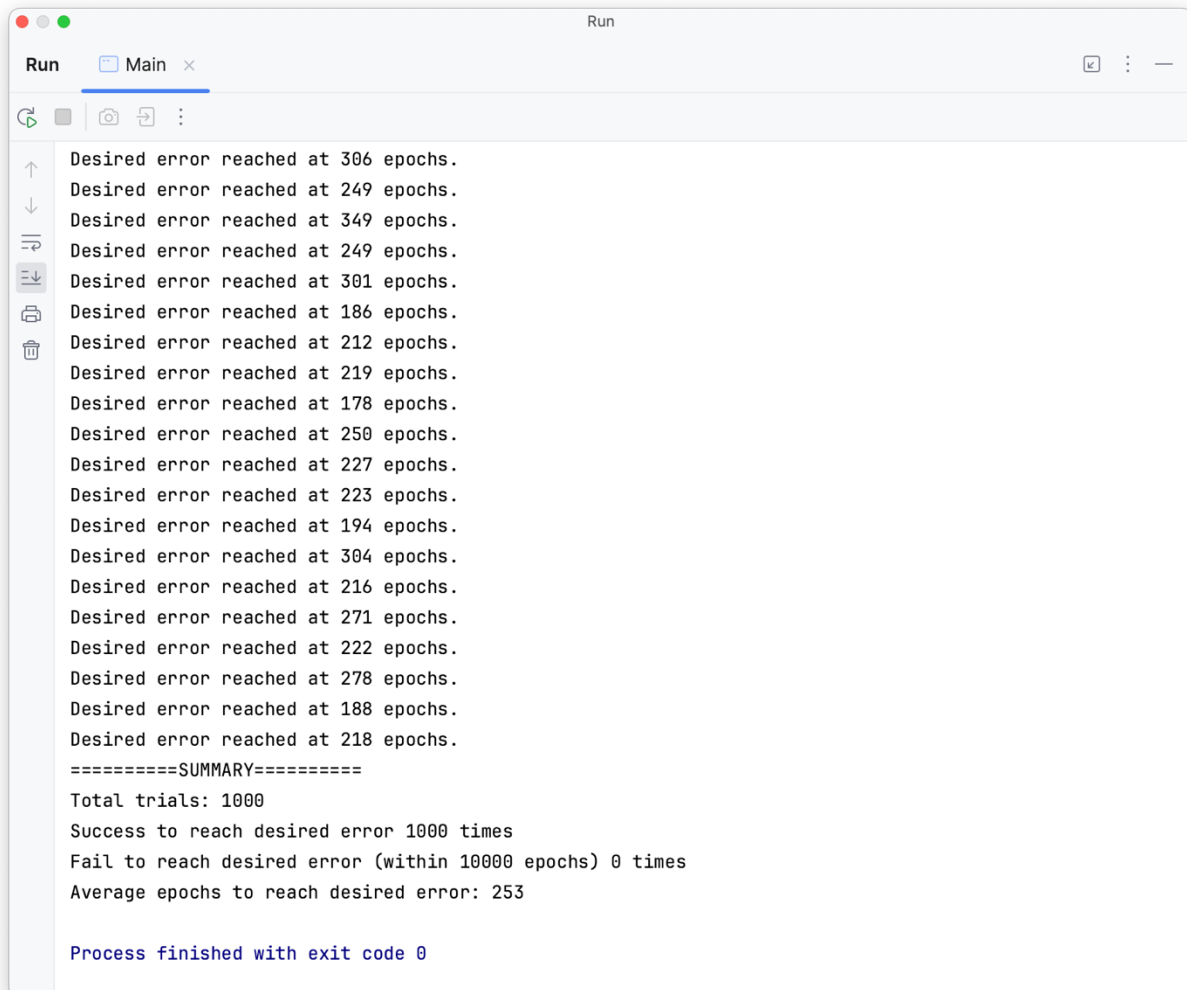
Question 1b

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? If you run into problems, here is some advice from past students that might help: *“We found it interesting that if we update all the δ and then all the weights, we are getting a convergence rate at around 40%. However, if we update the output δ , then the weights in the hidden-to-output layer, then update the δ at the hidden neurons with the just updated hidden-to-output weights, then finally the weights in the input-to-hidden layer -- we will get 100% convergence rate (combined several hundred of trials).”*

This is a graph of total error against number of epochs in one trial, which reaches the desired error less than 0.05 at 229 epochs.



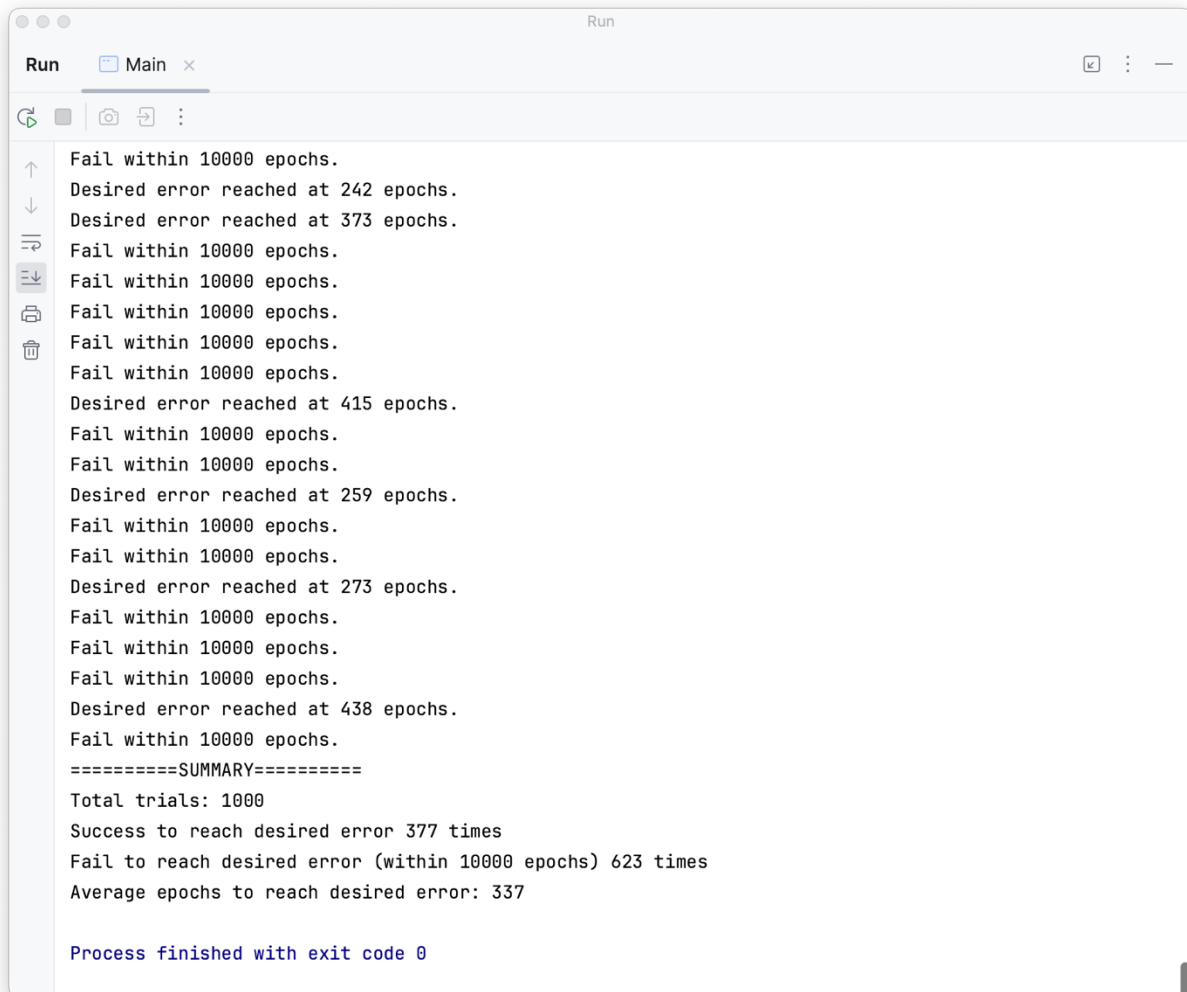
I performed 1000 trials, and on average 253 epochs it needs to take to reach a total error of less than 0.05. The printed result is as follows:



```
Desired error reached at 306 epochs.
Desired error reached at 249 epochs.
Desired error reached at 349 epochs.
Desired error reached at 249 epochs.
Desired error reached at 301 epochs.
Desired error reached at 186 epochs.
Desired error reached at 212 epochs.
Desired error reached at 219 epochs.
Desired error reached at 178 epochs.
Desired error reached at 250 epochs.
Desired error reached at 227 epochs.
Desired error reached at 223 epochs.
Desired error reached at 194 epochs.
Desired error reached at 304 epochs.
Desired error reached at 216 epochs.
Desired error reached at 271 epochs.
Desired error reached at 222 epochs.
Desired error reached at 278 epochs.
Desired error reached at 188 epochs.
Desired error reached at 218 epochs.
=====SUMMARY=====
Total trials: 1000
Success to reach desired error 1000 times
Fail to reach desired error (within 10000 epochs) 0 times
Average epochs to reach desired error: 253

Process finished with exit code 0
```

Note, the above result for bipolar representation is observed using the advice from the question. When we are not using the advice, (i.e., we update the error signal for the output neuron and the hidden neurons, and then update the weights for the output neuron and then the hidden neurons), there is a high failure rate (62.3%) to reach the error less than 0.05 within 10000 epochs. The result is as follows:



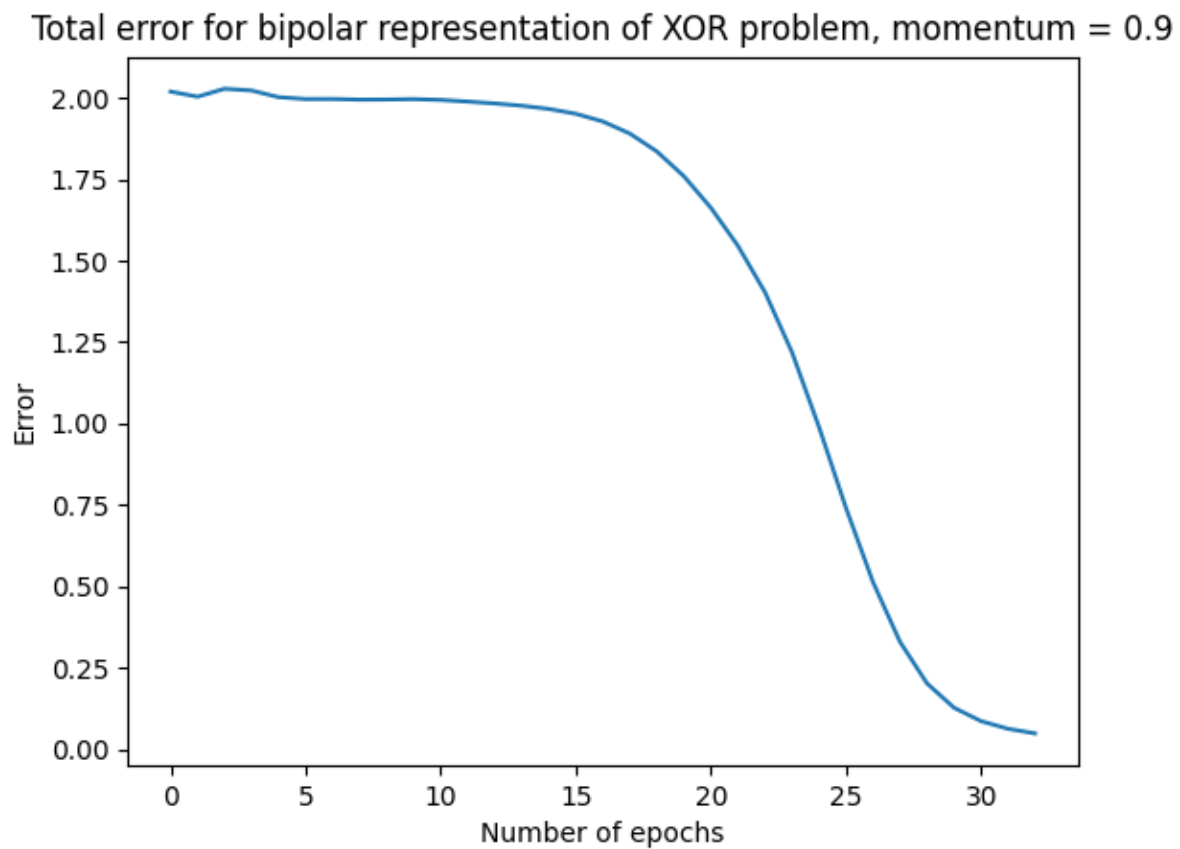
```
Run
Main x
Fail within 10000 epochs.
Desired error reached at 242 epochs.
Desired error reached at 373 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Desired error reached at 415 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Desired error reached at 259 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Desired error reached at 273 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Fail within 10000 epochs.
Desired error reached at 438 epochs.
Fail within 10000 epochs.
=====SUMMARY=====
Total trials: 1000
Success to reach desired error 377 times
Fail to reach desired error (within 10000 epochs) 623 times
Average epochs to reach desired error: 337

Process finished with exit code 0
```

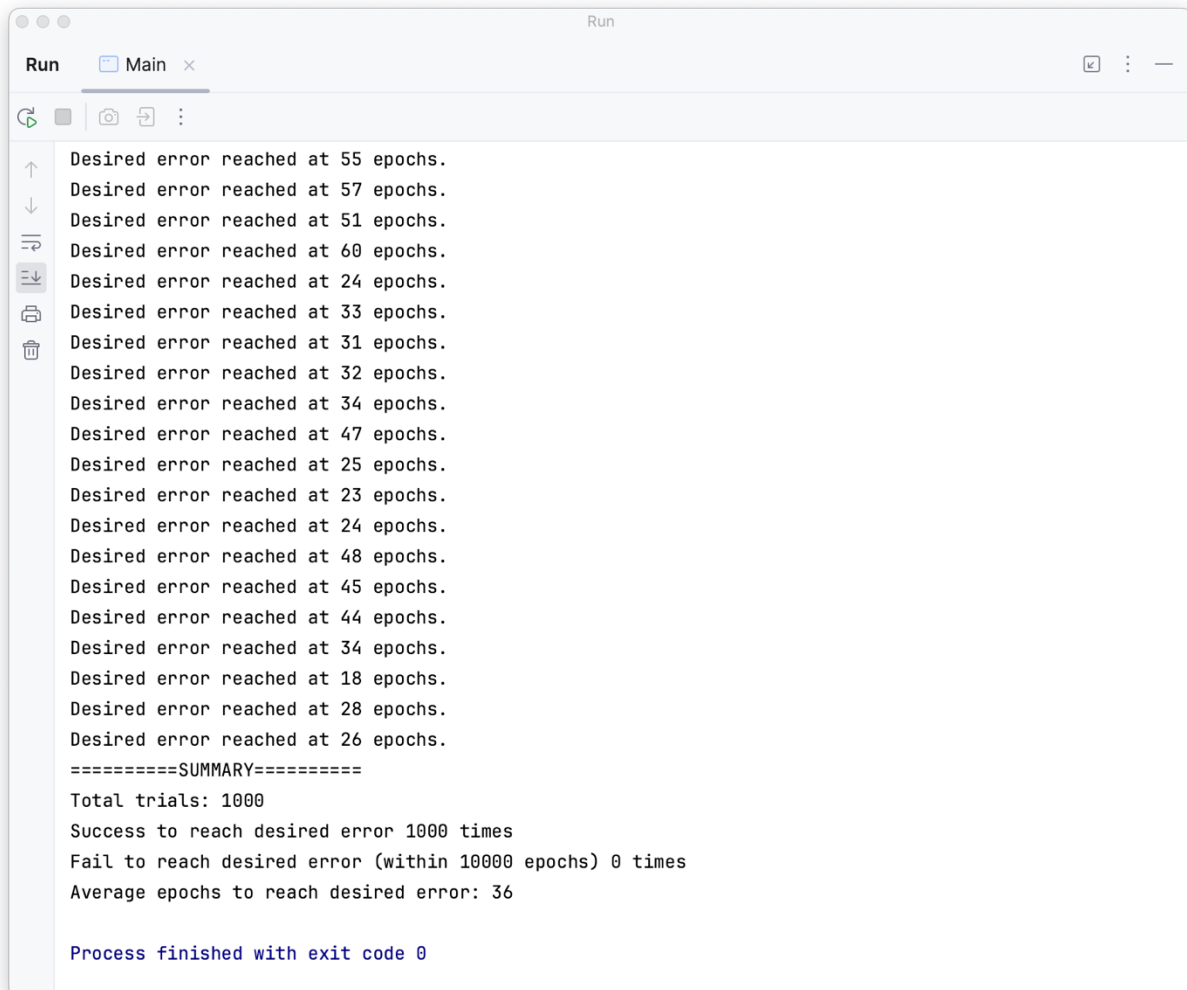
Question 1c

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

This is a graph of total error against number of epochs in one trial, which reaches the desired error less than 0.05 at 32 epochs.



I performed 1000 trials, and on average 36 epochs it needs to take to reach a total error of less than 0.05. The printed result is as follows:



```
Run Main x
Desired error reached at 55 epochs.
Desired error reached at 57 epochs.
Desired error reached at 51 epochs.
Desired error reached at 60 epochs.
Desired error reached at 24 epochs.
Desired error reached at 33 epochs.
Desired error reached at 31 epochs.
Desired error reached at 32 epochs.
Desired error reached at 34 epochs.
Desired error reached at 47 epochs.
Desired error reached at 25 epochs.
Desired error reached at 23 epochs.
Desired error reached at 24 epochs.
Desired error reached at 48 epochs.
Desired error reached at 45 epochs.
Desired error reached at 44 epochs.
Desired error reached at 34 epochs.
Desired error reached at 18 epochs.
Desired error reached at 28 epochs.
Desired error reached at 26 epochs.
=====SUMMARY=====
Total trials: 1000
Success to reach desired error 1000 times
Fail to reach desired error (within 10000 epochs) 0 times
Average epochs to reach desired error: 36

Process finished with exit code 0
```


Appendix for source code

Neuron.java

```
import java.util.Random;
public class Neuron {
    public double argA;
    public double argB;
    public double[] weights;

    public double[] lastWeightsChanges;
    public double output;

    public double weightedSum;

    // error signal
    public double delta;

    final double bias = 1.0;

    public Neuron(int argNumInputs, double argA, double argB){
        // initialize the weights for the neuron (including the bias)
        this.weights = new double[argNumInputs+1];
        this.lastWeightsChanges = new double[weights.length];
        this.argA = argA;
        this.argB = argB;
    }

    public void initializeWeight(){
        for(int i=0; i< weights.length; i++){
            Random random = new Random();
            weights[i] = -0.5 + random.nextDouble();
        }
    }

    public double[] getLastWeightsChanges(){
        return this.lastWeightsChanges;
    }

    public double[] getWeights(){
        return this.weights;
    }

    public void setWeights(double[] newWeights){
        double[] oldWeights = this.weights;
        double[] changes = new double[oldWeights.length];
        if(newWeights.length != this.weights.length){
            System.out.println("newWeights length does not match the weights for this neuron!");
        }
        for(int i = 0; i < oldWeights.length; i++){
            changes[i] = newWeights[i] - oldWeights[i];
        }
        this.weights = newWeights;
        this.lastWeightsChanges = changes;
    }

    public double getWS(){
        return this.weightedSum;
    }

    public void updateOutput(double[] inputs){
        if(inputs.length != this.weights.length-1){
            System.out.println("inputs length don't match the weights for this neuron!");
            System.out.println("inputs length " + inputs.length + " and weights length " + this.weights.length);
        }
        double sum = bias * weights[weights.length-1];
        for (int i = 0; i < inputs.length; i++) {
            sum += inputs[i] * weights[i];
        }
        this.output = customSigmoid(sum);
        this.weightedSum = sum;
    }

    public double getOutput(){
        return this.output;
    }
}
```

```
}

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

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

public void setDelta(double delta){
    this.delta = delta;
}

}
```

NeuralNet.java

```

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

public class NeuralNet implements NerualNetInterface{
    public double argA;
    public double argB;
    public Neuron[] hiddenNeurons;
    public Neuron outputNeuron;
    public double learningRate;
    public double momentumTerm;
    public int backwardPropagationVersion;

    /**
     * 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 NeuralNet(
        int argNumInputs,
        int argNumHidden,
        double argLearningRate,
        double argMomentumTerm,
        double argA,
        double argB,
        int backwardPropagationVersion){
        // initialize variables for the NN
        this.backwardPropagationVersion = backwardPropagationVersion;
        this.argA=argA;
        this.argB=argB;
        this.learningRate = argLearningRate;
        this.momentumTerm = argMomentumTerm;
        // initialize hidden cells
        this.hiddenNeurons = new Neuron[argNumHidden];
        for(int i = 0; i < hiddenNeurons.length; i++){
            hiddenNeurons[i] = new Neuron(argNumInputs, argA, argB);
        }
        // initialize output cell
        this.outputNeuron = new Neuron(hiddenNeurons.length, argA, argB);

        initializeWeights();
    };

    @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;
    }

    public double deri(double x) {
        return (argB-argA) * Math.pow((1+Math.exp(-x)), -2) * Math.exp(-x);
    }

    @Override
    public void initializeWeights() {
        for(int i = 0; i < hiddenNeurons.length; i++){
            hiddenNeurons[i].initializeWeight();
        }
        outputNeuron.initializeWeight();
    }

    @Override
    public void zeroWeights() {
    }

    @Override
    public double outputFor(double[] X) {
        double[] hiddenLayerOutputs = new double[this.hiddenNeurons.length];
        for(int i = 0; i<hiddenLayerOutputs.length; i++){
            hiddenNeurons[i].updateOutput(X);
            hiddenLayerOutputs[i] = hiddenNeurons[i].getOutput();
        }
        outputNeuron.updateOutput(hiddenLayerOutputs);
        return outputNeuron.getOutput();
    }

    /**
     * Given X, update outputs for hidden neurons and output neuron
     * @param X input vector

```

```

    */
    public void forward(double[] X){
        double[] hiddenLayerOutputs = new double[this.hiddenNeurons.length];
        for(int i = 0; i<hiddenNeurons.length; i++){
            hiddenNeurons[i].updateOutput(X);
            hiddenLayerOutputs[i] = hiddenNeurons[i].getOutput();
        }
        outputNeuron.updateOutput(hiddenLayerOutputs);
    }

    /**
     * backward propagation: update delta for output neuron and hidden neurons
     * and update weights for output neuron and hidden neurons
     * (also update the lastWeightsChanges)
     *
     * version 1 and version 2 use different orders
     */
    public void backwardPropagationVersion1(double[] X, double argValue){
        updateErrorSignalOutputN(argValue);
        updateErrorSignalHiddenNs();
        updateWeightsOutputN();
        updateWeightsHiddenNs(X);
    }

    /**
     * backward propagation: update delta for output neuron and hidden neurons
     * and update weights for output neuron and hidden neurons
     * (also update the lastWeightsChanges)
     *
     * version 1 and version 2 use different orders
     */
    public void backwardPropagationVersion2(double[] X, double argValue){
        updateErrorSignalOutputN(argValue);
        updateWeightsOutputN();
        updateErrorSignalHiddenNs();
        updateWeightsHiddenNs(X);
    }

    private void updateErrorSignalOutputN(double argValue) {
        // update and set up the delta for output cell
        double y_o = outputNeuron.getOutput();
        double ws_o = outputNeuron.getWS();
        double deri_o = deri(ws_o);
        double δ_o = (argValue-y_o) * (deri_o);
        outputNeuron.setDelta(δ_o);
    }

    private void updateErrorSignalHiddenNs() {
        // update and set up the delta for hidden cells
        for(int i = 0; i < hiddenNeurons.length; i++){
            double y_hi = hiddenNeurons[i].getOutput();
            double ws_hi = hiddenNeurons[i].getWS();
            double deri_hi = deri(ws_hi);
            double δ_hi = outputNeuron.weights[i] * outputNeuron.getDelta() * (deri_hi);
            hiddenNeurons[i].setDelta(δ_hi);
        }
    }

    private void updateWeightsOutputN() {
        double δ_o = outputNeuron.getDelta();
        // update weights for output cell
        //  $w[i] = w[i] + \eta * \delta_o * y_{hi} + a * \Delta w[i]$ 
        double[] oldW_o = outputNeuron.getWeights();
        double[] newW_o = new double[oldW_o.length];
        double[] Δw_o = outputNeuron.getLastWeightsChanges();
        // update everything but the weight for bias
        for(int i = 0; i < newW_o.length-1; i++){
            double y_hi = hiddenNeurons[i].getOutput();
            newW_o[i] = oldW_o[i] + this.learningRate * δ_o * y_hi + this.momentumTerm * Δw_o[i];
        }
        // update weight for bias
        int indexForBias = newW_o.length-1;
        newW_o[indexForBias] = oldW_o[indexForBias] + this.learningRate * δ_o * bias + this.momentumTerm * Δw_o[indexForBias];
        // set up the weights back
        outputNeuron.setWeights(newW_o);
    }

    private void updateWeightsHiddenNs(double[] X) {
        // for each, update weights for hidden cells
        for(int i = 0; i < hiddenNeurons.length; i++){
            double[] oldW_hi = hiddenNeurons[i].getWeights();
            double[] newW_hi = new double[oldW_hi.length];
            double[] Δw_hi = hiddenNeurons[i].getLastWeightsChanges();
            double δ_hi = hiddenNeurons[i].getDelta();
            // update everything but the weight for bias
            for(int j = 0; j < newW_hi.length-1; j++){
                newW_hi[j] = oldW_hi[j] + this.learningRate * δ_hi * X[j] + this.momentumTerm * Δw_hi[j];
            }
            // update weight for bias
            int indexForBias = newW_hi.length-1;
            newW_hi[indexForBias] = oldW_hi[indexForBias] + this.learningRate * δ_hi * bias + this.momentumTerm * Δw_hi[indexForBias];
            // set up the weights back
            hiddenNeurons[i].setWeights(newW_hi);
        }
    }

```

```

    }
}

@Override
public double train(double[] X, double argValue) {
    /*
     * For each pattern:
     *   using train data and weights to do forward calculation
     *   get error signals through backward propagation
     *   update weights for each cell
     *
     * After training all pattern, check if total error is acceptable,
     * if yes, stop
     * otherwise, repeat training
     */
    forward(X);
    switch(this.backwardPropagationVersion){
        case 1:
            backwardPropagationVersion1(X, argValue);
            break;
        case 2:
            backwardPropagationVersion2(X, argValue);
            break;
    }
    return (0.5 * Math.pow((outputFor(X)- argValue), 2));
}

@Override
public void save(File argFile) {
}

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

/**
 * helper function to print the NN
 */
public void printNN(){
    // System.out.println(outputNeuron.getWeights().length); //5
    // System.out.println(hiddenNeurons.length); // 4
    System.out.println("-----print NN-----");
    System.out.println("output Neuron weights:");
    int i = 0;
    while(i < 5){
        System.out.println(outputNeuron.getWeights()[i]);
        i++;
    }

    i = 0;
    while(i < 4){
        System.out.println("hidden Neuron " + i + " weights:");
        int j = 0;
        while(j<3){
            System.out.println(hiddenNeurons[i].getWeights()[j]);
            j++;
        }
        i++;
    }
    System.out.println("-----end-----");
}
}

```

CommonInterface.java

```
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 {
    /**
     * @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);

    /**
     * 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 NN weights from file. The load must of course
     * have knowledge of how the 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 nerual net whose structure does not match
     * the data in the file. (e.g., wrong number of hidden neurons)
     * @param argFileName
     * @throws IOException
     */
    public void load(String argFileName) throws IOException;
}
```

NerualNetInterface.java

```
public interface NerualNetInterface extends CommonInterface{
    final double bias = 1.0; // The input for each neurons bias weight

    /**
     * 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();
}
```


Main.java

```
public class Main {
    static final double[][] XORInputBinary = {{0.0, 0.0}, {0.0, 1.0}, {1.0, 0.0}, {1.0, 1.0}};
    static final double[] XOROutputBinary = {0.0, 1.0, 1.0, 0.0};

    static final double[][] XORInputBipolar = {{-1.0, -1.0}, {-1.0, 1.0}, {1.0, -1.0}, {1.0, 1.0}};
    static final double[] XOROutputBipolar = {-1.0, 1.0, 1.0, -1.0};

    public static void main(String[] args) {
        Q1a();
        Q1b();
        Q1c();
    }

    public static void Q1a() {
        int totalTrials = 1000;
        int numOfConverge=0, numOfFail = 0;
        int totalEpochsToConverge = 0;

        for (int trial=0; trial < totalTrials; trial++) {
            NeuralNet XORNNBinary = new NeuralNet(2, 4, 0.2, 0.0, 0, 1, 2);

            double errs[] = new double[4];
            int numEpoch = 0;
            double totalErr;
            do {
                totalErr = 0;
                // train all pattern once
                for (int i = 0; i < 4; i++) {
                    errs[i] = XORNNBinary.train(XORInputBinary[i], XOROutputBinary[i]);
                    totalErr += errs[i];
                }

                //after one epoch, calculate the total error
                for (int i = 0; i < 4; i++) {
                    double output = XORNNBinary.outputFor(XORInputBinary[i]);
                    totalErr += Math.pow((output - XOROutputBinary[i]), 2);
                }
                totalErr *= 0.5;
                System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
                numEpoch++;

                if(totalErr < 0.05){
                    numOfConverge++;
                    totalEpochsToConverge += numEpoch;
                    System.out.println("Desired error reached at " + numEpoch + " epochs.");
                    break;
                }
            } while (numEpoch < 10000);

            if(numEpoch >= 10000){
                System.out.println("Fail within " + numEpoch + " epochs.");
                numOfFail++;
            }
        }
        System.out.println("=====SUMMARY=====");
        System.out.println("Total trials: " + totalTrials);
        System.out.println("Success to reach desired error " + numOfConverge + " times");
        System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
        System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge/numOfConverge);
    }

    public static void Q1b() {
        int totalTrials = 1000;
        int numOfConverge=0, numOfFail = 0;
        int totalEpochsToConverge = 0;

        for (int trial=0; trial < totalTrials; trial++) {
            NeuralNet XORNNBinary = new NeuralNet(2, 4, 0.2, 0.0, -1, 1, 2);
        }
    }
}
```

```

double errs[] = new double[4];
int numEpoch = 0;
double totalErr;
do {
    totalErr = 0;
    // train all pattern once
    for (int i = 0; i < 4; i++) {
        errs[i] = XORNNBinary.train(XORInputBipolar[i], XOROutputBipolar[i]);
        totalErr += errs[i];
    }

    //after one epoch, calculate the total error
    for (int i = 0; i < 4; i++) {
        double output = XORNNBinary.outputFor(XORInputBipolar[i]);
        totalErr += Math.pow((output - XOROutputBipolar[i]), 2);
    }
    totalErr *= 0.5;
    System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
    numEpoch++;

    if(totalErr < 0.05){
        numOfConverge++;
        totalEpochsToConverge += numEpoch;
        System.out.println("Desired error reached at " + numEpoch + " epochs.");
        break;
    }

} while (numEpoch < 10000);

if(numEpoch >= 10000){
    System.out.println("Fail within " + numEpoch + " epochs.");
    numOfFail++;
}
}
System.out.println("=====SUMMARY=====");
System.out.println("Total trials: " + totalTrials);
System.out.println("Success to reach desired error " + numOfConverge + " times");
System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge/numOfConverge);
}

public static void Q1c() {
    int totalTrials = 1000;
    int numOfConverge=0, numOfFail = 0;
    int totalEpochsToConverge = 0;

    for (int trial=0; trial < totalTrials; trial++) {
        NeuralNet XORNNBinary = new NeuralNet(2, 4, 0.2, 0.9, -1, 1, 2);

        double errs[] = new double[4];
        int numEpoch = 0;
        double totalErr;
        do {
            totalErr = 0;
            // train all pattern once
            for (int i = 0; i < 4; i++) {
                errs[i] = XORNNBinary.train(XORInputBipolar[i], XOROutputBipolar[i]);
                totalErr += errs[i];
            }

            //after one epoch, calculate the total error
            for (int i = 0; i < 4; i++) {
                double output = XORNNBinary.outputFor(XORInputBipolar[i]);
                totalErr += Math.pow((output - XOROutputBipolar[i]), 2);
            }
            totalErr *= 0.5;
            System.out.println("Total error " + totalErr + " at " + numEpoch + " epochs.");
            numEpoch++;

            if(totalErr < 0.05){
                numOfConverge++;
                totalEpochsToConverge += numEpoch;
                System.out.println("Desired error reached at " + numEpoch + " epochs.");
                break;
            }
        }
    }
}

```

```

    }

    } while (numEpoch < 10000);

    if(numEpoch >= 10000){
        System.out.println("Fail within " + numEpoch + " epochs.");
        numOfFail++;
    }
}
System.out.println("=====SUMMARY=====");
System.out.println("Total trials: " + totalTrials);
System.out.println("Success to reach desired error " + numOfConverge + " times");
System.out.println("Fail to reach desired error (within 10000 epochs) " + numOfFail + " times");
System.out.println("Average epochs to reach desired error: " + totalEpochsToConverge/numOfConverge);
}
}

```