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.

Figure 1. Total Error against epoch (x-axis: epochs, y-axis: total error, using Binary representation, with Learning rate=0.2 and Momentum=0.0)

In order to get the average epochs for the case, we use 400 trials in one group and conducted 4 groups to see the trend. In order to prevent the case that too many epochs happened in one round, we set the epochs upper bound as 10000. The statistics is shown in the following table. In this table, the second column shows the number of trial groups, the third column shows average epochs taken in these 400 trials, following are the minimum epochs taken and maximum taken. In the last column is the number of trials that goes beyond 10000.

The average epochs taken is 3584 for this case. Figure 1 shows a typical Error rate against epoch pattern.



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?

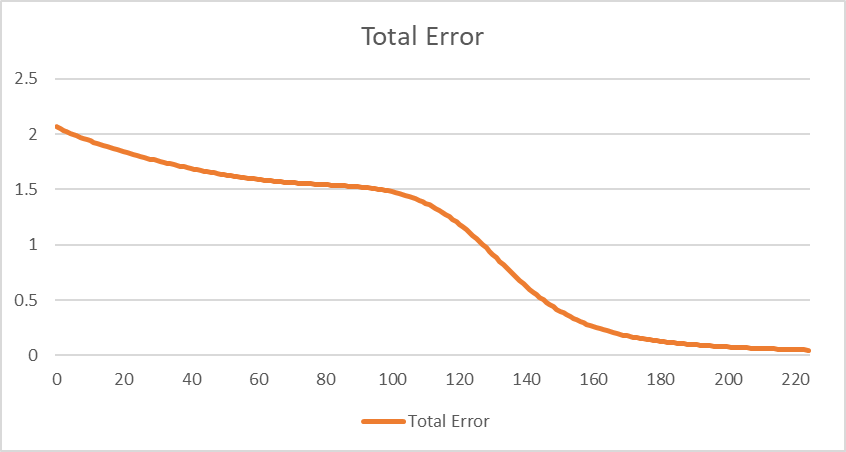


Figure 2. Total Error against epoch (x-axis: epochs, y-axis: total error, using Bipolar representation, with Learning rate=0.2 and Momentum=0.0)

The average epochs in 4 groups are shown in the following table. The total average is 342 for this case. Figure 2 shows a typical Error rate against epoch pattern. Through observation of the tests, it can be seen that sometimes the error rate oscillates and results in high epochs.



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

Here we conducted trials with momentum of 0.9 of both binary and bipolar representation. The first graph shows the binary case and the second graph shows the bipolar case under 0.9 momentum. As shown in the figure, both got much faster than a 0.0 momentum (almost 10 times faster).

Figure 3. Total Error against epoch (x-axis: epoch, y-axis: error rate, Binary, Learning rate=0.2 and Momentum=0.9)

The following table shows the average epochs taken for this case. In general, it takes 364 epochs to reach the required error.



Figure 4. Total against epoch (x-axis: epoch, y-axis: error rate, Bipolar, Learning rate=0.2 and Momentum=0.9)

The following table shows the average epochs taken for this case. In general, it takes 49 epochs to reach the required error.



**Appendix:**

( CommonInterface and NeuralNetInterface not included)

1. **Main.java**

**package** xor;  
  
**import** java.io.BufferedWriter;  
**import** java.io.File;  
**import** java.io.FileWriter;  
**import** java.io.IOException;  
  
**public class** main {  
 **public static void** main(String[] args) {  
*// NeurualNetWork neurualNetWork = new NeurualNetWork(2, 4, 1, 0.2, 0.0, -0.5,0.5);  
// neurualNetWork.initializeWeights();  
// double[][] input = {{1,1},{1,0},{0,1},{0,0}};  
// double[][] input2 = {{1,1},{1,-1},{-1,1},{-1,-1}};  
// double[] output = {0,1,1,0};  
// double[] output2 = {-1,1,1,-1};  
// double error = Double.MAX\_VALUE;  
// int epoch = 0, counter = 0;* File outFile = **new** File(**"output.txt"**);  
 **try** {  
 **boolean** createNewFile = outFile.createNewFile();  
 BufferedWriter out = **new** BufferedWriter(**new** FileWriter(outFile));  
 *//put the epoch number and total error into the output file  
// while(error >= 0.05){  
// error = 0;  
// for(int i = 0; i < 4; i++){  
// error += neurualNetWork.train(input[i],output[i]);  
// }  
// epoch++;  
// out.write(String.valueOf(error));  
// out.write("\r\n");  
// }* **double** avg = 0, beyond\_bound = 0, min\_epoch = 10000, max\_epoch = 0;  
 **for**(**int** k = 0; k < 400; k++) {  
 NeurualNetWork neurualNetWork = **new** NeurualNetWork(2, 4, 1, 0.2, 0.9, -0.5,0.5);  
 neurualNetWork.initializeWeights();  
 **double**[][] input = {{1,1},{1,0},{0,1},{0,0}};  
 **double**[][] input2 = {{1,1},{1,-1},{-1,1},{-1,-1}};  
 **double**[] output = {0,1,1,0};  
 **double**[] output2 = {-1,1,1,-1};  
 **double** error = Double.***MAX\_VALUE***;  
 **int** epoch = 0, counter = 0;  
  
 **for** (epoch = 0; epoch < 10000; epoch++) {  
 error = 0;  
 **for** (**int** i = 0; i < 4; i++) {  
 error += neurualNetWork.train(input2[i], output2[i]);  
 }  
*// out.write(String.valueOf(error));  
// out.write("\r\n");* **if** (error < 0.05){  
 **if**(epoch<min\_epoch){  
 min\_epoch = epoch;  
 }  
 **if**(epoch>max\_epoch){  
 max\_epoch = epoch;  
 }  
 avg+=epoch;  
 **break**;  
 }  
 **if**(epoch==9999){  
 beyond\_bound++;  
 }  
 }  
  
  
 }  
 avg /=400;  
 out.write(String.*valueOf*(avg));  
 out.write(**"\r\n"**);  
 out.write(String.*valueOf*(min\_epoch));  
 out.write(**"\r\n"**);  
 out.write(String.*valueOf*(max\_epoch));  
 out.write(**"\r\n"**);  
 out.write(String.*valueOf*(beyond\_bound));  
 out.write(**"\r\n"**);  
 out.flush();  
 out.close();  
 } **catch** (IOException e){  
 System.***out***.println(**"IOException"**);  
 }  
  
  
 }  
  
}

1. **NeurualNetWork.java**

**package** xor;  
  
**import** java.io.BufferedWriter;  
**import** java.io.File;  
**import** java.io.FileWriter;  
**import** java.io.IOException;  
**import** java.util.Random;  
  
**public class** NeurualNetWork **implements** NeuralNetInterface {  
  
 **private double NeuronCell**[][];  
 **private double S**[][];  
 **private double Weight**[][][]; *//Weight on every layer* **private double WeightChange**[][][];  
 **private double layerError**[][];  
  
 **private int argNumInputs**;  
 **private int argNumHidden**;  
 **private int argNumOutputs**;  
 **private double argLearningRate**;  
 **private double argMomentumTerm**;  
 **private double argA**;  
 **private double argB**;  
  
 **private double WeightSum**;  
  
 */\*\*  
 \* 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 argNumOutput*** *Number of output neurons  
 \** ***@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** NeurualNetWork(  
 **int** argNumInputs,  
 **int** argNumHidden,  
 **int** argNumOutput,  
 **double** argLearningRate,  
 **double** argMomentumTerm,  
 **double** argA,  
 **double** argB ){  
  
 **this**.**argNumInputs** = argNumInputs;  
 **this**.**argNumHidden** = argNumHidden;  
 **this**.**argNumOutputs** = argNumOutput;  
 **this**.**argLearningRate** = argLearningRate;  
 **this**.**argMomentumTerm** = argMomentumTerm;  
 **this**.**argA** = argA;  
 **this**.**argB** = argB;  
  
 **NeuronCell** = **new double**[3][];  
 **NeuronCell**[0] = **new double**[3];  
 **NeuronCell**[1] = **new double**[5];  
 **NeuronCell**[2] = **new double**[1];  
 *//bias term* **NeuronCell**[0][2] = 1;  
 **S** = **new double**[3][];  
 **S**[0] = **new double**[3];  
 **S**[1] = **new double**[5];  
 **S**[2] = **new double**[1];  
 **S**[0][2] = 1;  
 **layerError**=**new double**[3][];  
 **layerError**[0]=**new double**[3];  
 **layerError**[1]=**new double**[5];  
 **layerError**[2]=**new double**[1];  
 **Weight**=**new double**[2][][];  
 **Weight**[0]=**new double**[3][4];  
 *//don't need weight to bias term in next layer* **Weight**[1]=**new double**[5][1];  
 **WeightChange**=**new double**[2][][];  
 **WeightChange**[0]=**new double**[3][4];  
 **WeightChange**[1]=**new double**[5][1];  
 **WeightSum**=0;  
 }  
 @Override  
 */\*\*  
 \* 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) {  
 **return** 1/(1+Math.*exp*(-x));  
 }  
  
 @Override  
 */\*\*  
 \* 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  
 \*/  
 // may not need this here* **public double** customSigmoid(**double** x) {  
 **return** (2.0/(1+Math.*exp*(-x)) - 1.0);  
 }  
  
 @Override  
 */\*\*  
 \* 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() {  
 Random random = **new** Random();  
 *//weight from input x[i] to hidden layer h[j]* **for**(**int** i=0;i<**argNumInputs**;i++){  
 *//i<=argNumInputs for bias?* **for**(**int** j=0;j<**argNumHidden**;j++){  
 **Weight**[0][i][j]=random.nextDouble() + **argA**;  
 *//random value from -0.5 to 0.5* }  
 }  
  
 **for**(**int** i=0;i<**argNumHidden**;i++){  
 **for**(**int** j = 0; j< **argNumOutputs**; ++j){  
 **Weight**[1][i][j]=random.nextDouble() + **argA**;  
 }  
 }  
 }  
  
 @Override  
 */\*\*  
 \* Initialize the weights to 0.  
 \*/* **public void** zeroWeights() {  
 **for**(**int** i=0;i<=**argNumInputs**;i++){  
 *//i<=argNumInputs for bias?* **for**(**int** j=0;j<=**argNumHidden**;j++){  
 **Weight**[0][i][j] = 0;  
 *//random value from -0.5 to 0.5* }  
 }  
  
 **for**(**int** i=0;i<=**argNumHidden**;i++){  
 **for**(**int** j = 0; j< **argNumOutputs**; ++j){  
 **Weight**[1][i][j] = 0;  
 }  
 }  
 }  
  
 @Override  
 */\*\*  
 \** ***@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) {  
 *//step forward  
 //first get initial input* **int** i = 0, j = 0;  
 **for**(i=0;i<**argNumInputs**;i++){  
 **S**[0][i] = X[i];  
 **NeuronCell**[0][i] = X[i];  
 }  
 *//then add the bias term* **S**[0][**argNumInputs**] = 1;  
 **NeuronCell**[0][**argNumInputs**] = customSigmoid(**S**[0][**argNumInputs**]);  
 *//NeuronCell[0][argNumInputs] = sigmoid(S[0][argNumInputs]);* **S**[1][**argNumHidden**] = 1;  
 **NeuronCell**[1][**argNumHidden**] = customSigmoid(**S**[1][**argNumInputs**]);  
 *//NeuronCell[1][argNumHidden] = sigmoid(S[1][argNumHidden]);  
// NeuronCell[0][argNumInputs] = sigmoid(S[0][argNumInputs]);  
// NeuronCell[1][argNumHidden] = sigmoid(S[1][argNumHidden]);* **for**(i=0;i<**argNumHidden**;i++){  
 **for**(j=0;j<=**argNumInputs**;j++){  
 *//Wji : weigth from j to i* **WeightSum**+=**NeuronCell**[0][j]\***Weight**[0][j][i];  
 }  
 *//Sj = sigma(Wji \* Xi)* **S**[1][i]=**WeightSum**;  
 **NeuronCell**[1][i]=(customSigmoid(**WeightSum**));  
 *//reset weigthsum* **WeightSum**=0;  
 }  
  
 **for**(i = 0; i < **argNumOutputs**; i++){  
 **for**(j = 0;j <= **argNumHidden**;j++){  
 **WeightSum** += **NeuronCell**[1][j] \* **Weight**[1][j][i];  
 }  
 **NeuronCell**[2][i]=customSigmoid(**WeightSum**);  
 **S**[2][i]=**WeightSum**;  
 **WeightSum**=0;  
 }  
 *//if we only return 1 double, it means we only have one output, so actually we can write return NeuronCell[2][0]* **return NeuronCell**[2][0];  
 }  
  
 @Override  
 */\*\*  
 \* 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) {  
  
 outputFor(X);  
  
 *// for output unit  
 // Ei = (Ci - yi)\*yi\*(1-yi)* **for**(**int** i=0;i<**argNumOutputs**;i++){  
 **for**(**int** j=0;j<**argNumHidden**;++j){  
 *//layerError[2][i] = (argValue-NeuronCell[2][i])\*NeuronCell[2][i]\*(1-NeuronCell[2][i]);* **layerError**[2][i] = (argValue-**NeuronCell**[2][i])\*0.5\*(1-**NeuronCell**[2][i]\***NeuronCell**[2][i]);  
 }  
 }  
  
 *// for hidden unit  
 // Ei = Sigma{Whi \* Eh \* yi \* (1-yi)}* **for**(**int** h = 0; h < **argNumOutputs**; h++) {  
 **for** (**int** i = 0; i < **argNumHidden**; i++) {  
 *//layerError[1][i] = Weight[1][i][h] \* layerError[2][h] \* NeuronCell[1][i] \* (1 - NeuronCell[1][i]);* **layerError**[1][i] = **Weight**[1][i][h] \* **layerError**[2][h] \* 0.5 \* (1 - **NeuronCell**[1][i]\* **NeuronCell**[1][i]);  
 }  
 }  
  
 **double** temp=0;  
 *//Ej = Yj \* (1-Yj) \* Sigma(Eh \* Whj)  
 //We first calculate Sigma(Eh \* Whj) term and store it into temp* **for**(**int** h = 0; h < **argNumOutputs**; h++) {  
 **for** (**int** j = 0; j < **argNumHidden**; j++) {  
 temp += **Weight**[0][2][j] \* **layerError**[1][j];  
 }  
 }  
 *//layerError[0][2]=temp\*(1-NeuronCell[0][2])\*NeuronCell[0][2];* **layerError**[0][2]=temp\*0.5\*(1-**NeuronCell**[0][2]\***NeuronCell**[0][2]);  
  
 *//now we need to change the weight: Wji\* = Wji + (Learning Rate)\*(Layer Error)\*(Xi)* **double** Wji\_With\_Momentum=0;  
 *///wight change* **for**(**int** j=0;j<1;j++) {  
 **for** (**int** i = 0; i <= **argNumHidden**; i++) {  
 *//Wji\** Wji\_With\_Momentum = **Weight**[1][i][j] + **argMomentumTerm** \* **WeightChange**[1][i][j] + **argLearningRate** \* **layerError**[2][j] \* **NeuronCell**[1][i];  
 **WeightChange**[1][i][j] = Wji\_With\_Momentum - **Weight**[1][i][j];  
 **Weight**[1][i][j] = Wji\_With\_Momentum;  
 }  
 }  
  
 **for**(**int** i=0;i<=**argNumInputs**;i++){  
 **for**(**int** j=0;j<**argNumHidden**;j++){  
 Wji\_With\_Momentum=**Weight**[0][i][j]+**argMomentumTerm**\***WeightChange**[0][i][j]+**argLearningRate**\***layerError**[1][j]\***NeuronCell**[0][i];  
 **WeightChange**[0][i][j]=Wji\_With\_Momentum-**Weight**[0][i][j];  
 **Weight**[0][i][j]=Wji\_With\_Momentum;  
  
 }  
 }  
 **double** error = 0.5 \* Math.*pow*(**NeuronCell**[2][0]-argValue, 2);  
 **return** error;  
 }  
  
 @Override  
 */\*\*  
 \* A method to write either a LUT or weights of an neural net to a file.  
 \** ***@param argFile*** *of type File.  
 \*/* **public void** save(File argFile) {  
 File output = **new** File(**"output.txt"**);  
 **try** {  
 **boolean** createNewFile = output.createNewFile();  
 BufferedWriter out = **new** BufferedWriter(**new** FileWriter(output));  
 *//put the epoch number and total error into the output file* } **catch** (IOException e){  
 System.***out***.println(**"IOException"**);  
 }  
  
 }  
  
 @Override  
 */\*\*  
 \* 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).  
 \** ***@param argFileName*** *\** ***@throws*** *IOException  
 \*/* **public void** load(String argFileName) **throws** IOException {  
  
 }  
}