***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***

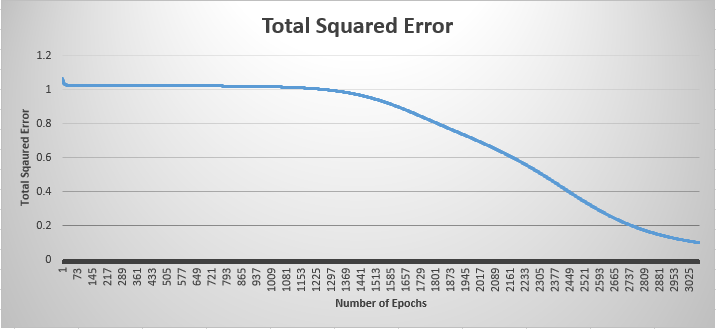
***error of less than 0.05? You should perform many trials to get your results, although you***

***don’t need to plot them all.***

Binary Representation of XOR Problem:

|  |  |  |
| --- | --- | --- |
| **Input A** | **Input B** | **Output** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

On an average, it takes around **3200** epochs to converge to a total error of less than 0.05. Maximum epochs required was about **4912** while the minimum was about **2800** epochs. Total error is computed by adding square of error (i.e. difference of activated output and required output) in each epoch. This implementation was using binary training pair and a binary sigmoid activation function. Below is the graph of # of epochs vs Total error:



**1 a) Graph of Binary Sigmoid based Neural Net Implementation with Learning rate – 0.2 and 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***

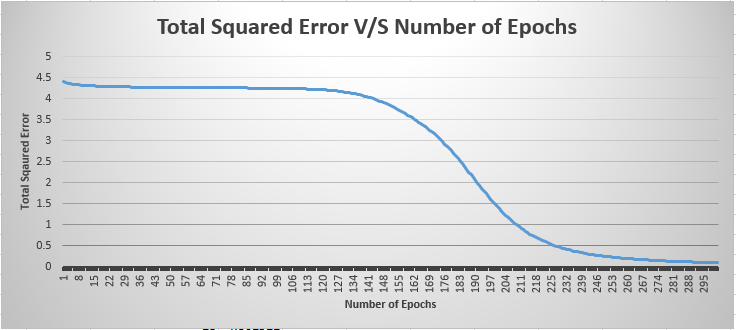
***of less than 0.05?***

Bipolar Representation of XOR Problem:

|  |  |  |
| --- | --- | --- |
| **Input A** | **Input B** | **Output** |
| -1 | -1 | -1 |
| -1 | 1 | 1 |
| 1 | -1 | 1 |
| 1 | 1 | -1 |

On an average, it takes around **300** epochs to converge to a total error of less than 0.05. Maximum epochs required was about **530** while the minimum was about **220** epochs. Total error is computed by adding square of error (i.e. difference of activated output and required output) in each epoch.

The implementation still shows a non-convergence rate of about 15%, i.e., the neural net doesn’t converge to 0.05 error and instead keeps fluctuating around local minima, hence to solve this the initial weights are randomised between [-1 ,1] rather than the required [-0.5,0.5]. This implementation was using bipolar training pair and a bipolar sigmoid activation function. Below is the graph of # of epochs vs Total error:



**1 b) Graph of Bipolar Sigmoid based Neural Net Implementation with Learning rate – 0.2 and momentum – 0.0**

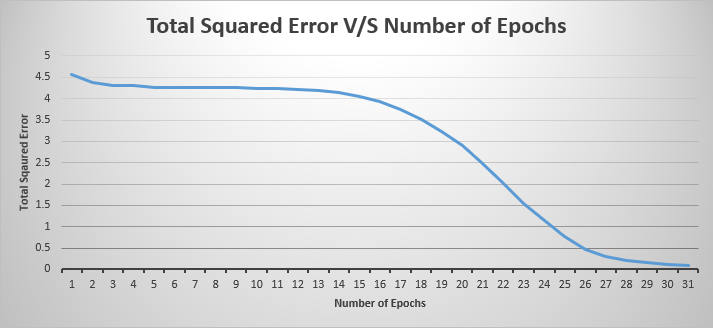
***c) Now set the momentum to 0.9. What does the graph look like now and how fast can 0.05***

***be reached?***

The below implementation involves the use of a tweaking parameter called momentum, this has a range of 0 to 1. This parameter is used to define the amount of influence the past weight change will have on the current weight changes. A value of 0 makes the current change independent of past weight changes where as a value of 1 means the current weight change is completely dependent on previous weight changes.

This implementation also uses a bipolar representation and a bipolar sigmoid along with a learning rate of 0.2, along with a momentum of 0.9.

On an average, it takes around **30** epochs to converge to a total error of less than 0.05. Maximum epochs required was about **49** while the minimum was about **25** epochs. Total error is computed by adding square of error (i.e. difference of activated output and required output) in each epoch. This implementation was using bipolar training pair and a bipolar sigmoid activation function. Below is the graph of # of epochs vs Total error as computed above:



**1 c) Graph of Bipolar Sigmoid based Neural Net Implementation with Learning rate – 0.2 and momentum – 0.9**

A brief description about my implementation is that the algorithm used here is of back propagation to train a 2 input, 4 hidden and 1 output neural net. It uses online weight update that is, the weights are calculated and updated after each input pair is presented to the network. There is also a batch update in which one updates the weights only after all the training pairs have been presented, i.e., only at the end of an epoch.

Please find below the source code for the above implementation.

Note – Changing the value of **minvalue [0, -1]** in the main function, changes the run mode from binary [0] to bipolar [-1] and vice versa. Also change the value of **runMode [‘A’ – binary, ‘B’ – bipolar, ‘C’ – With momentum]** variable to output different files for different implementations.

Appendix –

Source Code for XOR Neural Net. Uses package named **xorpackage**

/\*\*\*\*\*\*\*\*\*\*\*\*\*Package details \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

package xorpackage;

/\*\*\*\*\*\*\*\*\*\*\* Used to write to a CSV File \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileWriter;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.io.IOException;

import java.util.Random;

public class My\_XOR\_NN {

private void InitializeInputs(double min, double max){

/\*\*\*\*\*\*\*\*\* Initializes the Input Patterns \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

inputpattern = new double[numberofpattern][patterndimension ];

inputpattern[0][0] = min;

inputpattern[0][1] = max;

//inputpattern[0][2] = 0;

inputpattern[1][0] = max;

inputpattern[1][1] = min;

//inputpattern[1][2] = 1;

inputpattern[2][0] = max;

inputpattern[2][1] = max;

//inputpattern[2][2] = 0;

inputpattern[3][0] = min;

inputpattern[3][1] = min;

//inputpattern[3][2] = 1;

}

private void InitializeOutputs(double min,double max){

/\*\*\*\*\*\*\*\*\* Initializes the Output Pattern \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

targetpattern = new double[numberofpattern];

targetpattern[0] = max;

targetpattern[1] = max;

targetpattern[2] = min;

targetpattern[3] = min;

}

private void GenerateNeuralNet(){

/\*\*\*\*\*\*\*\*\* Creates a neural net based on required values \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

numberofinputneuron = patterndimension;

numberofoutputneuron = targetdimension;

hiddenNeuronsAndBias = new double[numberofhiddenlayer][numberofhiddenneuron + 1]; //Contains values for hidden layer

hiddenNeuronsAndBias[0][0] = 1; //extra biased neuron

backpropagationhiddenlayer = new double[numberofhiddenlayer][numberofhiddenneuron + 1];

backpropagationhiddenlayer[0][0] = 1; //extra biased neuron

errorhidden = new double[numberofhiddenlayer][numberofhiddenneuron + 1];

outputNeuronAndBias = new double[numberofoutputneuron]; //contains bias for output layer

outputerror = new double[numberofoutputneuron];

activatedhiddenlayer = new double[numberofhiddenlayer][numberofhiddenneuron + 1];

activatedoutputlayer = new double[numberofoutputneuron];

patternerror = new double[numberofpattern];

currentweights = new double[2][][];

currentweights[0] = new double[numberofinputneuron+1][numberofhiddenneuron]; //currentweights from input to hidden

currentweights[1] = new double[numberofhiddenneuron+1][numberofoutputneuron]; //currentweights from hidden to output

deltaweights = new double[2][][];

deltaweights[0] = new double[numberofinputneuron+1][numberofhiddenneuron];

deltaweights[1] = new double[numberofhiddenneuron+1][numberofoutputneuron];

previousdeltaweights = new double[2][][];

previousdeltaweights[0] = new double[numberofinputneuron+1][numberofhiddenneuron];

previousdeltaweights[1] = new double[numberofhiddenneuron+1][numberofoutputneuron];

deltaWeightforMomentum = new double[2][][];

deltaWeightforMomentum[0] = new double[numberofinputneuron+1][numberofhiddenneuron];

deltaWeightforMomentum[1] = new double[numberofhiddenneuron+1][numberofoutputneuron];

EPOCH = new double[maxiteration];

/\*

System.out.println("InputToHiddenWeight:");

for(int i=0;i<=numberofinputneuron;i++){

for(int j=0;j<numberofhiddenneuron;j++){

currentweights[0][i][j] = RandomNumberGenerator();

System.out.println(currentweights[0][i][j]);

}

}

System.out.println("HiddenToOutputWeight:");

for(int i=0;i<=numberofhiddenneuron;i++){

for(int j=0;j<numberofoutputneuron;j++){

currentweights[1][i][j] = RandomNumberGenerator();

System.out.println(currentweights[1][i][j]);

}

}

}

currentweights[0][0][0] = -0.3378; //Weight from bias to hidden 1

currentweights[0][0][1] = 0.2271; //Weight from bias to hidden 2

currentweights[0][0][2] = 0.2859; //Weight from bias to hidden 3

currentweights[0][0][3] = -0.3329; //Weight from bias to hidden 4

currentweights[0][1][0] = 0.197; //Weight from input 1 to hidden 1

currentweights[0][1][1] = 0.3191; //Weight from input 1 to hidden 2

currentweights[0][1][2] = -0.114; //Weight from input 1 to hidden 3

currentweights[0][1][3] = 0.3594; //Weight from input 1 to hidden 4

currentweights[0][2][0] = 0.3099; //Weight from input 2 to hidden 1

currentweights[0][2][1] = 0.1904; //Weight from input 2 to hidden 2

currentweights[0][2][2] = -0.0347; //Weight from input 2 to hidden 3

currentweights[0][2][3] = -0.04861; //Weight from input 2 to hidden 4

currentweights[1][0][0] = -0.1401; //Weight from hidden bias to output

currentweights[1][1][0] = 0.4919; //Weight from hidden 1 to output

currentweights[1][2][0] = -0.2913; //Weight from hidden 2 to output

currentweights[1][3][0] = -0.3979; //Weight from hidden 3 to output

currentweights[1][4][0] = 0.3581; //Weight from hidden 4 to output

\*/

Random rnd = new Random();

System.out.println("InputToHiddenWeight:");

for(int i=0;i<=numberofinputneuron;i++){

for(int j=0;j<numberofhiddenneuron;j++){

if (minvalue == -1)

currentweights[0][i][j] = rnd.nextDouble() \* 2 - 1;

else

currentweights[0][i][j] = rnd.nextDouble() - 0.5;

//currentweights[0][i][j] = Math.random() - 0.5;

System.out.println(currentweights[0][i][j]);

}

}

System.out.println("HiddenToOutputWeight:");

for(int i=0;i<=numberofhiddenneuron;i++){

for(int j=0;j<numberofoutputneuron;j++){

if (minvalue == -1)

currentweights[1][i][j] = rnd.nextDouble() \* 2 - 1;

else

currentweights[1][i][j] = rnd.nextDouble() - 0.5;

//currentweights[1][i][j] = Math.random() - 0.5;

System.out.println(currentweights[1][i][j]);

}

}

}

private double ActivationFunction(double value){

/\*\*\*\*\*\*\*\*\* Functions returns an activated value of input parameter \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double result = 0;

switch(activationtype){

case 1://sigmoid

result = 1 / (1+Math.exp(-value));

break;

case 2://bipolar sigmoid

result = (2 / (1+Math.exp(-value))) - 1;

break;

case 3:

result = (Math.exp(value) - Math.exp(-value))/((Math.exp(value) + Math.exp(-value)));

default:

result = 1 / (1+Math.exp(-value));

break;

}

return result;

}

private double DerivationOfActivationFunction(double value){

/\*\*\*\*\*\*\*\*\* Returns the derivative of Activated value of Input Parameter \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double result=0;

switch(activationtype){

case 1:

result = ActivationFunction(value) \* (1 -ActivationFunction(value));

break;

case 2:

result = 0.5 \* (1 + ActivationFunction(value)) \* (1 -ActivationFunction(value));

break;

case 3:

result = (1 + ActivationFunction(value)) \* (1 -ActivationFunction(value));

break;

default:

result = ActivationFunction(value) \* (1 -ActivationFunction(value));

break;

}

return result;

}

double checkerror = 1;

double totalerror = 1;

private void FeedForward(int iteration, double maxerror) throws IOException{

/\*\*\*\*\*\*\*\*\* Main body of the Training Implementation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int epoch;

// Iterates through 4 patterns in one loop also known as epoch

for(epoch=0;epoch<iteration &&checkerror > maxerror ;epoch++){

//Iterates through individual input pattern, computes error and updates weight

for(int k=0;k<numberofpattern;k++){

//Forward

for(int i=1;i<=numberofhiddenneuron;i++)

hiddenNeuronsAndBias[0][i] = 0;

for(int i=0;i<numberofoutputneuron;i++)

outputNeuronAndBias[i] = 0;

for(int i=1;i<=numberofhiddenneuron;i++){

for(int j=0;j<=numberofinputneuron;j++){

if(j!=0){ //Checking for bias input

hiddenNeuronsAndBias[0][i] +=currentweights[0][j][i-1]\*inputpattern[k][j-1];

}

else{

hiddenNeuronsAndBias[0][i] += currentweights[0][j][i-1];

}

}

//System.out.println("hiddenout["+i+"]:"+hiddenNeuronsAndBias[0][i]);

}

activatedhiddenlayer[0][0] = 1; //Biased Term

for(int i=1;i<=numberofhiddenneuron;i++){

activatedhiddenlayer[0][i] = ActivationFunction(hiddenNeuronsAndBias[0][i]);

//System.out.println("activatedhiddenout["+i+"]:"+activatedhiddenlayer[0][i]);

}

for(int i=0;i<numberofoutputneuron;i++){

for(int j=0;j<=numberofhiddenneuron;j++){

if(j!=0){

outputNeuronAndBias[i] += currentweights[1][j][i] \*activatedhiddenlayer[0][j];

}

else{

outputNeuronAndBias[i] += currentweights[1][j][i];

}

}

//System.out.println("Outputout["+i+"]:"+outputNeuronAndBias[i]);

}

for(int i=0;i<numberofoutputneuron;i++){

activatedoutputlayer[i] = ActivationFunction(outputNeuronAndBias[i]);

//System.out.println("activatedhiddenout["+i+"]:"+activatedoutputlayer[i]);

}

for(int i=0;i<numberofoutputneuron;i++){

outputerror[i] = (targetpattern[k] -activatedoutputlayer[i])\*(DerivationOfActivationFunction(outputNeuronAndBias[i]));

//System.out.println("outputerror["+i+"]:"+outputerror[i]);

}

patternerror[k] = 0;

for(int i=0;i<numberofoutputneuron;i++){

patternerror[k] += Math.pow((targetpattern[k]-activatedoutputlayer[i]),2);

//System.out.println("patternerror["+k+"]:"+patternerror[k]);

}

//patternerror[k] = 0.5 \* patternerror[k];

if(k==numberofpattern-1){

totalerror = 0;

for(int i=0;i<numberofpattern;i++){

totalerror += patternerror[i];

}

checkerror = totalerror/2;

//EPOCH[epoch] = totalerror / numberofpattern;

EPOCH[epoch] = totalerror/2;

System.out.println("EPOCH["+epoch+"]:"+EPOCH[epoch]);

}

//if(k!=0) {

for(int i=0;i<numberofoutputneuron;i++){

for(int j=0;j<=numberofhiddenneuron;j++){

deltaWeightforMomentum[1][j][i] = previousdeltaweights[1][j][i];

}

}

//}

for(int i=0;i<numberofoutputneuron;i++){

for(int j=0;j<=numberofhiddenneuron;j++){

if(j!=0){

deltaweights[1][j][i] = (learningrate\*outputerror[i]\*activatedhiddenlayer[0][j])+(momentum\*deltaWeightforMomentum[1][j][i]);

}

else{

deltaweights[1][j][i] =(learningrate\*outputerror[i])+(momentum\*deltaWeightforMomentum[1][j][i]);

}

}

}

for(int i=0;i<numberofoutputneuron;i++){

for(int j=0;j<=numberofhiddenneuron;j++){

previousdeltaweights[1][j][i] =deltaweights[1][j][i];

}

}

//backpropagation

for(int i=1;i<=numberofhiddenneuron;i++){

backpropagationhiddenlayer[0][i] = 0; //this is the delta\_inj

}

for(int i=1;i<=numberofhiddenneuron;i++){

for(int j=0;j<numberofoutputneuron;j++){

backpropagationhiddenlayer[0][i] +=outputerror[j]\*currentweights[1][i][j];

}

}

for(int i=1;i<=numberofhiddenneuron;i++){

errorhidden[0][i] =backpropagationhiddenlayer[0][i]\*DerivationOfActivationFunction(hiddenNeuronsAndBias[0][i]);

}

//if(k!=0) {

for(int i=0;i<numberofhiddenneuron;i++){

for(int j=0;j<=numberofinputneuron;j++){

deltaWeightforMomentum[0][j][i] =previousdeltaweights[0][j][i];

}

}

//}

for(int i=1;i<=numberofhiddenneuron;i++){

for(int j=0;j<=numberofinputneuron;j++){

if(j!=0){

deltaweights[0][j][i-1] =(learningrate\*errorhidden[0][i]\*inputpattern[k][j-1])+(momentum\*deltaWeightforMomentum[0][j][i-1]);

}

else{

deltaweights[0][j][i-1] =(learningrate\*errorhidden[0][i])+(momentum\*deltaWeightforMomentum[0][j][i-1]);

}

}

}

for(int i=0;i<numberofhiddenneuron;i++){

for(int j=0;j<=numberofinputneuron;j++){

previousdeltaweights[0][j][i] =deltaweights[0][j][i];

}

}

for(int i=0;i<numberofoutputneuron;i++){

for(int j=0;j<=numberofhiddenneuron;j++){

currentweights[1][j][i] += deltaweights[1][j][i];

}

}

for(int i=0;i<numberofhiddenneuron;i++){

for(int j=0;j<=numberofinputneuron;j++){

currentweights[0][j][i] += deltaweights[0][j][i];

}

}

}

}

if (epoch == iteration)

System.out.println("!Error training try again");

else {

WriteErrorToFileForPlot(epoch,EPOCH);

WriteWeightsToRetain(currentweights,numberofinputneuron,numberofhiddenneuron,numberofoutputneuron);

}

}

public void CheckFinalOutput(){

/\*\*\*\*\*\*\*\*\* Checks the Trained Neural Net \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

for(int k=0;k<numberofpattern;k++){

for(int i=1;i<=numberofhiddenneuron;i++)

hiddenNeuronsAndBias[0][i] = 0;

for(int i=0;i<numberofoutputneuron;i++)

outputNeuronAndBias[i] = 0;

for(int i=1;i<=numberofhiddenneuron;i++){

for(int j=0;j<=numberofinputneuron;j++){

if(j!=0){

hiddenNeuronsAndBias[0][i] +=currentweights[0][j][i-1]\*inputpattern[k][j-1];

}

else{

hiddenNeuronsAndBias[0][i] += currentweights[0][j][i-1];

}

}

//System.out.println("hiddenout["+i+"]:"+hiddenNeuronsAndBias[0][i]);

}

for(int i=1;i<=numberofhiddenneuron;i++){

activatedhiddenlayer[0][i] = ActivationFunction(hiddenNeuronsAndBias[0][i]);

//System.out.println("activatedhiddenout["+i+"]:"+activatedhiddenlayer[0][i]);

}

for(int i=0;i<numberofoutputneuron;i++){

for(int j=0;j<=numberofhiddenneuron;j++){

if(j!=0){

outputNeuronAndBias[i] += currentweights[1][j][i] \*activatedhiddenlayer[0][j];

}

else{

outputNeuronAndBias[i] += currentweights[1][j][i];

}

}

//System.out.println("Outputout["+i+"]:"+outputNeuronAndBias[i]);

}

for(int i=0;i<numberofoutputneuron;i++){

activatedoutputlayer[i] = ActivationFunction(outputNeuronAndBias[i]);

System.out.println("Mapping input["+k+"] gives output :"+activatedoutputlayer[i]);

}

}

}

public void WriteErrorToFileForPlot(int totaliterations,double[] errorvals) throws IOException {

/\*\*\*\*\*\*\*\*\* Writes the total error to a CSV file \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

StringBuilder sb = new StringBuilder();

/\* Write to File these values \*/

int filesuffix = 0;

File folder = new File("F:\\\\UBC\\\\Winter Term 1 Sept - Dec\\\\EECE 592\\\\XOR Results\\\\Error Results");

File[] listOfFiles = folder.listFiles();

if(listOfFiles.length > 0) {

String name\_of\_file = listOfFiles[listOfFiles.length - 1].getName();

String numberOnly= name\_of\_file.replaceAll("[^0-9]", "");

filesuffix = Integer.parseInt(numberOnly) + 1;

}

String path = "F:\\UBC\\Winter Term 1 Sept - Dec\\EECE 592\\XOR Results\\Error Results\\"+runMode+"\_test\_"+filesuffix+".csv";

File file = new File(path);

file.createNewFile();

FileWriter fw = new FileWriter(file.getAbsoluteFile(),true);

BufferedWriter bw = new BufferedWriter(fw);

sb.append("epoch Number");

sb.append(',');

sb.append("Square Error");

sb.append('\n');

for(int errval = 0; errval < totaliterations ;errval ++) {

//System.out.println("EPOCH # "+ errval + " : Squared Error is : "+errorvals[errval] + " ");

sb.append(errval);

sb.append(',');

sb.append(errorvals[errval]\*2);

sb.append('\n');

}

bw.write(sb.toString());

bw.close();

}

public void WriteWeightsToRetain(double[][][] weights\_To\_Write, int num\_of\_input, int num\_of\_hidden, int num\_of\_output) throws IOException {

/\*\*\*\*\*\*\*\*\* Writes the final weights for further implementation to a CSV File \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

StringBuilder sb = new StringBuilder();

/\* Write to File these values \*/

int filesuffix = 0;

File folder = new File("F:\\\\UBC\\\\Winter Term 1 Sept - Dec\\\\EECE 592\\\\XOR Results\\\\Trained Weights");

File[] listOfFiles = folder.listFiles();

if(listOfFiles.length > 0) {

String name\_of\_file = listOfFiles[listOfFiles.length - 1].getName();

String numberOnly= name\_of\_file.replaceAll("[^0-9]", "");

filesuffix = Integer.parseInt(numberOnly) + 1;

}

String path = "F:\\UBC\\Winter Term 1 Sept - Dec\\EECE 592\\XOR Results\\Trained Weights\\"+runMode+"\_Weights\_of\_Run\_"+filesuffix+".csv";

File file = new File(path);

file.createNewFile();

FileWriter fw = new FileWriter(file.getAbsoluteFile(),true);

BufferedWriter bw = new BufferedWriter(fw);

sb.append("Weight Variable");

sb.append(',');

sb.append("Weight Value");

sb.append('\n');

for(int i=0;i<num\_of\_hidden;i++){

for(int j=0;j<=num\_of\_input;j++){

sb.append("currentweights[0]["+j+"]["+i+"]");

sb.append(',');

sb.append(currentweights[0][j][i]);

sb.append('\n');

}

}

for(int i=0;i<num\_of\_output;i++){

for(int j=0;j<=num\_of\_hidden;j++){

sb.append("currentweights[1]["+j+"]["+i+"]");

sb.append(',');

sb.append(currentweights[1][j][i]);

sb.append('\n');

}

}

bw.write(sb.toString());

bw.close();

}

public void RunNeuralNet() throws IOException{

/\*\*\*\*\*\*\*\*\* A function that runs all the required functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

InitializeInputs(minvalue,maxvalue);

InitializeOutputs(minvalue,maxvalue);

GenerateNeuralNet();

FeedForward(maxiteration,minErrorCondition);

CheckFinalOutput();

}

//variable definition

private double[][] inputpattern;

private int numberofpattern;

private int patterndimension;

private double[] targetpattern;

private int targetdimension;

private double[][] hiddenNeuronsAndBias;

private double[][] backpropagationhiddenlayer;

private double[][] errorhidden;

private double[] outputNeuronAndBias;

private double[][] activatedhiddenlayer;

private double[] activatedoutputlayer;

private double[] outputerror;

private double[][][] currentweights;

private double[][][] deltaWeightforMomentum;

private double[][][] deltaweights;

private double[][][] previousdeltaweights;

private double[] patternerror;

private double[] EPOCH;

private double learningrate;

private double momentum;

private int numberofhiddenneuron;

private int numberofinputneuron;

private int numberofoutputneuron;

private int numberofhiddenlayer;

private int activationtype;

private int maxiteration;

private double minvalue;

private double maxvalue;

private double minErrorCondition = 0.05;

//Variables to select activation function

public static final char runMode = 'C'; // 'A' for binary, 'B' for bipolar, 'C' for with momentum

public static final int BINARY\_SIGMOID = 1;

public static final int BIPOLAR\_SIGMOID = 2;

public static final int HYPERBOLIC\_TANGENT = 3;

//Main Function is defined below

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

My\_XOR\_NN NN = new My\_XOR\_NN();

NN.numberofpattern = 4;

NN.patterndimension = 2; //2 inputs

NN.targetdimension = 1;

NN.numberofhiddenlayer = 1;

NN.numberofhiddenneuron = 4;

NN.minvalue = -1; //Change this value to change the run mode as well as activation function

NN.maxvalue = 1;

if(NN.minvalue == -1)

NN.activationtype = BIPOLAR\_SIGMOID;

else

NN.activationtype = BINARY\_SIGMOID;

NN.learningrate = 0.2;

NN.momentum = 0.9;

NN.maxiteration = 7000;

NN.RunNeuralNet();

}

}