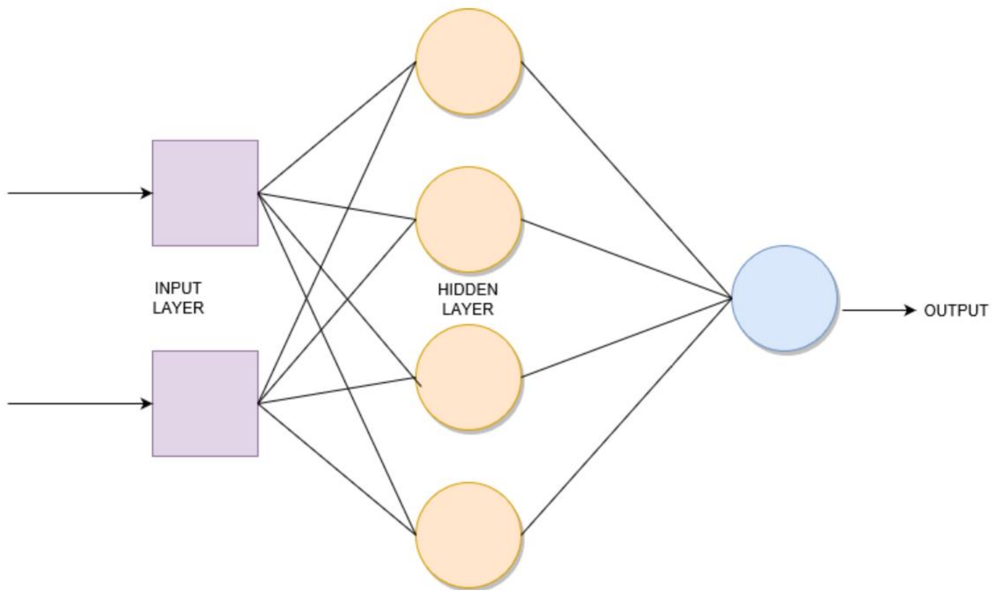
Student Name: Ravneet Kaur Student Number: 41580994

CPEN-502 101

Part 1a - Backpropagation Learning



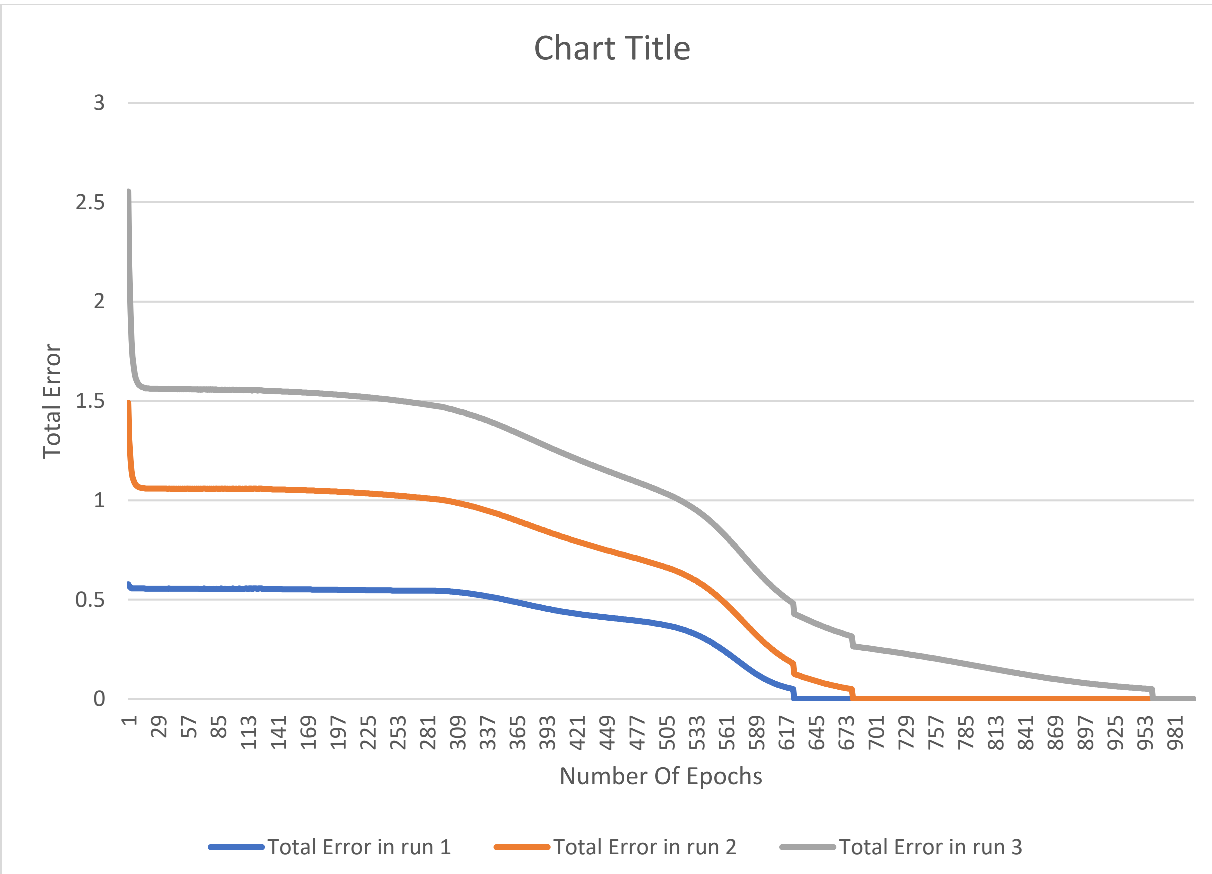
XOR with Bipolar Inputs

X1	X2	Y
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1

XOR with Binary Inputs

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

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.



(a)This graph shows the total error vs number of epochs required using Binary representation

Findings:

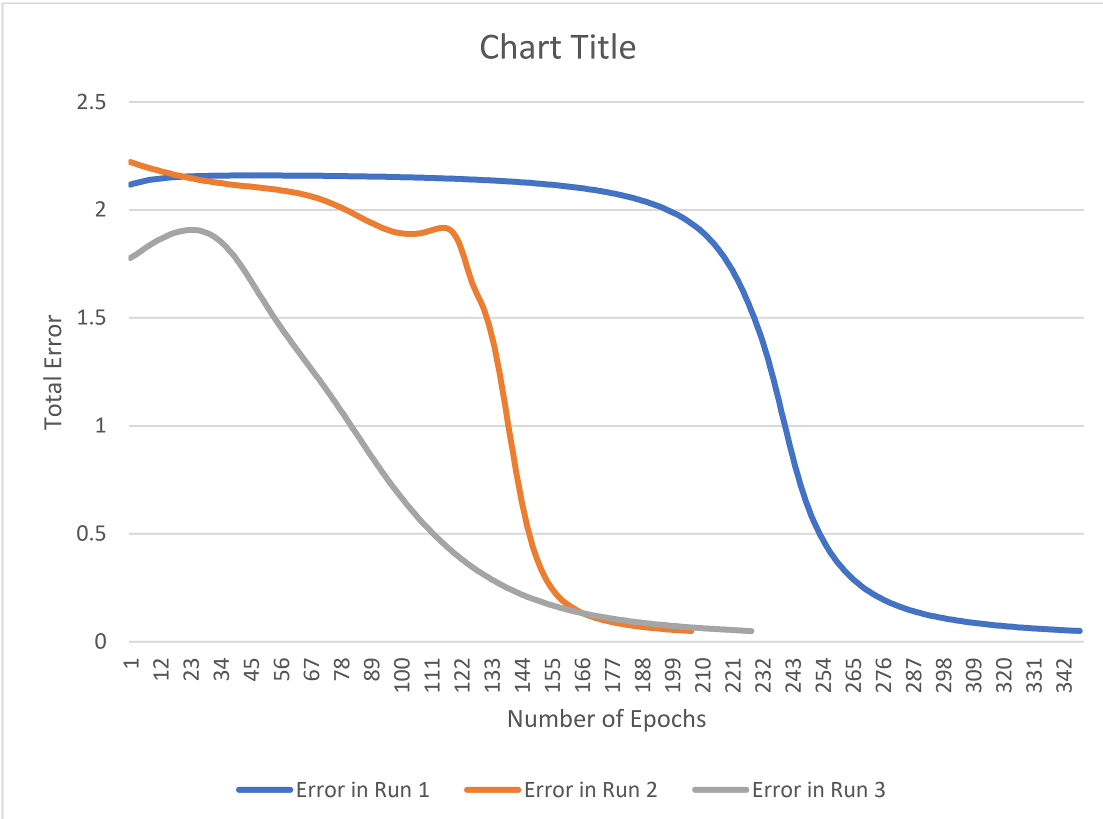
1st run: Minimum number of epochs required are 625

2nd run: Minimum number of epochs required are 680

3rd run: Minimum number of epochs required are 960

The average number of epochs for 10 runs with momentum 0.0 are 1002.

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?



(b)This graph shows the total error vs number of epochs required using Bipolar representation

Findings:

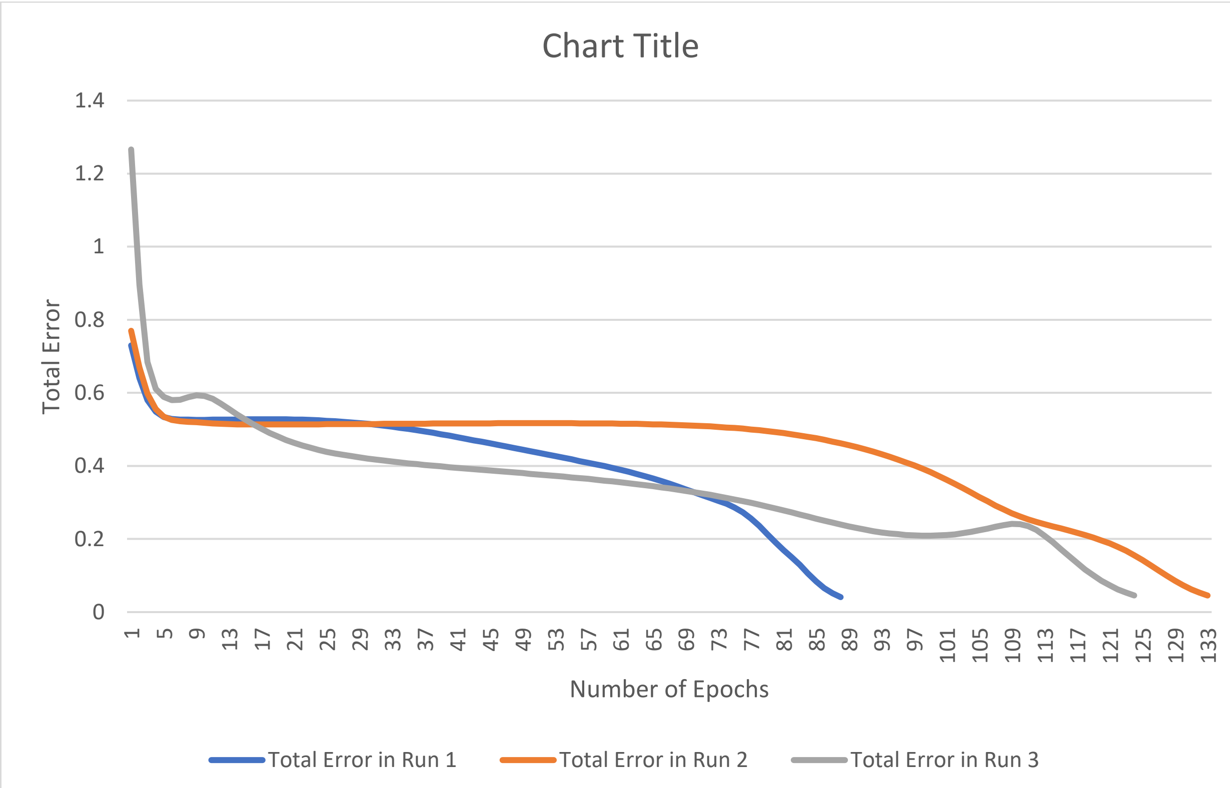
1st run: Minimum number of epochs required are 348

2nd run: Minimum number of epochs required are 206

3rd run: Minimum number of epochs required are 228

The average number of epochs for 10 runs with momentum 0.0 are 228.

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



(c1) This graph shows the total error vs number of epochs required using Binary representation with Momentum 0.9

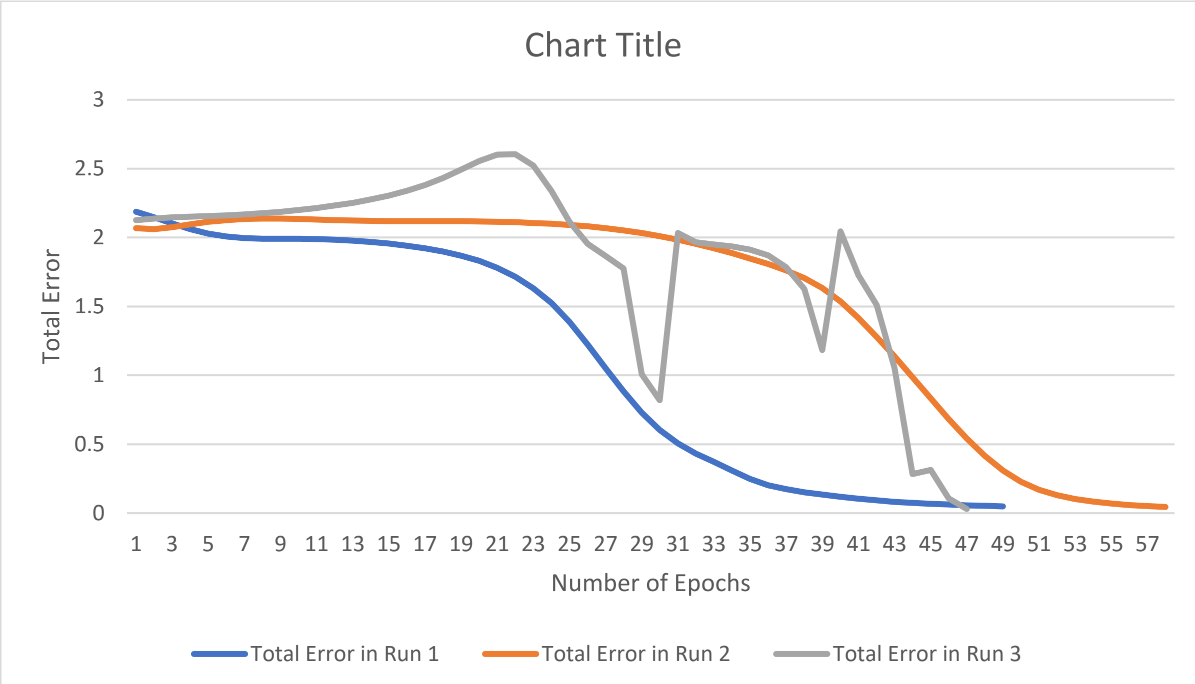
Findings:

1st run: Minimum number of epochs required are 88

2nd run: Minimum number of epochs required are 133

3rd run: Minimum number of epochs required are 124

The average number of epochs in case of Binary representation for 10 runs with momentum 0.9 are 96



(c2) This graph shows the total error vs number of epochs required using Bipolar representation with Momentum 0.9

Findings:

1st run: Minimum number of epochs required are 49

2nd run: Minimum number of epochs required are 58

3rd run: Minimum number of epochs required are 57

The average number of epochs in case of Bipolar representation for 10 runs with momentum 0.9 are 100.

Source Code

ActivationFunction.java

```
public class ActivationFunction
{

public double bipolarSigmoid(double x)

       {
              return ((2.0/(1+Math.exp(-x)))-1);
   }
public double binarySigmoid(double x)

       {
              return (1.0/(1+(Math.exp(-x))));
   }

double derivativeBinarySigmoid(double x)
{
       return (binarySigmoid(x)*(1-binarySigmoid(x)));

}
double derivativeBipolarSigmoid(double x)
{
       return ( (1+bipolarSigmoid(x)) * 0.5 * (1-bipolarSigmoid(x)) );

}

}
```

Neuron.java

```
import java.util.Random;

public class Neuron {
public int number_Inputs=2;
public int number_Hidden=4;
public int number_Outputs=1;

public double momemtum=0.9;
public double learningRate=0.2;
```

```java
double[] outputFromIn= new double[number_Inputs];
double outputError;




double finalError=0.05;
int epoch=0;
public double [][]weightInputToHidden=new double[number_Inputs][number_Hidden];
public double [][]weightHiddenToOutput=new double[number_Hidden][number_Outputs];

public double[] weightBiasHidden= new double[number_Hidden];
public double[] weightBiasOutput= new double[number_Outputs];

public void calculateRandomWeight()
{

	for (int i=0; i<number_Hidden; i++)
	{
		for (int j=0; j<number_Inputs; j++)
		{
			Random r =new Random();
			weightInputToHidden[j][i]= r.nextDouble() - 0.5;

		}
		Random rand =new Random();
		weightBiasHidden[i] = rand.nextDouble() -0.5;
	}




		for (int k=0; k<number_Outputs; k++)

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

			Random r =new Random();
			weightHiddenToOutput[j][k]= r.nextDouble()-0.5;

			}
			Random rand =new Random();
			weightBiasOutput[k] = rand.nextDouble() -0.5;
	    }

}
}
```

BackPropogationAlgorithm.java

```java
public class BackpropogationAlgorithm {
```

```
      double [] X = new double [2];
      double myError=1.0;
 double finalError=0.05;
 int epoch=0;
 public double input_Bias=1.0;
 public double hidden_Bias=1.0;
// public double [][]inputData= {{0,0},{0,1},{1,0},{1,1}};
 //double []expectedOutput= {0,1,1,0};

double [][]inputData= {{-1,-1},{-1,1},{1,-1},{1,1}};
double []expectedOutput= {-1,1,1,-1};

 int number_Inputs=2;
 int number_Hidden=4;
 int number_Outputs=1;
 double[] outputError=new double[number_Outputs];
 double[] finalHiddenInput=new double[number_Hidden];
 double[] finalHiddenOutput=new double[number_Hidden];
 double[] finalOutput= new double[number_Outputs];
 boolean activationFunction=true;
 double[] hiddenEror=new double[number_Hidden];

 double [][]oldWeightInputToHidden=new double[number_Hidden][number_Inputs];
 double [][]oldWeightHiddenToOutput=new double[number_Hidden][number_Outputs];
 double [][]newWeightInputToHidden=new double[number_Hidden][number_Inputs];
 double [][]newWeightHiddenToOutput=new double[number_Hidden][number_Outputs];
 double[] outIn = new double [number_Hidden];


 double[] in=new double[number_Inputs];

 public void backPropagation()
 {
//System.out.println(finalError);
 Neuron n=new Neuron();
 n.calculateRandomWeight();
 //System.out.println(n.weightInputToHidden[0][0]);
 ActivationFunction aF=new ActivationFunction();
 n.calculateRandomWeight();

 while (myError>finalError && epoch<10000)
 {
       myError = 0;
       double weightSumInputHidden=0.0;
       double weightSumHiddenOutput=0.0;

       epoch++;

       for(int nInputs=0;nInputs<4;nInputs++)
       {
             X[0] = inputData[nInputs][0];
             X[1] = inputData[nInputs][1];
             for(int i=0;i<number_Hidden;i++)
             {
                   for(int j=0;j<number_Inputs;j++)
```

```
                        {
                               weightSumInputHidden= weightSumInputHidden+ (X[j]*
n.weightInputToHidden[j][i]);
                        }
                        finalHiddenInput[i]= weightSumInputHidden +
(n.weightBiasHidden[i] * input_Bias);
                        weightSumInputHidden=0;

                        if(activationFunction==false)
                              {
                                     finalHiddenOutput[i]=
aF.binarySigmoid(finalHiddenInput[i]);
                              }
                        else
                              {
                                     finalHiddenOutput[i]=
aF.bipolarSigmoid(finalHiddenInput[i]);
                              }
                }


                for (int i=0;i<number_Outputs;i++)
                {
                       for (int j=0;j<number_Hidden;j++)
                       {
                              weightSumHiddenOutput += (finalHiddenOutput[j] *
n.weightHiddenToOutput[j][i]);
                       }
                       outIn[i] = weightSumHiddenOutput+(hidden_Bias *
n.weightBiasOutput[0]);
                       if (activationFunction == false)
                       {
                              finalOutput[i] = aF.binarySigmoid(outIn[i]);
                       }
                       else
                       {
                              finalOutput[i] = aF.bipolarSigmoid(outIn[i]);
                       }
                       //System.out.println(finalOutput[i]);
                       weightSumHiddenOutput=0;
                }



                double errorChange= (Math.pow((expectedOutput[nInputs]-finalOutput[0]),
2))*0.5;
                myError= myError+ errorChange;
                //System.out.println(expectedOutput[nInputs]);
                //System.out.println(errorChange);
                for (int i=0; i<number_Outputs; i++)
                {
                        if (activationFunction == false)
                        {
                               outputError[i] = (expectedOutput[nInputs] -
finalOutput[i])*(aF.derivativeBinarySigmoid(outIn[i]));
```

```
                    }
                    else
                    {
                          outputError[i] = (expectedOutput[nInputs] -
finalOutput[i])*(aF.derivativeBipolarSigmoid(outIn[i]));
                    }

                    for (int k=0; k<number_Hidden; k++)
                    {
                          oldWeightHiddenToOutput[k][i] =
newWeightHiddenToOutput[k][i];
                          newWeightHiddenToOutput[k][i] = (outputError[i] *
finalHiddenOutput[k] * n.learningRate) + (oldWeightHiddenToOutput[k][i]* n.momemtum);
                    }
                         n.weightBiasOutput[i] += n.learningRate * outputError[i];

              }

              for (int k=0; k<number_Hidden; k++)
              {
                    for (int i=0; i<number_Outputs; i++)
                    {
                          weightSumHiddenOutput += outputError[i]*
n.weightHiddenToOutput[k][i];
                    }

                    if (activationFunction == false)
                    {
                          hiddenEror[k] = weightSumHiddenOutput *
aF.derivativeBinarySigmoid(finalHiddenInput[k]);
                    }
                    else
                    {
                          hiddenEror[k] = weightSumHiddenOutput *
aF.derivativeBipolarSigmoid(finalHiddenInput[k]);
                    }


                    for (int g=0; g<number_Inputs;g++)
                    {
                          oldWeightInputToHidden[k][g] =
newWeightInputToHidden[k][g];
                          newWeightInputToHidden[k][g] = (n.learningRate *
hiddenEror[k] * X[g]) + (oldWeightInputToHidden[k][g]* n.momemtum);
                    }
                    n.weightBiasHidden[k] += n.learningRate * hiddenEror[k];
                    }



                   for (int i=0; i<number_Outputs; i++)
                   {
                    for (int k=0; k<number_Hidden; k++)
                    {
```

```
                     n.weightHiddenToOutput[k][i] = n.weightHiddenToOutput[k][i] +
newWeightHiddenToOutput[k][i];
                     }
                     }
                     for (int k=0; k<number_Hidden; k++)
                     {
                     for (int i=0; i<number_Inputs; i++)
                     {
                     n.weightInputToHidden[i][k] = n.weightInputToHidden[i][k] +
newWeightInputToHidden[k][i];
                     }
                     }

 }

        //System.out.println("epoch =" + epoch + "\t Error =" + myError);
        System.out.println(myError);
        //System.out.println("epoch =" + epoch + "\t Error =" + myError);
       }

                    if (epoch==10000)
                    {
                    System.out.println("Error");
                    }

 }
                    public static void main(String[] args) {
                    BackpropogationAlgorithm n1 = new BackpropogationAlgorithm() ;
                    n1.backPropagation();

}

}
```