EECE 592 Project Part 1: Backpropagation (November 2010)

Alison Michan 79172995, UBC M.A.Sc. Student
EECE 592 Architectures for Learning Systems, Dr. Sarbjit Sarkaria

***Abstract*—A backpropagation algorithm was implemented in order to train a 2-4-1 network. Algorithm parameters were investigated including learning rate, momentum, and initial weight values. Additionally, both binary and bipolar representations were investigated. The fastest convergence was found using a bipolar representation and including momentum. Performance was measured by calculating total error after each epoch.**

I. Introduction

Backpropagation is an iterative learning method that can be used to train neural networks. In order to investigate the method in detail, the algorithm was implemented with Java using an XOR training set and a 2-4-1 network. A 2-4-1 neural network with a bias of 1 is illustrated below:

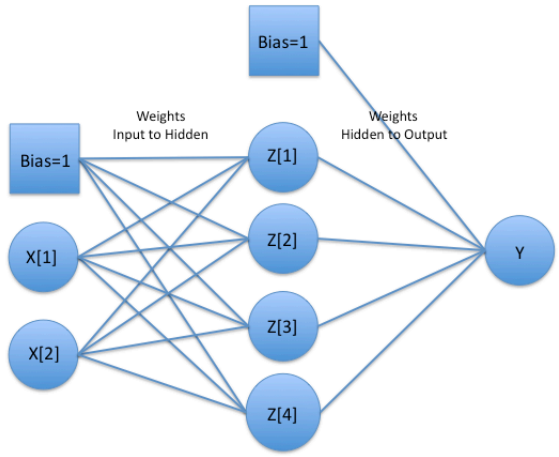

Figure 1: Backpropagation 2-4-1 neural network

The input vector is connected to a hidden layer of neurons that is connected to the final output. All connections have weighted values that are adjusted as a part of the iterative learning method. The backpropagation method can be divided into three steps, as described by Fausett [1]

1) *Feedforward of the input pattern*
2) *Calculation and Backpropagation of error*
3) *Weight adjustments*

Weights were adjusted after each training pattern was completed, step 3, and the total error was measured after each epoch. The number of epochs was measured to achieve a total error of less than 0.05. Experimentation of algorithm parameters was completed in order to understand their impact on the algorithm results. The results will be utilized in future parts of the Robocode project.

II. Experimental Results, Question 1

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. Question 1a

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?

A binary XOR representation was used and the training pattern is shown below.

x1	x2	y
1	0	1
0	1	1
1	1	0
0	0	0

Table 1: Binary XOR training pattern

A sigmoid 'squashing' function was used and for the binary training set it was used on the interval [0,1]. The equations as described by Faucett [1] are shown below.

$$f(x) = \frac{1}{1+e^{-x}} \qquad (1)\ [1]$$

$$f'(x) = f(x)*(1-f(x)) \qquad (2)\ [1]$$

Example results are shown in Figure 2, where the algorithm requires 2815 epochs to converge to less than 0.05.

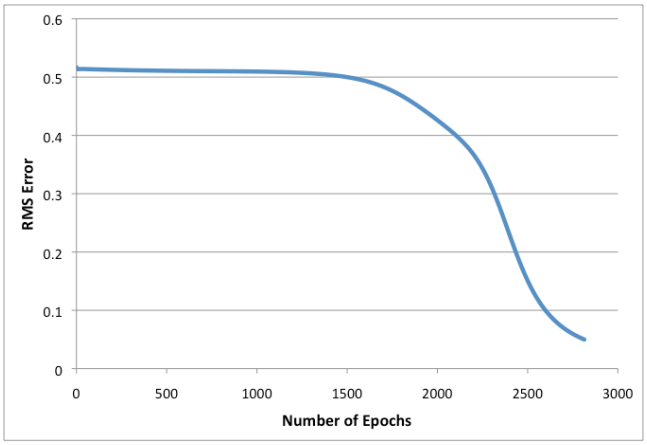

Figure 2: (1a) Example of convergence, 2815 epochs

For 100 trials, an average of 4394 epochs were taken, with a standard deviation of 1260 epochs and results are summarized below.

Average Number Epochs	4394
Standard deviation	1260

Table 2: Binary representation results

B. *Question 1b*

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 input was modified so the XOR training input and the training output are shown below.

x1	x2	y
1	-1	1
-1	1	1
1	1	-1
-1	-1	-1

Table 3: Bipolar XOR training pattern

A generalized sigmoid 'squashing' function as described by Faucett (pp 309) [1] was used for the bipolar training set on the interval [a=-1,b=1] with parameters defined below.

$$\gamma = b - a$$

$$\eta = -a$$

The generalized sigmoid function and its derivative are shown below in equation 3 and 4.

$$g(x) = \gamma * f(x) - \eta \qquad (3)\ [1]$$

$$g'(x) = \frac{1}{\gamma} * (\eta + g(x)) * (\gamma - \eta - g(x)) \qquad (4)\ [1]$$

The convergence time was improved by an approximate factor of 10 compared with binary. An example of the convergence with number of epochs is shown in Figure 3.

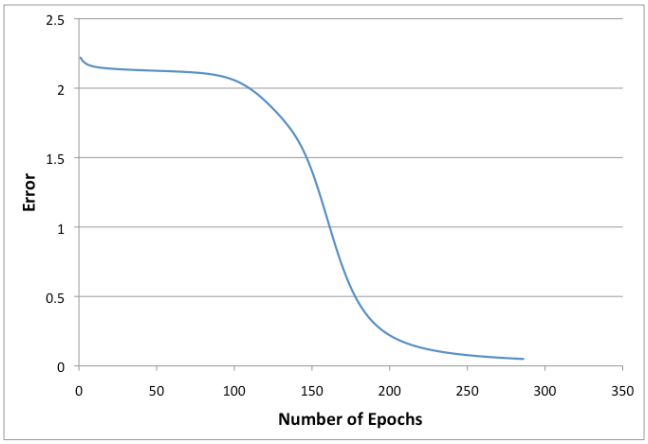

Figure 3: (1b) Example of convergence, 286 epochs

Several iterations were completed, using random values in the range of -0.5 to +0.5 for initial weights for both the input and hidden layer. When larger numbers of epochs were required, a longer plateau occurred during the beginning of training. This is thought to relate to the initial weight values. An example where a larger number of epochs and longer plateau were observed is shown in Figure 3, where 384 epochs were required.

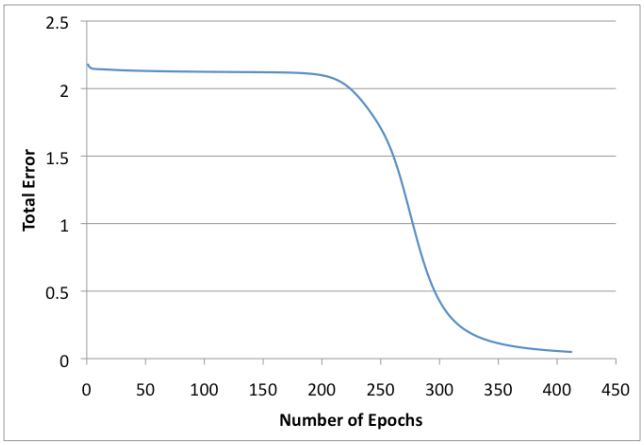

Figure 4: (1b) Example of convergence, 384 epochs

The overall results for 100 trials were tabulated and are summarized below.

Non-convergent	62
Convergent	38
Average Number Epochs	344
Standard deviation	72

Table 4: (1b) convergence of bipolar representation (100 trials). An average of 344 epochs were required.

38% of the 100 trials were convergent. The average number of epochs required to converge to a total error less than 0.05 was 344, calculated from 38 trials.

C. *Question 1c*

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

Using a momentum term of 0.9, convergence had the potential to converge at a rate improved by another factor of 10. The convergence was observed to be as fast as 26 epochs. The total error did have some small oscillations and was much more sensitive to initial weights so had a higher percentage of non-convergent cases (leaving all the parameters the same). An example of the algorithm convergence using momentum of 0.9 is shown below in Figure 5.

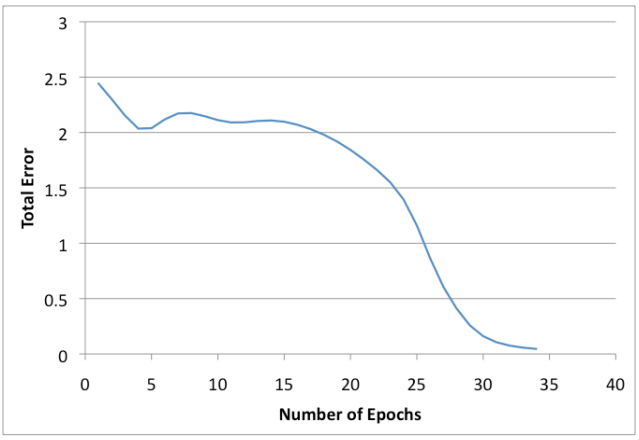

Figure 5: (1c) Convergence using momentum

Overall, adding a momentum term improves the algorithm performance.

D. Question 1d

If you are allowed to freely change any parameters you like, what is the best (i.e. fastest) number of epochs you can obtain to reach the 0.05 level? Do you think there are any other ways (that you did not implement) to improve this further?

Three parameter changes were investigated: initial weights, learning rate, and momentum. First, initial weight values were bounded, such that random values had an absolute value greater than 0.15. This resulted in reduced cases of non-convergence and a small improvement on number of epochs. Secondly, a momentum term was added in order to reduce number of epochs required by a factor of 10. Finally, the learning rate was increased to 0.4 from 0.2 to further reduce number of epochs required to reach the total error of less than 0.05. A direct comparison of 1b with the parameter adjustments is shown below in Table 5.

(1c) Changing parameter values to improve Performance (100 Trials)	1b	1c_a	1c_b	1c_c
Non-convergent	62	3	2	24
Convergent	38	97	98	76
Average Number Epochs	344	216	24	17
Standard deviation	72	66	27	5

Table 5: (1c) Comparison of parameter values to improve performance (100 trials): (1b) parameters given in part 1b (1c_a) Initial weights are bounded <-0.15 and >0.15 (1c_b) Initial weights bounded and momentum term of 0.9 applied (1c _c) Initial weights bounded, momentum 0.9, and learning rate increased to 0.4 from 0.2

Using initial weight bounds, momentum, and an increased learning rate to 0.4, convergence was found to occur with as few as 14 epochs. Further experimentation could be completed with implementations of known backpropagation techniques, but the rate of convergence is not likely improved significantly from 14 epochs for this particular data set. It may be that additional techniques could provide a fast convergence and maintain a higher probability of convergence. It was observed that there is a trade-off between speed of convergence and probability of convergence. For example, it was noted that increasing the learning rate (Table 5 1c_c) also increased the percentage of cases of non-convergence.

E. Question 1e

For a total error of 0.05, compute the average error that can be expected in the output of the neural network.

The total error was calculated from equation 5 below. Calculating the average error for a single training pattern based on a total error of 0.05 gives an average squared error of 2.5% for a single training pattern, or an average difference (output-target) of 15.8% or 0.158.

$$TotalError = \frac{1}{2}\sum_{Pattern=1}^{4}\left(Output - Target\right)^2 \qquad (5)$$

The final output values for one specific trial after a total error of less than 0.05 was achieved are shown below. Note the output value compared to the target value.

outputCurr: 0.8485609846839148	Target: 1.0
outputCurr: 0.8605133479967948	Target: 1.0
outputCurr: -0.8981338530091398	Target: -1.0
outputCurr: -0.9155645822663228	Target: -1.0

On average, this particular trial set had an average absolute difference of output values from target values of 0.12 and a squared error of 1.42% that agrees with the calculated value above. The value is lower since the final iteration had a total error of less than 0.05 of 0.03 in this trial.

III. Experimental Results, Question 2

2) In your bipolar training set, replace the -1 in the input patterns with -1000000. Keep +1 as-is and all other parameters as in (1d). (The target vectors remain at +1 & -1)

A. Question 2a

2a) Again, graph your results and state on average the number of epochs taken to reach a total error of 0.05.

Training inputs were adjusted from -1 to -1000000 and no successful cases of convergence were observed. Example results are shown below, where the program was limited to 99000 epochs before termination.

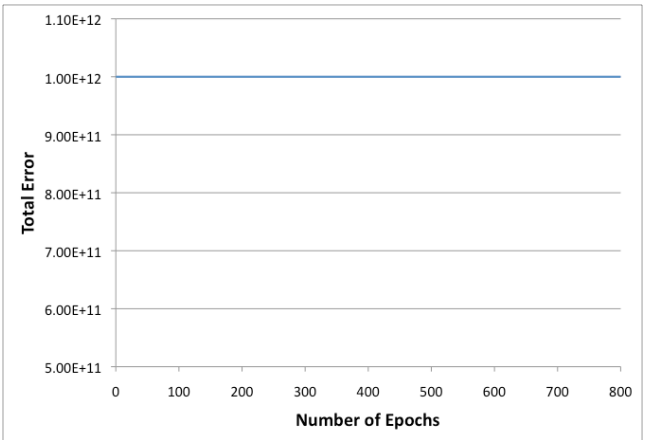

Figure 6: (2a) The algorithm did not converge successfully for adjusted training set values to -1000000 from -1

B. *Question 2b*

2b) Provide an explanation for the difference in results between (2a) and (1d). What does this tell you about how you might want to represent the Robocode data when training a neural net?

It is thought that the weights are required to adjust by such a large amount to fit the training set that they are not able to adjust fast enough or on the scale required, so convergence to an error of less than 0.05 cannot be obtained. This illustrates the need to scale input values so the algorithm can work effectively. Robocode data will need to be scaled.

C. *Question 2c*

2c) During your experimentation, you may or may not have observed that learning does not always converge (i.e. reach a minimum) every time. Can you suggest why this might be?

Examples of non-convergence were observed and it was found that bounding initial weights so they did not start too close to a 0 value and scaling training data so all value were on the same order of magnitude increased the probability of convergence dramatically. Adjusting the learning rate very high also had a trade-off of increased percentage of non-convergence as shown in Table 5 where increasing the learning rate increased the cases of non-convergence from 2% to 24%. If the algorithm is not converging it is because the weight values are adjusted too far away from the values that they need to be in order for the training set to reduce error and converge. Additionally, the algorithm can get trapped in a local minimum without finding the global minimum and momentum helps to avoid this particular situation.

IV. CONCLUSION

A backpropagation network was implemented using XOR training data and the impact of algorithm parameters were investigated. The binary representation required an average of a few thousand epochs, bipolar improved the performance by a factor of 10 to a few hundred epochs, and adding a momentum term of 0.9 improved the algorithm again by a factor of 10. The fastest convergence for a bipolar XOR training set was found using a learning rate coefficient of 0.4 and momentum coefficient of 0.9 as well as bounding the absolute value of initial weights to be greater than 0.15. It was concluded that training input must be scaled in order to achieve a fast convergence and that increasing the speed of convergence with parameter adjustments can reduce the probability of convergence.

APPENDIX

Backpropagation Java code attached, Appendix A

REFERENCES

[1] Fausett, L., “Fundamentals of Neural Networks”, Prentice Hall, 1994, pp. 289–334.

Appendix A: Backpropagation code

```
[1]  package ca.ubc.ece.backprop;
[2]
[3]  public class BPDriver {
[4]
[5]   /**
[6]    * @param args
[7]    */
[8]   public static void main(String[] args) {
[9]       // TODO Auto-generated method stub
[10]
[11]      for(int test=0; test<100; test++){
[12]      NeuralNet Trial = new NeuralNet(2,4,0.4,0.9);
[13]      //numInputs=2, numHidden=4, learningRate=0.2, momentum=0
[14]      double error=0.0;
[15]      double  outputCurr=0.0;
[16]      double X[]=new double[3];
[17]      double Target=0;;
[18]      int numEpochs = 0;
[19]      double RMSError=0.0;
[20]      NeuralNet.initializeTrainingDataBipolar();  //load xor training data
[21]      //NeuralNet.initializeTrainingDataBinary(); //load xor training data
[22]      do{
[23]          RMSError=0.0;
[24]
[25]          for(int trainingPairNum=0; trainingPairNum<4; trainingPairNum++)
[26]          {
[27]              for(int p=0;p<3;p++) //load training inputs
[28]              {
[29]                  X[p]=Trial.trainingDataInput[trainingPairNum][p];
[30]              }
[31]              Target=Trial.trainingDataOutput[trainingPairNum];
[32]              outputCurr=Trial.outputFor(X);
[33]              error=Trial.train(X, Target);
[34]              RMSError=RMSError+0.5*error*error;
[35]
[36]              /**System.out.print("\terrorCurr: ");
[37]              System.out.print(error);
[38]              System.out.print("\toutputCurr: ");
[39]              System.out.print(outputCurr);
[40]              System.out.print("\tTarget: ");
[41]              System.out.println(Target);
[42]              **/
[43]          }
[44]          numEpochs++;
[45]          if(numEpochs>99000)
[46]          {
[47]              RMSError=0.04;
[48]          }
[49]          //System.out.println(RMSError);
[50]          //System.out.println(RMSError);
[51]      }while(RMSError>0.05);
[52]      //System.out.println("numEpochs  ");
[53]      System.out.println(numEpochs);
[54]      }
[55]  }
[56]  }
```

```
[1]  package ca.ubc.ece.backprop;
[2]
[3]  import java.io.File;
[4]  import java.io.IOException;
[5]  import java.math.*;
[6]
[7]  /*
[8]   * @author Michan
[9]   */
[10]
[11]  public class NeuralNet {
[12]
```

```
[13]  /*
[14]   *  Private members of this class
[15]   */
[16]
[17]
[18]  /*
[19]   * Public attributes of this class. Used to capture the largest Q
[20]   */
[21]   public static double[] hiddenNeurons; //hidden neuron vector
[22]   public static double[] lastWeightChangeHiddenOutput; //momentum
[23]   public static double [][] lastWeightChangeInputToHidden; //momentum
[24]   public static double []weightHiddenOutput;//= {-0.1401,0.4919,-0.2913,-0.3979,0.3581};//text weights
[25]   public static double [][]weightInputToHidden;//={ {-0.3378, 0.2771, 0.2895, -0.3329},{0.1970,0.3191,-0.1448, 0.3594},{0.3099, 0.1904, -0.0347, -
       0.4861} };//text weights
[26]   public static double sigma_a=-1; //sigmoid bound a
[27]   public static double sigma_b=1; //sigmoid bound b
[28]   public static double gamma=sigma_b-sigma_a; //for custom sigmoid and prime
[29]   public static double n=-sigma_a; //for custom sigmoid and prime
[30]   public static int numHidden;
[31]   public static int numInputs;
[32]   public static double learningRate;
[33]   public static double momentumWeight;
[34]
[35]   public static int numPatterns;
[36]   public static double[][] trainingDataInput;
[37]   public static double[] trainingDataOutput;
[38]
[39]   public static int patNum; //process variable
[40]   public static double y_output;
[41]  /*
[42]   * Constructor.
[43]   */
[44]    public NeuralNet (int argNumInputs,
[45]              int argNumHidden,
[46]              double argLearningRate,
[47]              double argMomentum)
[48]    {
[49]        numHidden=argNumHidden;
[50]        numInputs=argNumInputs+1; //add 1 for bias
[51]        learningRate=argLearningRate;
[52]        momentumWeight=argMomentum;
[53]        lastWeightChangeHiddenOutput=new double[numHidden+1]; //momentum
[54]        lastWeightChangeInputToHidden=new double [numInputs][numHidden]; //momentum
[55]        hiddenNeurons=new double[numHidden+1];//add 1 for bias of 1 at [0]
[56]        numPatterns=4; //xor
[57]        trainingDataInput = new double[numPatterns][numInputs];//xor
[58]        trainingDataOutput= new double[numPatterns]; //xor
[59]        y_output=0.0;
[60]        weightHiddenOutput=new double[numHidden+1];
[61]       weightInputToHidden=new double[numInputs][numHidden];
[62]        initializeWeights();
[63]    }
[64]
[65]  /*
[66]   *    Return a sigmoid of the input X //make private
[67]   */
[68]   private double sigmoid( double x ) {
[69]       double temp = Math.exp(-x);
[70]       return (1/(1+temp));
[71]   }
[72]  /*
[73]   * This method implements a bipolar sigmoid
[74]   */
[75]   private double customSigmoid( double x ) {
[76]
[77]       double temp=Math.exp(-x);
[78]       temp=1/(1+temp);
[79]       temp=gamma*temp-n;
[80]       return temp;
[81]
[82]   }
[83]
[84]  /*
[85]   * Initialize the weights to random values.
[86]   */
```

```
[87]
[88]   private void initializeWeights() {
[89]       double temp=0.0;
[90]
[91]       for(int j=0;j<=numHidden;j++)
[92]       {
[93]           temp=(Math.random()-0.5);
[94]           if((Math.abs(temp))<0.15){
[95]               temp=temp*10;
[96]           }
[97]           weightHiddenOutput[j]=temp; //make random
[98]       }
[99]       for(int j=0;j<numHidden;j++)
[100]      {
[101]          for(int k=0;k<numInputs;k++)
[102]          {
[103]              temp=(Math.random()-0.5);
[104]              if((Math.abs(temp))<0.15)
[105]              {
[106]                  temp=temp*10;
[107]              }
[108]              weightInputToHidden[k][j]=temp; //make random
[109]          }
[110]      }
[111] }
[112]
[113]/*
[114] * Computes output of the NN without training. ie a forward pass
[115] * pass in training pair including bias
[116] */
[117] public double outputFor( double[] X)
[118] {
[119]      hiddenNeurons[0]=1.0; //apply bias of 1 at hiddenNeurons Zo
[120]      for (int j=1; j<=numHidden; j++) //count through hidden neurons not bias (4 in assign 1)
[121]      {
[122]          hiddenNeurons[j]=0.0;
[123]          for(int i=0;i<numInputs;i++) //count through bias and inputs x1,x2
[124]          {
[125]              hiddenNeurons[j]=hiddenNeurons[j]+(X[i]*weightInputToHidden[i][j-1]);
[126]          }
[127]          hiddenNeurons[j]=customSigmoid(hiddenNeurons[j]);//bipolar
[128]          //hiddenNeurons[j]=sigmoid(hiddenNeurons[j]);//binary
[129]      }
[130]      y_output=0.0;
[131]      for(int j=0; j<=numHidden; j++)
[132]      {
[133]          y_output=y_output+hiddenNeurons[j]*weightHiddenOutput[j];
[134]      }
[135]      y_output=(customSigmoid(y_output));//bipolar
[136]      return y_output;
[137]      //return (sigmoid(y_output));//binary
[138] }
[139]
[140]/*
[141] * This method is used to update the weights of the neural net.
[142] */
[143] public double train( double[] argInputVector, double argTargetOutput)
[144] {
[145]      double errorDelta=0.0;
[146]      double outputError=0.0;
[147]      double hiddenDelta =0.0;
[148]      double[] weightChangeHiddenOutput = new double[numHidden+1]; //momentum
[149]      double[][] weightChangeInputToHidden= new double [numInputs][numHidden]; //momentum
[150]
[151]      //backpropagation to hidden layer
[152]      outputError=(argTargetOutput-y_output); //bipolar
[153]     errorDelta=(outputError*(1/gamma)*(n+y_output)*(gamma-n-y_output));//bipolar
[154]      //outputError=argTargetOutput-(sigmoid(y_output));//binary/bipolar
[155]      //errorDelta=outputError*sigmoid(y_output)*(1-sigmoid(y_output));//hard coded - switch to bipolar/binary
[156]      for(int j=0; j<=numHidden; j++) //j=0 is bias, where hiddenNeurons[0]=1
[157]      {
[158]          //calculate weight correction, added momentum here
[159]          weightChangeHiddenOutput[j]=(learningRate*errorDelta*hiddenNeurons[j])+(momentumWeight*lastWeightChangeHiddenOutput[j]);
[160]          lastWeightChangeHiddenOutput[j]=weightChangeHiddenOutput[j];
[161]      }
```

```
[162]     //backpropagation to input layer
[163]     for(int j=1; j<=numHidden; j++) //offset since bias=1 term at hiddenNeurons[0]
[164]     {
[165]         hiddenDelta=weightHiddenOutput[j]*errorDelta;
[166]         hiddenDelta=(hiddenDelta*(1/gamma)*(n+hiddenNeurons[j])*(gamma-n-hiddenNeurons[j]));//bipolar
[167]         //hiddenDelta=hiddenDelta*(hiddenNeurons[j])*(1-hiddenNeurons[j]); //binary, sigmoid already applied to hiddenNeuron values
[168]         for(int i=0; i<numInputs; i++) //i=0, X[0] is bias=1
[169]         {
[170]             //calculate weight correction, added momentum here
[171]             weightChangeInputToHidden[i][j-1]=(learningRate*hiddenDelta*argInputVector[i])+(momentumWeight*lastWeightChangeInputToHidden[i][j-
      1]);
[172]             lastWeightChangeInputToHidden[i][j-1]=weightChangeInputToHidden[i][j-1];
[173]         }
[174]     }
[175]     //weight change updates Hidden Output
[176]     for(int j=0; j<=numHidden; j++)
[177]     {
[178]         weightHiddenOutput[j]=(weightHiddenOutput[j]+weightChangeHiddenOutput[j]);
[179]     }
[180]     //weight change updates InputToHidden
[181]     for(int j=1; j<=numHidden; j++)
[182]     {
[183]         for(int i=0; i<numInputs; i++)
[184]         {
[185]             weightInputToHidden[i][j-1]=(weightInputToHidden[i][j-1]+weightChangeInputToHidden[i][j-1]);
[186]         }
[187]     }
[188] return outputError;
[189] }
[190]
[191]/*
[192] * saves the weights to file. format of the output is as follows
[193] */
[194] public void save( File argFile){
[195]
[196] }
[197]
[198]/*
[199] * loads the weights from file. format of the file is expected to follow
[200] */
[201]
[202] public void load( String argFileName) throws IOException{
[203]
[204] }
[205]
[206] public static void initializeTrainingDataBinary(){
[207]     //binary
[208]
[209]     trainingDataInput[0][1]=1;
[210]     trainingDataInput[0][2]=0;
[211]     trainingDataInput[0][0]=1;//bias
[212]     trainingDataOutput[0]=1;
[213]
[214]     trainingDataInput[1][1]=0;
[215]     trainingDataInput[1][2]=1;
[216]     trainingDataInput[1][0]=1;//bias
[217]     trainingDataOutput[1]=1;
[218]
[219]     trainingDataInput[2][1]=1;
[220]     trainingDataInput[2][2]=1;
[221]     trainingDataInput[2][0]=1;//bias
[222]     trainingDataOutput[2]=0;
[223]
[224]     trainingDataInput[3][1]=0;
[225]     trainingDataInput[3][2]=0;
[226]     trainingDataInput[3][0]=1;//bias
[227]     trainingDataOutput[3]=0;
[228] }
[229]
[230] public static void initializeTrainingDataBipolar(){
[231]     //bipolar
[232]     trainingDataInput[0][1]=1;
[233]     trainingDataInput[0][2]=-1;
[234]     trainingDataInput[0][0]=1;//bias
[235]     trainingDataOutput[0]=1;
```

```
[236]
[237]     trainingDataInput[1][1]=-1;
[238]     trainingDataInput[1][2]=1;
[239]     trainingDataInput[1][0]=1;//bias
[240]     trainingDataOutput[1]=1;
[241]
[242]     trainingDataInput[2][1]=1;
[243]     trainingDataInput[2][2]=1;
[244]     trainingDataInput[2][0]=1;//bias
[245]     trainingDataOutput[2]=-1;
[246]
[247]     trainingDataInput[3][1]=-1;
[248]     trainingDataInput[3][2]=-1;
[249]     trainingDataInput[3][0]=1;//bias
[250]     trainingDataOutput[3]=-1;
[251] }
[252]
[253]}//End of public class NeuralNet
```