2015

Back Propagation

Back Propagation - XOR problem

**Wong, Vincent P (96139150)**

Contents

[Table of Figures and Tables 1](#_Toc433753678)

[Introduction: 2](#_Toc433753679)

[Results: 2](#_Toc433753680)

[1 a. XOR problem with binary representation [0,1]: 3](#_Toc433753681)

[1 b. Bipolar representation 4](#_Toc433753682)

[1 c. Momentum Term 0.9 5](#_Toc433753683)

[Conclusion: 7](#_Toc433753684)

[Appendix: 7](#_Toc433753685)

[A. Neural Net Implementation of NeuralNetInterface() 7](#_Toc433753686)

[B. Main function code that implements NeuralNetInterface 12](#_Toc433753687)

# Table of Figures and Tables

[Table 1: Epoch results for different parameters in Neural Network 2](#_Toc433753688)

[Table 2: Example of Resulted Weights based on Inputs 7](#_Toc433753689)

[Figure 1: XOR binary results with 0 momentum 3](#_Toc433753695)

[Figure 2: Bipolar 0.05 error, no momentum, sigmoid activation 4](#_Toc433753696)

[Figure 3: Tanh() with Bipolar Input 5](#_Toc433753697)

[Figure 4: Binary, Momentum 0.9 6](#_Toc433753698)

[Figure 5: Bipolar Input, 0.9 MT 6](#_Toc433753699)

# Introduction:

This assignment I utilize a 2 input, 4 hidden, and 1 output neural network for solving XOR problem. In this report I will present the learning curves of adjusting the parameters in my neural network solving of back propagation, and discuss on the challenges and difference seen.

# Results:

First we did the XOR with binary input representation, and then with bipolar representation, and then with momentum turned on to 0.9 for both representation. The number of epochs to reach 0.05 total error where error is calculated as:

Where p is the pattern set (i.e. 4 for XOR problem). Also when binary representation of [0,1] is used, the following activation function is used:

**public** **double** sigmoid(**final** **double** x) {

**return** (1.0 / (1 + Math.*pow*(Math.***E***, (-1) \* x)));

}

Whereas if bipolar representation input is used [-1, 1] then the following activation function is used where the function is adjusted to fit f(x) between -1 and 1:

**public** **double** bipolar\_sig(**final** **double** x) {

**return** ((2.0 / (1 + Math.*pow*(Math.***E***, (-1) \* x))) - 1); // [-1, 1]

}

The following results are tabulated into this chart:

Table 1: Epoch results for different parameters in Neural Network

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type of Input and Parameters Setting | Average epochs till total error reaches 0.05 | Accuracy of results (out of 15 trials) | Standard Deviation | Num Epoch variance |
| Binary, error 0.05, learning rate 0.2 | ~3000 | 98% | 692.1 | 479125.7 |
| Bipolar, error 0.05, learning rate 0.2 | ~220 | 100% | 33.63 | 1080.9 |
| Binary, error 0.05, learning rate 0.2, momentum 0.9 | ~2300 | 88% | 688.4 | 473857.9 |
| Bipolar, error 0.05, learning rate 0.2, momentum 0.9 | ~150 | 93% | 25 | 600.0 |
| Binary, error 0.05, learning rate 0.2, momentum 0.9 (aggressive by using previous weight change) | ~300 | 80% | 140.9 | 19781.04 |
| Bipolar, error 0.05, learning rate 0.2, momentum 0.9 (aggressive by using previous weight change) | ~30 | 80% | 12.4 | 154.6 |

Limitations to the effectiveness of momentum include the fact that the learning rate places an upper limit on the amount by which a weight can be changed and the fact that momentum can cause the weight to be changed in a direction that would increase the error [Jacobs, 1988]. Hence there are less accurate results with momentum (after squashing the output). All results are calculated with 0.2 learning rate.

## 1 a. XOR problem with binary representation [0,1]:

Figure 1: XOR binary results with 0 momentum

On average out of 15 trials, the average number of epochs is ~3000. The graph above shows the data for first 4 trials as shown above. In excel, I fitted a polynomial best fit line over the 4 trails of data points. The reason why there is error noise (±0.1) is due to randomized weights and stochastic “online” update, and the gradient descent is finding the global minimum with 0.2 learning rate.

## 1 b. Bipolar representation

I ran bipolar representation (i.e. [-1,1]) with 0.05 total error, using activation function as such:

**return** ((2.0/(1 + Math.*pow*(Math.***E***, (-1) \* x))) - 1) ;

and using the derivative of that for the error signal calculation:

**return** ((1-Math.*pow*(x, 2))/2);

On average of 15 trials, the number of epochs decreases dramatically to ~200 number of epochs to reach total error of 0.05. Below is a graph of 4 trials with a fitted graph, where series are the trails.

Figure 2: Bipolar 0.05 error, no momentum, sigmoid activation

As an extra experiment, when I used a faster converging activation function like tanh() and derivative of that function as stated in the notes, I was able to get epochs decrease to 66 as shown in the graph below.

Figure 3: Tanh() with Bipolar Input

## 1 c. Momentum Term 0.9

Setting momentum term to 0.9, and using the previous weight difference as momentum, I get an average epoch of ~2000 improvements for binary representation (almost twice as fast):

Figure 4: Binary, Momentum 0.9

Likewise, for bipolar representation using a sigmoid function, the number of epoch reduced to 100 with momentum term 0.9.

Figure 5: Bipolar Input, 0.9 MT

# Conclusion:

Using momentum, Nguyen-Widow initialization, and fast converging activation function, and bipolar inputs can speed up the convergence to find global minimum of gradient descent significantly in the tens of orders. These converge to give the weights that solve the network problem the best, as shown in Table 2. As an extra, I tried using Nguyen-Widow initialization and that sped up by only 10-30 epochs less. In conclusion, back propagation algorithm NN required many iterative steps to achieve the best weights, and the speed to convergence depends on many heuristic and engineering approach to optimize for the best method.

Table 2: Example of Resulted Weights based on Inputs

|  |  |
| --- | --- |
| Binary | Bipolar |
| WOH[0]: 3.8993797986976317  WIH[0][0]: 3.4826865641577105  WIH[1][0]: 3.6956157193411365  WIH[2][0]: -0.5116649152811875  W\_input\_bias[2][0]: -0.5116649152811875  WOH[1]: -0.6880843074370865  WIH[0][1]: 0.19584068856069192  WIH[1][1]: 0.8757659399227469  WIH[2][1]: 0.35498725526594777  W\_input\_bias[2][1]: 0.35498725526594777  WOH[2]: 4.756635852168682  WIH[0][2]: 3.0831600408510393  WIH[1][2]: -4.415488155520359  WIH[2][2]: -1.4674964481048516  W\_input\_bias[2][2]: -1.4674964481048516  WOH[3]: -4.6906791356860795  WIH[0][3]: 4.215821454710786  WIH[1][3]: -2.559936840662255  WIH[2][3]: 1.0104259122201842  W\_input\_bias[2][3]: 1.0104259122201842  W\_bias\_output: -0.7081166827349964 | WOH[0]: -0.8783855159936594  WIH[0][0]: 1.164228811436342  WIH[1][0]: -0.24957810292565577  WIH[2][0]: 0.7156117613807198  W\_input\_bias[2][0]: 0.7156117613807198  WOH[1]: 2.9122786669918743  WIH[0][1]: 2.554187509956537  WIH[1][1]: 2.3407059520026277  WIH[2][1]: 2.465502638361188  W\_input\_bias[2][1]: 2.465502638361188  WOH[2]: 2.2812351633197676  WIH[0][2]: -1.4507829701115138  WIH[1][2]: -1.8393759353115373  WIH[2][2]: 1.315116813410946  W\_input\_bias[2][2]: 1.315116813410946  WOH[3]: 0.5670768833157905  WIH[0][3]: -0.21759120782037217  WIH[1][3]: -0.33165221373735093  WIH[2][3]: -0.6089838369380356  W\_input\_bias[2][3]: -0.6089838369380356  W\_bias\_output: -1.3473490087498234 |

# Appendix:

## Neural Net Implementation of NeuralNetInterface()

**package** NeuralNet;

**import** java.io.File;

**import** java.io.IOException;

**import** java.util.Random;

**public** **class** NeuralNet **implements** NeuralNetInterface {

**public** **int** NUM\_OUTPUTS, NUM\_HIDDEN;

**public** **static** **int** *NUM\_INPUTS*, *NUM\_PATTERNS*, *NUM\_EPOCH*;

**public** **double** errThisPat, LR, MT, LB, UB;

**public** **double** errSig, weightChange = 0.0;

// Outputs

**public** **double** outputNeuron[] = **new** **double**[NUM\_OUTPUTS];

**public** **double** hiddenNeuron[] = **new** **double**[NUM\_HIDDEN]; // u = Hidden node

// outputs.

**public** **double** ba\_hiddenNeuron[] = **new** **double**[NUM\_HIDDEN];

**public** **double** weightsIH[][] = **new** **double**[*NUM\_INPUTS*][NUM\_HIDDEN]; // Input

// to

// Hidden

// weights.

**public** **double** weightsHO[] = **new** **double**[NUM\_HIDDEN]; // Hidden to Output

// weights.

**public** **double** prev\_weightsIH[][] = **new** **double**[*NUM\_INPUTS*][NUM\_HIDDEN]; // Input

// to

// Hidden

// weights.

**public** **double** prev\_weightsHO[] = **new** **double**[NUM\_HIDDEN]; // Hidden to Output

// weights.

**public** **double** weightsBO[] = **new** **double**[NUM\_OUTPUTS];

**public** **double** prev\_deltaWIH[][] = **new** **double**[*NUM\_INPUTS*][NUM\_HIDDEN];

**public** **double** prev\_deltaWHO[] = **new** **double**[NUM\_HIDDEN];

**public** **double** prev\_deltaBO[] = **new** **double**[NUM\_OUTPUTS];

**public** **double** outPred = 0.0;

**public** **boolean** NW\_flag;

// Inputs

**public** **static** **double** *beta* = 1.0;

**public** **boolean** BF;

**public** NeuralNet(**final** **int** argNumInputs, **final** **int** argNumHidden, **final** **double** argLearningRate,

**final** **double** argMomentumTerm, **final** **double** argA, **final** **double** argB, **final** **boolean** bipolar\_flag,

**final** **boolean** NW) {

NUM\_OUTPUTS = 1;

*NUM\_INPUTS* = argNumInputs + 1; // argNumInputs [3 inputs in input vector

// X]

NUM\_HIDDEN = argNumHidden; // argNumHidden [4 hidden neurons in layer,

// only 1 layer]

*NUM\_PATTERNS* = 4;

BF = bipolar\_flag;

LR = argLearningRate; // argLearningRate, Learning rate, input to hidden

// weights

MT = argMomentumTerm; // argMomentumTerm

LB = argA; // argA,lowerbound of sigmoid by output neuron only

UB = argB; // argB,upperbound of custom sigmoid by output neuron only

hiddenNeuron = **new** **double**[NUM\_HIDDEN]; // u = Hidden node outputs.

ba\_hiddenNeuron = **new** **double**[NUM\_HIDDEN];

outputNeuron = **new** **double**[NUM\_OUTPUTS];

weightsIH = **new** **double**[*NUM\_INPUTS*][NUM\_HIDDEN]; // Input to Hidden

// weights.

weightsHO = **new** **double**[NUM\_HIDDEN]; // Hidden to Output weights.

prev\_weightsIH = **new** **double**[*NUM\_INPUTS*][NUM\_HIDDEN]; // Input to Hidden

// weights.

prev\_weightsHO = **new** **double**[NUM\_HIDDEN]; // Hidden to Output weights.

weightsBO = **new** **double**[NUM\_OUTPUTS];

prev\_deltaWIH = **new** **double**[*NUM\_INPUTS*][NUM\_HIDDEN];

prev\_deltaWHO = **new** **double**[NUM\_HIDDEN];

prev\_deltaBO = **new** **double**[NUM\_OUTPUTS];

NW\_flag = NW;

**if** (NW == **true**) {

*beta* = (0.7 \* Math.*pow*(NUM\_HIDDEN, (1.0 / *NUM\_INPUTS*)));

}

}

// Internal functions

**public** **double** tanh\_sig(**final** **double** x) {

**return** ((2.0 / (1 + Math.*pow*(Math.***E***, (-1) \* x))) - 1); // [-1, 1]

}

**public** **double** sigmoid(**final** **double** x) {

**return** (1.0 / (1 + Math.*pow*(Math.***E***, (-1) \* x))); // [0, 1]

}

**public** **double** d\_tanh\_sig(**final** **double** x) {

**return** ((1 - Math.*pow*(x, 2)) / 2);

}

**public** **double** d\_sig(**final** **double** x) {

**return** x \* (1 - x);

}

**public** **double** customSigmoid(**final** **double** x) {

**return** ((UB - LB) / (1 + Math.*pow*(Math.***E***, (-1) \* x)) - LB);

}

**public** **double** squash(**double** x) {

**if** (!BF) {

**if** (x >= 0.8) {

x = 1.0;

} **else** **if** (x <= 0.2) {

x = 0.0;

}

} **else** {

**if** (x >= 0.5) {

x = 1.0;

} **else** **if** (x <= -0.5) {

x = -1.0;

}

}

**return** x;

}

**public** **void** initializeWeights() {

// input vector X [0]=-1/1, [1]=1/-1, [2]=1

**double** InputsNorm = 0.0;

**for** (**int** j = 0; j < NUM\_HIDDEN; j++) {

weightsHO[j] = (**new** Random().nextDouble() - 0.5); // [-0.5, 0.5]

InputsNorm += Math.*pow*(weightsHO[j], 2);

System.***out***.println("HO Weight = " + weightsHO[j]);

**for** (**int** i = 0; i < *NUM\_INPUTS*; i++) {

weightsIH[i][j] = (**new** Random().nextDouble() - 0.5); // [-0.5,

// 0.5]

System.***out***.println("IH Weight = " + weightsIH[i][j]);

InputsNorm += Math.*pow*(weightsIH[i][j], 2);

}

}

weightsBO[0] = (**new** Random().nextDouble() - 0.5);

// Nguyen Widrow Adjustment

InputsNorm = Math.*sqrt*(InputsNorm);

**if** (NW\_flag == **true**) {

**for** (**int** j = 0; j < NUM\_HIDDEN; j++) {

weightsHO[j] = (*beta* \* weightsHO[j]) / InputsNorm;

**for** (**int** i = 0; i < *NUM\_INPUTS*; i++) {

weightsIH[i][j] = (*beta* \* weightsIH[i][j]) / InputsNorm;

}

}

}

**return**;

}

**public** **void** zeroWeights() {

**return**;

}

**public** **double** outputFor(**final** **double**[] X) {

outPred = 0.0;

// Calculate hidden nerons' activations (Forward propagation)

**for** (**int** i = 0; i < NUM\_HIDDEN; i++) {

hiddenNeuron[i] = 0.0;

**for** (**int** j = 0; j < *NUM\_INPUTS*; j++) {// size of input

// vector(including bias)

hiddenNeuron[i] += (X[j] \* weightsIH[j][i]);

}

ba\_hiddenNeuron[i] = hiddenNeuron[i];

hiddenNeuron[i] = BF ? tanh\_sig(hiddenNeuron[i]) : sigmoid(hiddenNeuron[i]);

}

// Calculate output neuron value (Forward propagation)

**for** (**int** i = 0; i < NUM\_HIDDEN; i++) {

outPred += (hiddenNeuron[i] \* weightsHO[i]);

}

// Calculate bias term from input to outputneuron

**for** (**int** i = 0; i < NUM\_OUTPUTS; i++) {

outPred += 1.0 \* weightsBO[i];

}

outputNeuron[0] = (BF ? tanh\_sig(outPred) : sigmoid(outPred));

**return** outputNeuron[0]; // if outputFor() will be iterated across # of

// neurons

}

**public** **double** train(**final** **double**[] X, **final** **double** argValue) {

outputNeuron[0] = outputFor(X);

errThisPat = (argValue - outputNeuron[0]);

errSig = BF ? d\_tanh\_sig(outputNeuron[0]) : d\_sig(outputNeuron[0]);

errSig = errThisPat \* errSig;

// Calculate weightHO changes based on errOutput/fprime\_err

**for** (**int** k = 0; k < NUM\_HIDDEN; k++) {

**final** **double** deltaweight = LR \* errSig;

**final** **double** x = deltaweight \* hiddenNeuron[k];

weightChange = x + (MT \* (prev\_deltaWHO[k]));

prev\_weightsHO[k] = weightsHO[k]; // store t-1 weights

weightsHO[k] += weightChange; // update t weight

prev\_deltaWHO[k] = weightChange;// store t weight

}

**for** (**int** i = 0; i < NUM\_OUTPUTS; i++) { // update output to bias weight

**final** **double** deltaweight = LR \* errSig;

**final** **double** x = deltaweight \* 1.0;

weightChange = x + (MT \* (prev\_deltaBO[i]));

weightsBO[i] += weightChange;

prev\_deltaBO[i] = weightChange;// store previous bias weight change

}

// Calculate weightIH changes based on weightsHO

**for** (**int** i = 0; i < NUM\_HIDDEN; i++) {

**for** (**int** j = 0; j < *NUM\_INPUTS*; j++) {

**final** **double** x = BF ? d\_tanh\_sig(hiddenNeuron[i]) : d\_sig(hiddenNeuron[i]);

**final** **double** deltaweight = LR \* errSig \* x \* (weightsHO[i]);

weightChange = deltaweight \* X[j];

weightChange = weightChange + (MT \* (prev\_deltaWIH[j][i]));

prev\_weightsIH[j][i] = weightsIH[j][i]; // t current set of

// weigths

weightsIH[j][i] += weightChange; // t+1

prev\_deltaWIH[j][i] = weightChange; // t-1 store prev weight

}

}

outputNeuron[0] = outputFor(X); // recalculate u0

errThisPat = argValue - outputNeuron[0];

**return** (errThisPat); // return error for training pattern

}

@Override

**public** **void** save(**final** File argFile) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** load(**final** String argFileName) **throws** IOException {

// **TODO** Auto-generated method stub

}

}

## Main function code that implements NeuralNetInterface

**package** NeuralNet;

**import** java.io.\*;

**import** java.util.Random;

**public** **class** XOR\_BP {

**static** **int** *NUM\_INPUTS* = 3;

**static** **int** *NUM\_EPOCH* = 0;

**static** **int** *NUM\_PATTERNS* = 4;

**static** **int** *NUM\_OUTPUTS* = 1;

**public** **static** **double** *trainInputs*[][] = **new** **double**[*NUM\_PATTERNS*][*NUM\_INPUTS*];

**public** **static** **double** *trainOutput*[] = **new** **double**[*NUM\_PATTERNS*]; // "Actual"

// output

// values.

**public** **static** **double** *errOut*[] = **new** **double**[*NUM\_EPOCH*];

**public** **static** **double** *errorOut*;

**public** **static** **double** *outNeuron*[] = **new** **double**[*NUM\_OUTPUTS*];

**private** **static** **double** *RMSerror* = 1.0;

**static** **boolean** *bipolar\_flag* = **true**; // set this flag for binary/bipolar

**private** **static** **void** init() {

*trainInputs*[0][0] = 1;

*trainInputs*[0][1] = -1;

*trainInputs*[0][2] = 1; // Bias

*trainOutput*[0] = 1;

*trainInputs*[1][0] = -1;

*trainInputs*[1][1] = 1;

*trainInputs*[1][2] = 1; // Bias

*trainOutput*[1] = 1;

*trainInputs*[2][0] = 1;

*trainInputs*[2][1] = 1;

*trainInputs*[2][2] = 1; // Bias

*trainOutput*[2] = -1;

*trainInputs*[3][0] = -1;

*trainInputs*[3][1] = -1;

*trainInputs*[3][2] = 1; // Bias

*trainOutput*[3] = -1;

}

**private** **static** **void** initBinaryData() {

*trainInputs*[0][0] = 1;

*trainInputs*[0][1] = 0;

*trainInputs*[0][2] = 1; // Bias

*trainOutput*[0] = 1;

*trainInputs*[1][0] = 0;

*trainInputs*[1][1] = 1;

*trainInputs*[1][2] = 1; // Bias

*trainOutput*[1] = 1;

*trainInputs*[2][0] = 1;

*trainInputs*[2][1] = 1;

*trainInputs*[2][2] = 1; // Bias

*trainOutput*[2] = 0;

*trainInputs*[3][0] = 0;

*trainInputs*[3][1] = 0;

*trainInputs*[3][2] = 1; // Bias

*trainOutput*[3] = 0;

}

**public** **static** **double** squash(**double** x) {

**if** (!*bipolar\_flag*) {

**if** (x > 0.5) {

x = 1.0;

} **else** **if** (x < 0.5) {

x = 0.0;

}

} **else** {

**if** (x > 0) {

x = 1.0;

} **else** **if** (x < 0) {

x = -1.0;

}

}

**return** x;

}

// Main function

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

**int** patNum = 0;

**double** errorValue = 1; // initialize errorValue high

**double** err = 0.0;

**final** **double** errthresh = 0.02;

String content\_in = System.*getProperty*("line.separator");

**final** **int** num\_trials = 15;

**final** **double** arrayEpoch[] = **new** **double**[num\_trials];

**int** wrong = 0, right = 0;

//// MAIN FUNCTION ///

**for** (**int** t = 0; t < num\_trials; t++) {

**final** NeuralNetInterface nn\_if = **new** NeuralNet(2, // num inputs

4, // num hidden (without bias)

0.2, // learning rate

0.9, // momentum

-1, // lower bound

1, // upper bound

*bipolar\_flag*, // bipolar flag

**false** // Nguyen-Widrow

);

**final** File file = **new** File("C://Users/vpwong/Google Drive/backprop/data" + t + ".xls");

**if** (!file.exists()) {

file.createNewFile();

}

**final** FileWriter fw = **new** FileWriter(file.getAbsoluteFile());

**final** BufferedWriter bw = **new** BufferedWriter(fw);

*NUM\_EPOCH* = 0; // reset NUM\_EPOCH

nn\_if.zeroWeights();

nn\_if.initializeWeights();

**if** (*bipolar\_flag*)

*init*();

**else**

*initBinaryData*();

// Train the network.

errorValue = 1;

**while** (errorValue >= errthresh) {

errorValue = 0;

**for** (**int** k = 0; k < *NUM\_PATTERNS*; k++) {

patNum = **new** Random().nextInt(4);

err = nn\_if.train(*trainInputs*[patNum], *trainOutput*[patNum]);

errorValue += Math.*pow*(err, 2);

}

errorValue /= 2; // total error calculation

*NUM\_EPOCH*++;

*RMSerror* = Math.*sqrt*(errorValue);

System.***out***.println("epoch = " + *NUM\_EPOCH* + " Error = " + errorValue + " rms err = " + *RMSerror*);

content\_in = *NUM\_EPOCH* + "," + errorValue;

**try** {

bw.write(content\_in + "\n");

} **catch** (**final** IOException e) {

e.printStackTrace();

}

}

arrayEpoch[t] = *NUM\_EPOCH*;

bw.close();

**for** (**int** i = 0; i < *NUM\_PATTERNS*; i++) {

**final** **double** final\_outNeuron = nn\_if.outputFor(*trainInputs*[i]);

**final** **double** ceil\_neuron = *squash*(final\_outNeuron);

System.***out***.println(

"pat = " + (i + 1) + " actual = " + *trainOutput*[i] + " neural model = " + final\_outNeuron);

**if** (ceil\_neuron != *trainOutput*[i]) {

wrong++;

} **else** {

right++;

}

content\_in = "pat = " + (i + 1) + " actual = " + *trainOutput*[i] + " neural model = " + final\_outNeuron;

}

}

**final** Statistics stat = **new** Statistics(arrayEpoch);

System.***out***.println("Mean Num Epoch = " + stat.getMean());

System.***out***.println("Standard Deviation Num Epoch = " + stat.getStdDev());

System.***out***.println("Num Epoch variance " + stat.getVariance());

System.***out***.println("accuracy percentage " + (right \* 100) / (wrong + right));

System.***out***.println("Median Epochs " + stat.median());

**return**;

}

}