AI Research Project

Create a Multilayer Perceptron Network

Zhenyan Liu

04/16/2019

1 INTRODUCTION AND BRIEF TECHNICAL DISCUSSION

As technology develops quickly, Artificial Intelligence has been paid more attention by all kinds of organizations. More and more researches have been done. In Artificial Intelligence, neural network is an important part among the current researches. In my project, I did research about the multilayer perceptron network and will discuss about it in this paper.

1.1 INTRODUCTION

Neural networks are a subset of machine learning which are inspired by human brains. In nowadays, the Neural Networks have been widely used in different kind of areas. Neural networks with enough training can be considered as great ‘brains’ for different kinds of classifier problems. Trained neural networks will help us to perform clustering and classifying with certain inputs. Also, neural networks have developed quickly. As we learned in class, the total number of neurons doubles every 2.4 years. Although we discussed a lot about the neural networks and deep learnings in class, we still did not have any homework or discussions about to the neural networks or deep learnings. Besides the neural network is almost the hottest topic in the AI area, so in this project I try to create a basic Multilayer Perceptron network to solve the basic classify problems. My code will follow the Multilayer Perceptron network structure which we have learned in class.

* 1. TECHNICAL DISCUSSION

Neural network is a network of connected neurons. Between each connected neuron, data can be transmitd one way or two ways. The neurons receive the data and will process the data. Compared with single perceptron, neural network can create the non-linear boundaries which means the neural can handle more complicate problems.

The most popular neural network is Multilayer Perceptron. Multilayer Perceptron is categorized as geometric rules in nonparametric approach in supervised learning which is an incomplete Classification/Clustering method.

Multilayer Perceptron is a kind of feedforward network which means the data will be only sent to next layer from current layer and cannot be sent back. Multilayer Perceptron is static compared to Recurrent Network which is dynamic since the Multilayer Perceptron is always stable. Neurons in Multilayer Perceptron can be categorized into three different types: 1. Input Layers; 2. Hidden Layers; 3. Output Layers.

The first layer of Multilayer Perceptron network is the input layer. The input layer will pass the input data to the hidden layer. The input layer will not make any changes to the inputs which means the function of the input layer is just to pass the data and won’t do any data processing.

The last layer of Multilayer Perceptron network is the output layer. The output layer will help to process the data from hidden layer and can provide the final outputs.

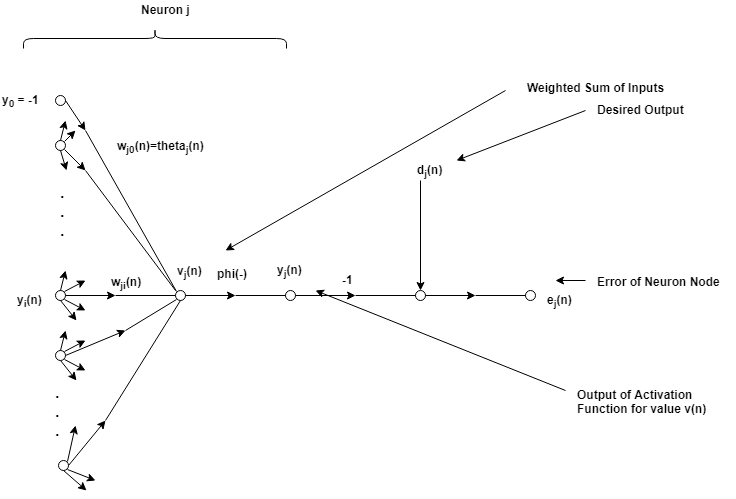
The layer/layers between the first layer and last layer are all hidden among layer/layers. The hidden layer will sum up the inputs with weights from previous layer and send the output to the next layer.

Except the layers, the links between the layers are really important. The links contain the weights for each node. The weights on the links will be updated with the errors between output values and target values. To update the weights of each node, we need the training algorithm to process the output values and target values.

Although the Multilayer Perceptron network is a feedforward network, we still need to find the way to train the whole network. The training algorithm applied in Multilayer Perceptron network is Back-Propagations which means the backward error propagation. In Back-propagation algorithm, the error of each training sample in each node output will change the weights of each link.

The Back-Propagation follows the delta rule which means to get the new weights of the old weights and derivative of weights. The equation is:

Based on the chain rules as we learned in the class, after calculating will be converted to the following equation:



is the learning rate. is the error. is the derivative of the activation function.

is the output of the activation function and it has the following equation:

However, compared with single perceptron system 1 and -1 thresholding, the Multilayer Perceptron network has more smooth thresholding functions which are also named as activation function. In activation function, the region near 0 will not have the distinguish hard limit style which will help us to obtain the derivative of activation function more easily. We know that the previous perceptron will have an infinity derivative at 0. In the region away from 0, the smooth activation function will have the hard-limited threshold. After I did several researches online, I find that we have several activation functions. For example, the activation function can be step functions, sigmoid function or tanh function.

Also, since the weight can be caught in the local minimum, we can add the momentum to let the weight go out of the local minimum. The weight without adding momentum will follow the following equation:

.

However, with adding momentum the equation will be the following:

.

As we learned in class, the Multilayer Perceptron network can have more than one hidden node and more than one hidden layer. However, increasing the layer or nodes will cost more computational resources.

* 1. CODE & TEST INTRODUCTION

My project is to create a basic Multilayer Perceptron network code which can be performed as a basic classifier for certain equations. With this Multilayer Perceptron network, I will test how the total nodes number will affect the convergence of certain problem. The technology tools I used in the project is as following:

Code: C++ code

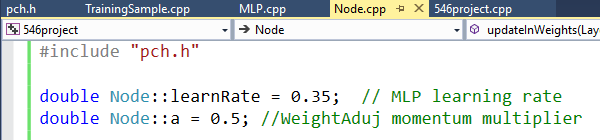
IDE: Microsoft Visual Studio Community 2017

Code classes: 1. MLP; 2. Node; 3. Training Sample

Class MLP will handle the data process between different layers which will perform the feedforward and back-propagation.

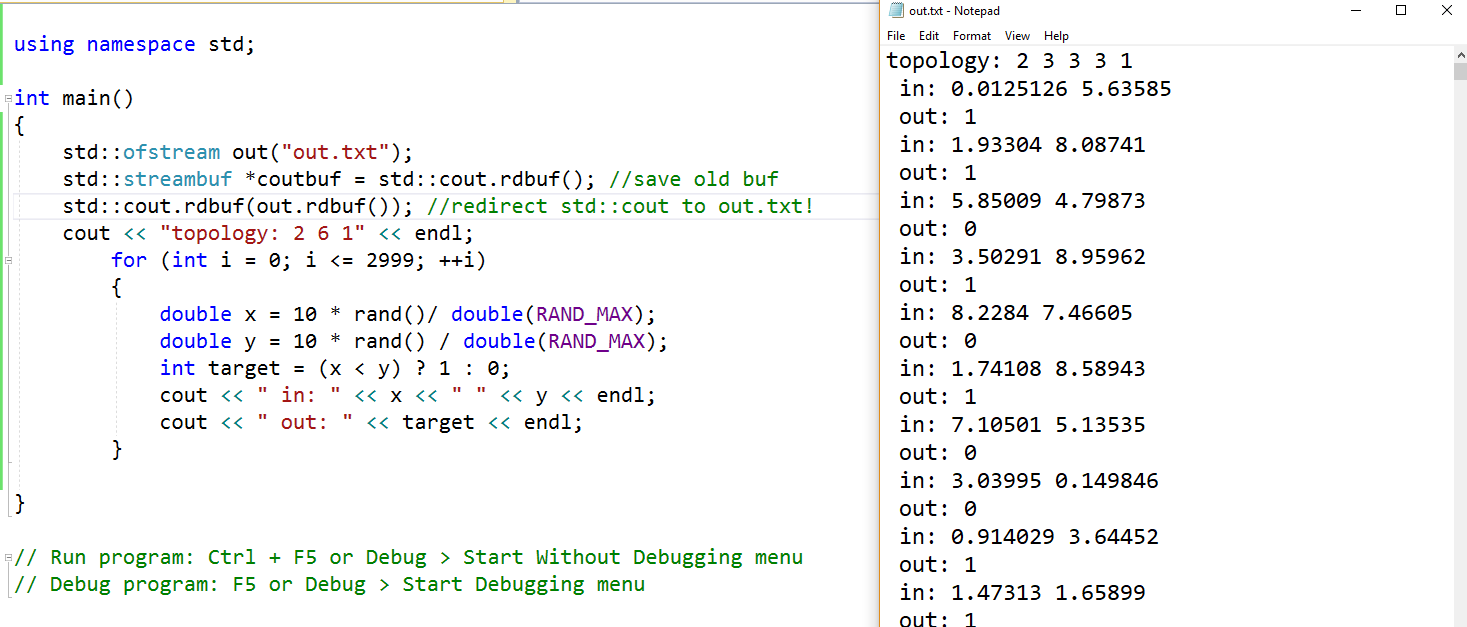
Class node will help the node to calculate the input value and output value for each node and derivative of the weight and new weight of each link. The derivative equation is calculated in this class node. The code has the tanh function as the activation function in the class node.

In class node, we have the two important parameters which are MLP learning rate and the add momentum adjuster parameter. For all the tests below, I assign the learning rate as 0.15 and add weight parameter as 0.5.



Class Training Sample will read in the training samples. Also, it will read in the topology of the Multilayer Perceptron network.

I test three different training samples with the code. To generate the training samples, I create code “TrainingDataGenerator” to generate the training samples. In each sample, the first line will be the topology of the Multilayer Perceptron network. The rest of the file follow the format: in: x.xxx out:x.xxx. For each sample, we will have 3000 training points.



***First sample:*** if y – x < 0, target = 0; if y – x >= 0, target = 1.

(0 <=x(int) <10; 0 <= y(int) <10)

***Second sample:*** if y – x^2 < 0, target = 0; if y – x^2 >= 0, target = 1.

(x =1,2,3; 0 <= y(int) <5)

***Third sample:*** target = x \* y.

(x = 1 or 0; y = 1 or 0)

The reason why I choose these three test samples is because the first has a liner classifier boundary, the second is a non-linear boundary classifier and the third sample test is the logical equation boundary. However, all the inputs of the three sample tests have the integer inputs. In fact, I also test a continuous input but it is not converged. I will discuss the continuous sample test in the result section too.

***The continues input test sample is:*** if y – x < 0, target = 0; if y – x >= 0, target = 1.

(0 <=x(**double**) <1; 0 <= y(**double**) <1).

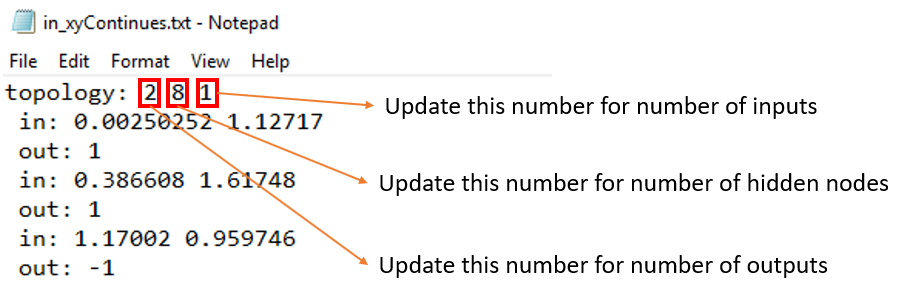
Within each sample, I also test three different Multilayer Perceptron networks with different numbers of hidden nodes. By these three inputs tests, we can tell how the nodes numbers will affect the Multilayer Perceptron network performance.

First topology: 2 inputs nodes, 2 nodes on hidden layer and 1 output node.

Second topology: 2 inputs nodes, 4 nodes on hidden layer and 1 output node.

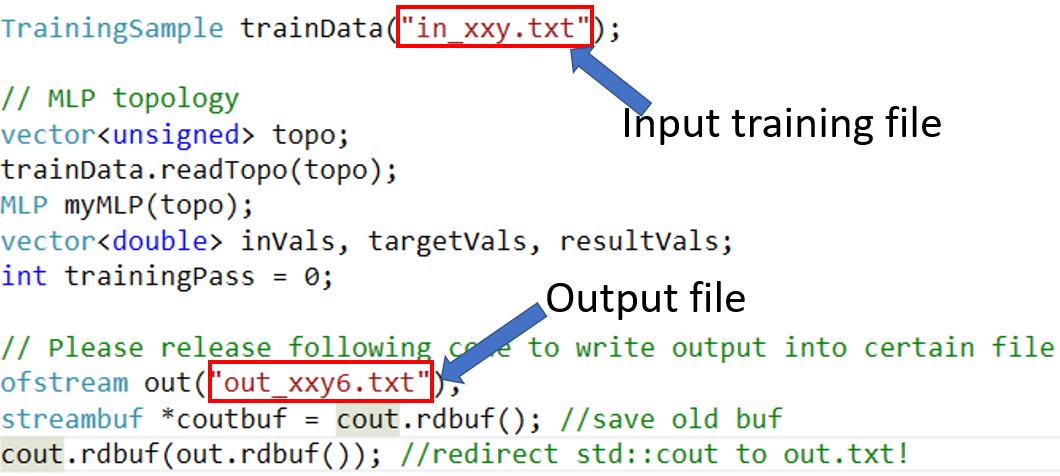
Third topology: 2 inputs nodes, 6 nodes on hidden layer and 1 output node.

To update the total hidden nodes numbers, input numbers and output numbers, we need to modify the first line of input files. For example, the following file has 2 inputs, 8 hidden nodes and 1 output.



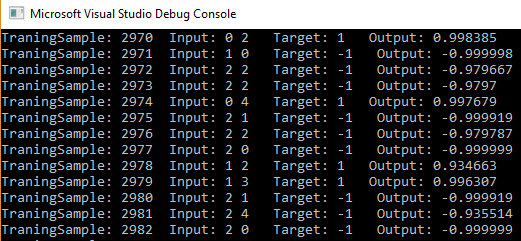
1. RESULTS

After running three samples with 2 nodes, 4 nodes and 6 nodes, Multilayer Perceptron network, 9 files have been created. We can control the output file names in the code:



Then I open the files in Excel and plot out the sample numbers and error plots. With each plot, we can easily tell when the Multilayer Perceptron network converge. I will attach the nine output txt files in the zip package.

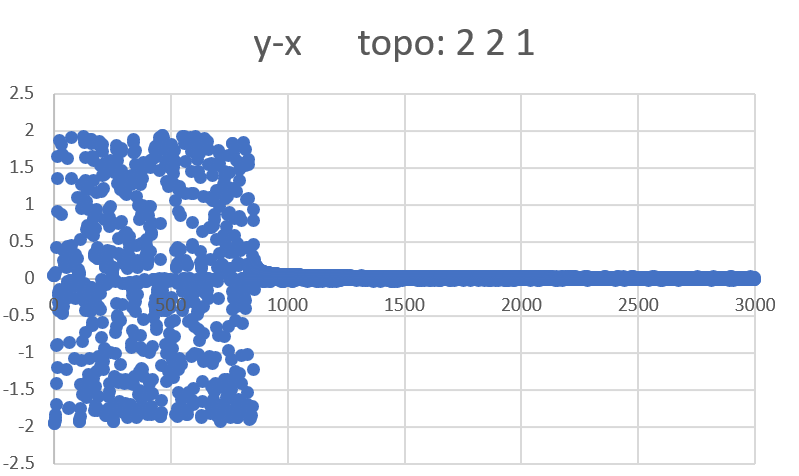
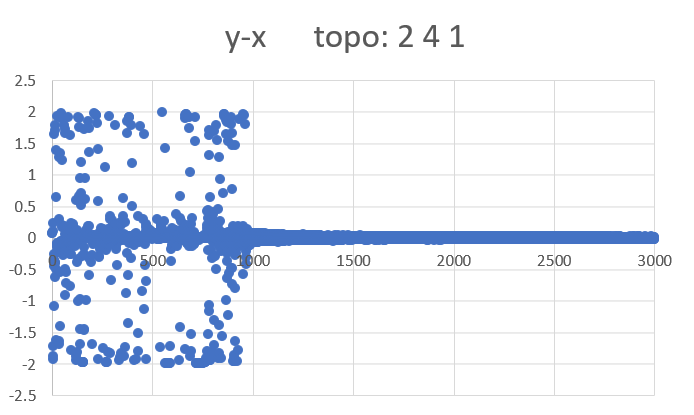
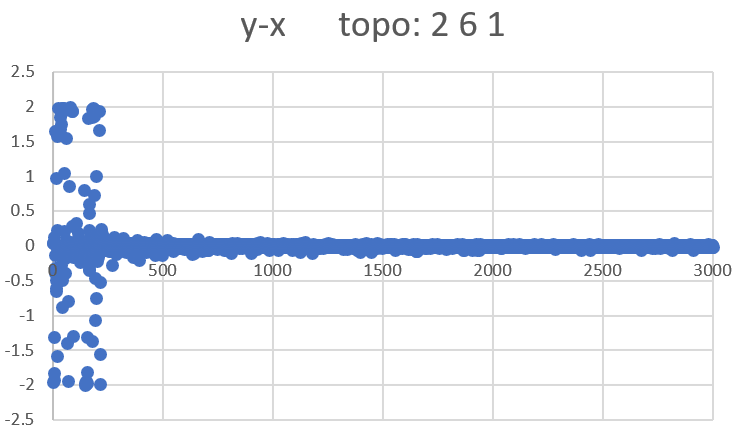
If we do not export the data, we can have the print out window as:



For each sample test, error is calculated as: error = target value – output value.

***Result of first sample***: following three pictures are the sample number - error plot.

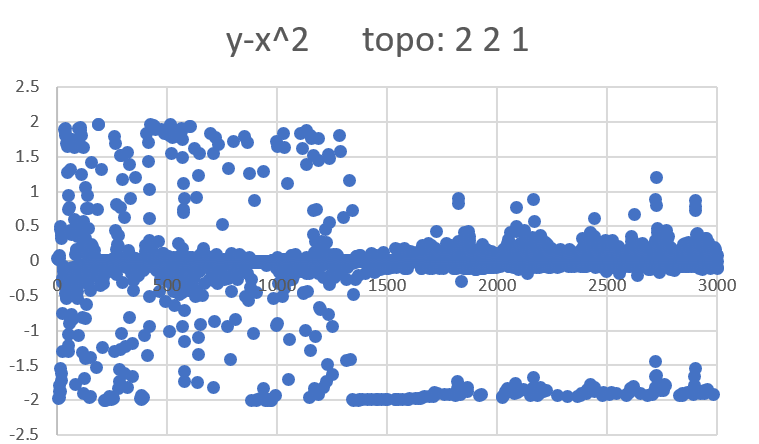
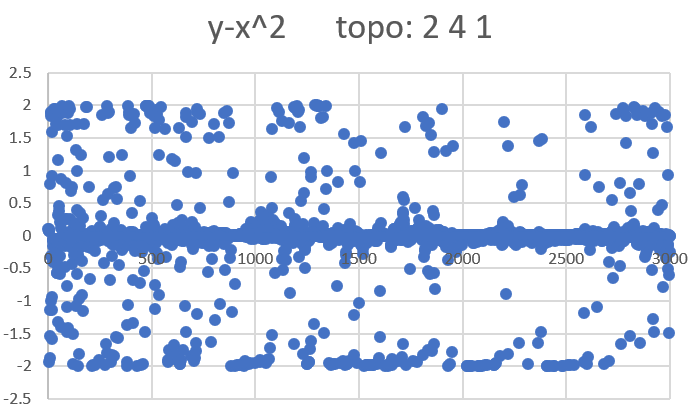
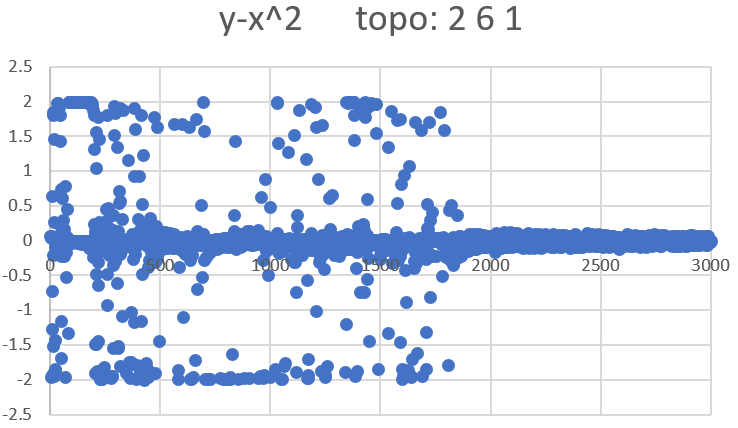
if y – x < 0, target = 0; if y – x >= 0, target = 1. (0 <=x (int)<10; 0 <= y (int)<10)

As we can see: for 2 and 4 hidden nodes, the results converge around 1000 samples training. For 6 hidden nodes network, the result converges at 250 samples training.

***Result of second sample:*** following three pictures are the sample number - error plot.

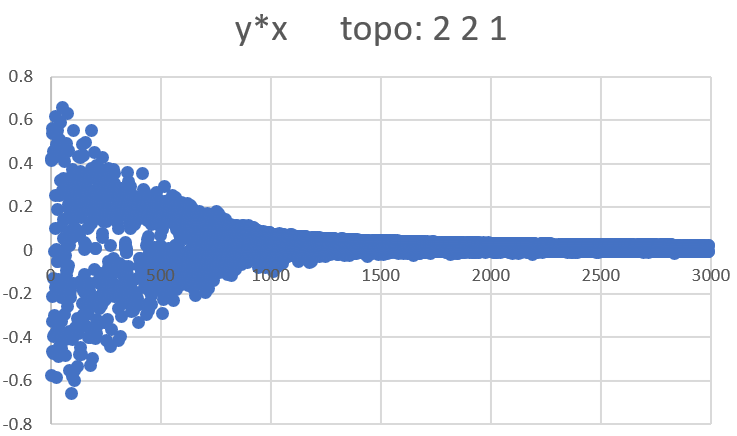
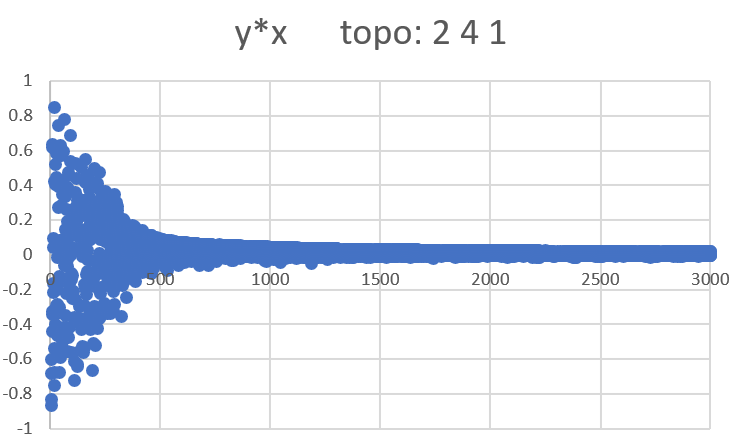
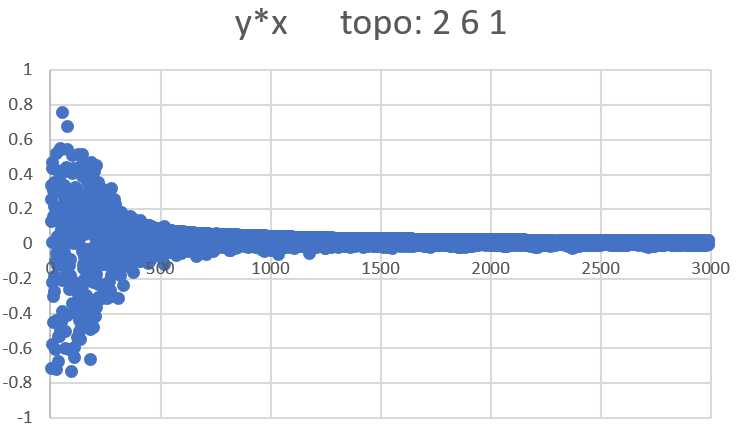
if y – x^2 < 0, target = 0; if y – x^2 >= 0, target = 1. (0 <=x (int)<3; 0 <= y (int)<5)

As we can see: for 2 and 4 hidden nodes, the results do not converge with 3000 samples training. For 6 hidden nodes network, the result converges at 1800 samples training.

***Result of third sample:*** following three pictures is the sample number - error plot.

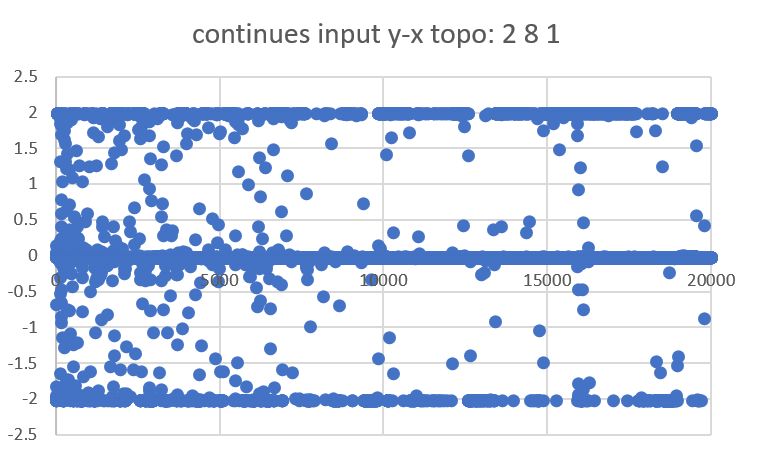
target = x \* y; (x = 1 or 0; y = 1 or 0)

As we can see: for 2 hidden nodes, the results do not converge with 1250 samples training. For 4 and 6 hidden nodes network, the result converges at 500 samples training.

I also test a continuous input sample: if y – x < 0, target = 0; if y – x >= 0, target = 1. (0 <=x(**double**) <1; 0 <= y(**double**) <1). However, even with 20000 training points and 8 nodes, it is still not converged. From my opinion, I think continuous inputs will induce huge input combinations which provide the difficulties for converging. For example, x = 1.11 and x = 1.12 are totally different inputs. I also attach the output txt files in the zip package. We need more technology tools to improve the neuron networks.

The following picture shows the result of 20,000 training points and 8 hidden nodes which is not converged:



1. CONCLUSIONS AND AREAS OF POSSIBLE FURTURE WORK

Firstly, the Multilayer Perceptron network code can successfully classify the three integer inputs in problem I test. In all the three sample sets with integer inputs, all the 6 hidden nodes Multilayer Perceptron network can have coverage results within 3000 training points.

With all these three integer inputs sample tests, we can clearly figure out that more nodes will help the Multilayer Perceptron network to converge with less training samples. Especially as we can see in the second sample, the 2 and 4 nodes network cannot converge within 3,000 samples training. However, the 6 hidden layer nodes Multilayer Perceptron network can have the converge result for the y-x^2 classifier.

However, for the continuous input test *if y – x < 0, target = 0; if y – x >= 0, target = 1. (0 <=x(****double****) <1; 0 <= y(****double****) <1)*, I cannot have converged results. Although in the result section, I only show the 8 nodes and 20,000 sample points test result, I run different kinds of test to reach a converge result. I run 2 nodes, 4 node and 6 nodes with different learning rate (0.15, 0.25, 0.35, 05) and different adding momentum coefficient (0.4, 0.5, 0.6) but none of them converged. To have a converged result, I have the following ideas:

I may need to run an even larger training sample. Maybe more learning samples will cover more inputs combination which will lead to a converge result.

I may need to add second hidden layer. Making the MLP deeper with at least one more hidden layer may guide the MLP be ‘smarter’.

Perform the pretraining of the system.

Another interested thing I can do is to test with the certain nodes how will the topology affect the result. For example, is one layer with 6 nodes better or three layers with 2 nodes in each layer better? As we learned in the class, the general rules for node topology is:

In the future, I can also can try more inputs or more output sample test with the Multilayer Perceptron network. For example, if z – y – x < 0, target = 0; if z – y – x >= 0, target = 1. (0 <=x (int)<10; 0 <= y (int)<10; 0 <= y (int)<10).

1. REFERENCES

[1] Haykin, S. Neural Networks: A Comprehensive Foundation. 2nd edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1998)

[2] Hua, W., Zhang, Z., Suh, G.E. Reverse engineering convolutional neural networks through side-channel information leaks) (2018)

[3] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS ’17, New York, NY, USA, ACM (2017) 363–376[4] Picek, S., Heuser, A., Jovic, A., Ludwig, S.A., Guilley, S., Jakobovic, D., Mentens, N.. Side-channel analysis and machine learning: A practical perspective. In: Neural Networks (IJCNN), 2017 International Joint Conference on, IEEE (2017) 4095–4102[5] Gilmore, R., Hanley, N., O’Neill, M.. Neural network based attack on a masked implementation of AES. In: 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). (May 2015) 106–111

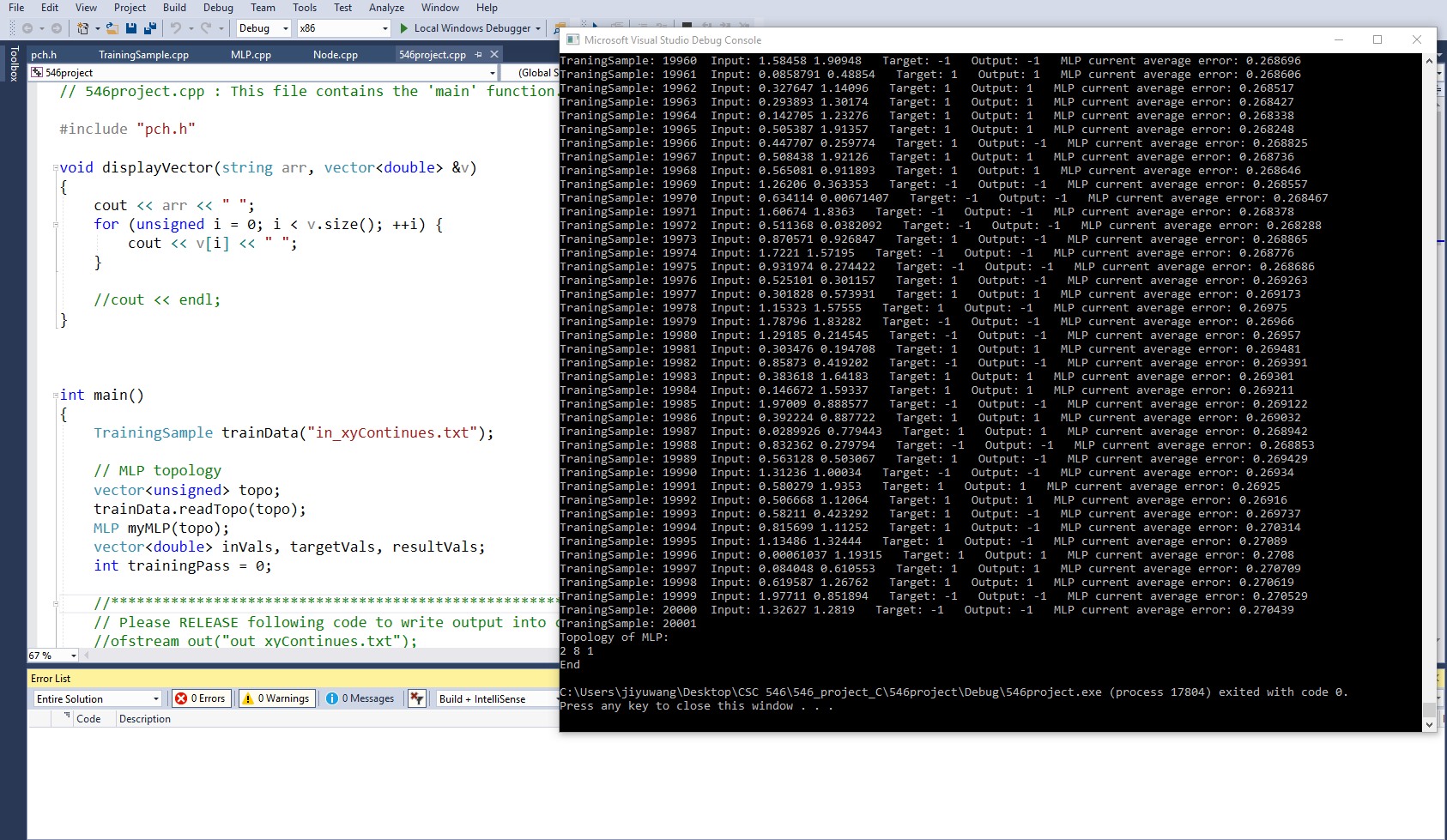
Files attached in the zip file:

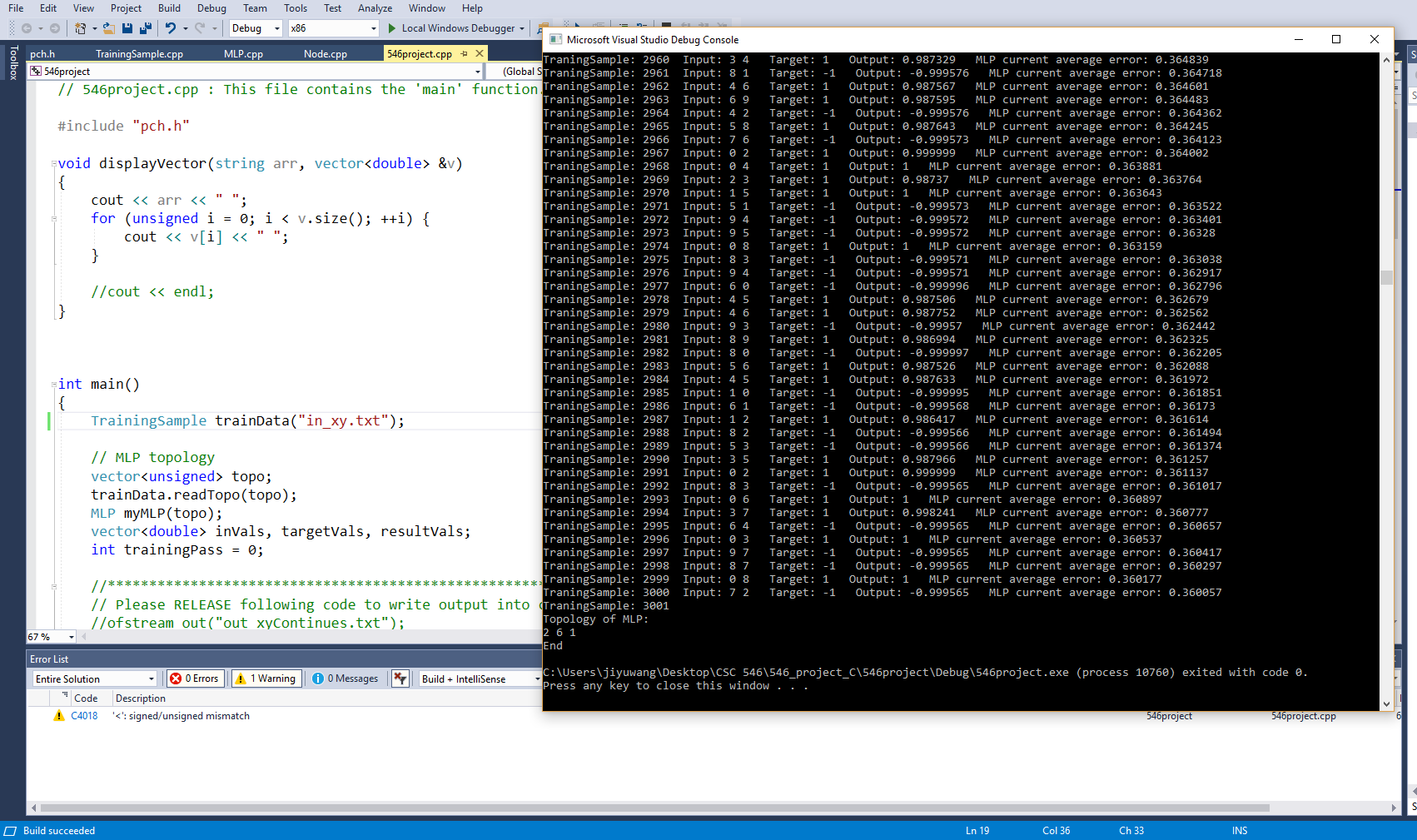
1. MLP Source code: files in folder ***546project***

This is the main program that have all the MLP classes.

1. TrainingDataGenerator Source code: files in folder ***TrainingDataGenerator***  This is the program that generate the training data points file.
2. Training files I tested: files in folder ***Input files***
3. in\_and.txt input file for case target = int x \* y
4. in\_xy.txt input file for case target = int y - int x > 0 or < 0
5. in\_xxy.txt input file for case target = int y - int x\*x > 0 or < 0
6. in\_xyContinues input file for case target = double y - double x > 0 or < 0
7. Output files generated: files in folder ***Output files***
8. out\_and2.txt output file for case target = int x \* y with 2 hidden nodes
9. out\_and4.txt output file for case target = int x \* y with 4 hidden nodes
10. out\_and6.txt output file for case target = int x \* y with 6 hidden nodes
11. out \_xy2.txt output file for case target = int y - int x > 0 or < 0 with 2 hidden nodes
12. out \_xy4.txt output file for case target = int y - int x > 0 or < 0 with 4 hidden nodes
13. out \_xy6.txt output file for case target = int y - int x > 0 or < 0 with 6 hidden nodes
14. out\_xxy2.txt output file for case target = int y - int x\*x > 0 or < 0 with 2 hidden nodes
15. out\_xxy4.txt output file for case target = int y - int x\*x > 0 or < 0 with 4 hidden nodes
16. out\_xxy6.txt output file for case target = int y - int x\*x > 0 or < 0 with 6 hidden nodes
17. out\_xyContinues output file for case target = double y - double x > 0 or < 0 with 8 hidden nodes

Screenshot prove program is running:





1. APPENDIX OF CODE: (I attached the key part of the codes)

***pch.h***

#ifndef PCH\_H

#define PCH\_H

#include <cmath>

#include <cstdlib>

#include <iostream>

#include <vector>

#include <cassert>

#include <fstream>

#include <sstream>

using namespace std;

struct Link

{

double weight;

double WeightAduj;

};

class Node;

typedef vector<Node> Layer;

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

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* class Node \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \\

// |||||||||||||||||||||||||||||||| start here |||||||||||||||||||||||||||||||| \\

class Node

{

public:

Node(unsigned totalOuts, unsigned nodeID);

void setOutVal(double val) { curr\_outVal = val; }

double getOutVal(void) const { return curr\_outVal; }

void feedForward(const Layer &fromLayer);

void derivativesOut(double targetVal);

void derivativesHidden(const Layer &toLayer);

void updateInWeights(Layer &fromLayer);

private:

static double learnRate; // learning rate

static double a; // weight change momentum

static double actiFunc(double x);

static double actiFuncDerivative(double x);

static double randomWeight(void) { return rand() / double(RAND\_MAX); }

double sumDerivative(const Layer &toLayer) const;

double curr\_outVal;

vector<Link> curr\_outWeights;

unsigned curr\_nodeID;

double curr\_Derivative;

};

// |||||||||||||||||||||||||||||||| end here |||||||||||||||||||||||||||||||| \\

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* class Node \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \\

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

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

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* class MLP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \\

// |||||||||||||||||||||||||||||||| start here |||||||||||||||||||||||||||||||| \\

class MLP

{

public:

MLP(const vector<unsigned> &topo);

void feedForward(const vector<double> &inVals);

void backProp(const vector<double> &targetVals);

void getResults(vector<double> &resultVals) const;

double getRecentAverageError(void) const { return curr\_recentAverageError; }

private:

//curr\_layers is a matrix, can access to certain node with [layerID][NodeNum]

vector<Layer> curr\_layers;

double curr\_error;

double curr\_recentAverageError;

static double curr\_recentAverageSmoothingFactor;

};

// |||||||||||||||||||||||||||||||| end here |||||||||||||||||||||||||||||||| \\

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* class MLP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \\

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

// read in data from training data.

class TrainingSample

{

public:

TrainingSample(const string filename);

bool fileEof(void) { return curr\_trainingSampleFile.eof(); }

void readTopo(vector<unsigned> &topo); // read in MLP topo from training data.

unsigned readNextIns(vector<double> &inVals); // read in input vals from training data

unsigned readTargetOuts(vector<double> &targetOutVals); // read in target vals from training data.

private:

ifstream curr\_trainingSampleFile;

};

// TODO: add headers that you want to pre-compile here

#endif //PCH\_H

***546project.cpp***

// 546project.cpp : This file contains the 'main' function. Program execution begins and ends there.

#include "pch.h"

void displayVector(string arr, vector<double> &v)

{

cout << arr << " ";

for (unsigned i = 0; i < v.size(); ++i) {

cout << v[i] << " ";

}

//cout << endl;

}

int main()

{

TrainingSample trainData("in\_xyContinues.txt");

// MLP topology

vector<unsigned> topo;

trainData.readTopo(topo);

MLP myMLP(topo);

vector<double> inVals, targetVals, resultVals;

int trainingPass = 0;

// Please release following code to write output into certain file

ofstream out("out\_xyContinues.txt");

streambuf \*coutbuf = cout.rdbuf(); //save old buf

cout.rdbuf(out.rdbuf()); //redirect std::cout to out.txt!

while (!trainData.fileEof()) {

++trainingPass;

cout<< "TraningSample: " << trainingPass;

// Read in new data and feed forward the data

if (trainData.readNextIns(inVals) != topo[0]) {

break;

}

displayVector(" Input:", inVals);

myMLP.feedForward(inVals);

//MLP training with target vals:

trainData.readTargetOuts(targetVals);

displayVector(" Target:", targetVals);

assert(targetVals.size() == topo.back());

// get the MLP calculated results:

myMLP.getResults(resultVals);

displayVector(" Output:", resultVals);

myMLP.backProp(targetVals);

cout << " MLP current average error: "

<< myMLP.getRecentAverageError() << endl;

}

cout << endl << "Topology of MLP: " << endl;

for (int i = 0; i < topo.size(); ++i)

{

cout << topo[i] << ' ';

}

cout << endl << "End" << endl;

}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu

// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:

// 1. Use the Solution Explorer window to add/manage files

// 2. Use the Team Explorer window to connect to source control

// 3. Use the Out window to see build out and other messages

// 4. Use the Error List window to view errors

// 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project

// 6. In the future, to open this project again, go to File > Open > Project and select the .sln file

***MLP.cpp***

#include "pch.h"

double MLP::curr\_recentAverageSmoothingFactor = 3000.0; // Total counts of training samples for averaging

void MLP::getResults(vector<double> &resultVals) const

{

resultVals.clear();

for (unsigned n = 0; n < curr\_layers.back().size() - 1; ++n) {

resultVals.push\_back(curr\_layers.back()[n].getOutVal());

}

}

// Back-propagation

void MLP::backProp(const vector<double> &targetVals)

{

// Calculate total MLP error which is (sum Node errors)^0.5

Layer &outLayer = curr\_layers.back();

curr\_error = 0.0;

for (unsigned n = 0; n < outLayer.size() - 1; ++n)

{

double delta = targetVals[n] - outLayer[n].getOutVal();

curr\_error += delta \* delta; //error^2

}

curr\_error /= outLayer.size() - 1; // average error^2

curr\_error = sqrt(curr\_error); // (error^2)^0.5

// Calculate current average error

// currAveError = (preSumError + currError)/(PrevTotalSamples +1)

curr\_recentAverageError =

(curr\_recentAverageError \* curr\_recentAverageSmoothingFactor + curr\_error)

/ (curr\_recentAverageSmoothingFactor + 1.0);

// Calculate out layer Derivatives

for (unsigned n = 0; n < outLayer.size() - 1; ++n)

{

outLayer[n].derivativesOut(targetVals[n]);

}

// Calculate Derivatives of hidden layers

for (unsigned layerID = curr\_layers.size() - 2; layerID > 0; --layerID)

{

Layer &hiddenLayer = curr\_layers[layerID];

Layer &toLayer = curr\_layers[layerID + 1];

for (unsigned n = 0; n < hiddenLayer.size(); ++n)

{

hiddenLayer[n].derivativesHidden(toLayer);

}

}

// update Link weights

for (unsigned layerID = curr\_layers.size() - 1; layerID > 0; --layerID)

{

Layer &layer = curr\_layers[layerID];

Layer &fromLayer = curr\_layers[layerID - 1];

for (unsigned n = 0; n < layer.size() - 1; ++n)

{

layer[n].updateInWeights(fromLayer);

}

}

}

// Feed forward

void MLP::feedForward(const vector<double> &inVals)

{

assert(inVals.size() == curr\_layers[0].size() - 1);

// Assign (latch) the in values into the in Nodes

for (unsigned i = 0; i < inVals.size(); ++i)

{

curr\_layers[0][i].setOutVal(inVals[i]);

}

// Foward propagate

for (unsigned layerID = 1; layerID < curr\_layers.size(); ++layerID)

{

Layer &fromLayer = curr\_layers[layerID - 1];

for (unsigned n = 0; n < curr\_layers[layerID].size() - 1; ++n)

{

curr\_layers[layerID][n].feedForward(fromLayer);

}

}

}

MLP::MLP(const vector<unsigned> &topo)

{

unsigned totalLayers = topo.size();

for (unsigned layerID = 0; layerID < totalLayers; ++layerID)

{

curr\_layers.push\_back(Layer());

unsigned totalOuts = layerID == topo.size() - 1 ? 0 : topo[layerID + 1];

// Push the nodes into correspond layer

for (unsigned NodeNum = 0; NodeNum <= topo[layerID];++NodeNum)

{

curr\_layers.back().push\_back(Node(totalOuts, NodeNum));

}

curr\_layers.back().back().setOutVal(1.0); // bias node y0 always = 1

}

}

***Node.cpp***

#include "pch.h"

double Node::learnRate = 0.35; // MLP learning rate

double Node::a = 0.5; //WeightAduj momentum multiplier

// Update the weights with learning rate and trend of old weights.

void Node::updateInWeights(Layer &fromLayer)

{

for (unsigned n = 0; n < fromLayer.size(); ++n)

{

Node &Node = fromLayer[n];

double oldWeightAduj = Node.curr\_outWeights[curr\_nodeID].WeightAduj;

// dW = learnRate\*error\*d(phi)\*y

// y = phi[sum(w\*y)] = "curr\_Derivative"

// error\*d(phi) = "Node.getOutVal()"

double newWeightAduj = learnRate \* Node.getOutVal() \* curr\_Derivative + a \* oldWeightAduj;

Node.curr\_outWeights[curr\_nodeID].WeightAduj = newWeightAduj;

Node.curr\_outWeights[curr\_nodeID].weight += newWeightAduj; //W\_new = W\_old + W\_adjust;

}

}

//Sum of errors at nodes we feed which is error in dW = learnRate\*error\*d(phi)\*y

double Node::sumDerivative(const Layer &toLayer) const

{

double sum = 0.0;

for (unsigned n = 0; n < toLayer.size() - 1; ++n)

{

// Sum of error which is e in equation

sum += curr\_outWeights[n].weight \* toLayer[n].curr\_Derivative;

}

return sum;

}

void Node::derivativesHidden(const Layer &toLayer)

{

double d = sumDerivative(toLayer);

// d(phi) = actiFuncDerivative(curr\_outVal)

curr\_Derivative = d \* Node::actiFuncDerivative(curr\_outVal);

}

void Node::derivativesOut(double targetVal)

{

double d = targetVal - curr\_outVal;

curr\_Derivative = d \* Node::actiFuncDerivative(curr\_outVal);

}

// Activation Function

double Node::actiFunc(double x)

{

return tanh(x);

}

// Derivative of Activation Function

double Node::actiFuncDerivative(double x)

{

//d(tanh)/dx ~= 1-x^2

return 1 - x \* x;

}

// Calculate the outputs of each nodes

void Node::feedForward(const Layer &fromLayer)

{

double sum = 0.0;

for (unsigned n = 0; n < fromLayer.size(); ++n)

{

// Sum all inputs\*weights = sum(w\*y)

sum += fromLayer[n].getOutVal()\*fromLayer[n].curr\_outWeights[curr\_nodeID].weight;

}

// y = phi[sum(w\*y)] = actiFunc(sum)

curr\_outVal = Node::actiFunc(sum);

}

Node::Node(unsigned totalOuts, unsigned nodeID)

{

for (unsigned c = 0; c < totalOuts; ++c)

{

curr\_outWeights.push\_back(Link());

// Initail weight

curr\_outWeights.back().weight = randomWeight();

}

curr\_nodeID = nodeID;

}