Main Report

Assignment 1 --- Neural Network

0340282 吳行方

A. Platform & Programming language

In the preliminary report, I would like to code this program in JavaScript. But I noticed that to code a neural network, floating-point calculation takes up a large part of calculation. So I change it to the C++, which has a better performance of floating-point calculation.

As for platform, I coded and tested it in Xcode 6.2 on Mac OS X 10.10.2.

B. General Design

It is really hard to code neural network for the first time because there are many algorithms and methods to choose from. Some of them are quite complex for their complex mathematical background.

In the preliminary report, I planned to design several classes. By communicating with each other, classes compose a well-organized neural network. But when I was about to code, I found it rather complex. So I redesigned the structure and figured out a simple but good one.

I only designed a single class, which named BPNeuralNetwork. It is rather like the NeuralNetwork class mentioned in the preliminary report. The neural cells and layers are directly coded in the procedures of the BPNeuralNetwork class.

C. Detailed Design

```
class BPNeuralNetwork
{
protected:
    static int sampleNumber;
    double
weight_Input_Hide[INPUTNODE][HIDENODE];
    double
weight_Hide_Output[HIDENODE][INPUTNODE];
    double threshold_Hide[HIDENODE];
    double threshold_Output[OUTPUTNODE];
public:
    void train(vector<vector<double>>
TrainData,vector<vector<double>> TrainResult);
    double* classify(vector<double>
ClassifyData,double
OutputLayerOutput[OUTPUTNODE]);
    BPNeuralNetwork();
    virtual ~BPNeuralNetwork();
};
```

BPNeuralNetwork class has some data structures to save the parameters of the neural network, such as weight_Input_Hide, threshold_Hide, and etc. weight_Hide_Output, for example, documents the weights between the hidden cell and the output cell. Threshold_Hide saves the hidden layer's neural cells' activating thresholds. When the propagation algorithm is applied, data in these data structures would be adjusted, which means a better response to input.

There are also some member functions provides to be called to utilize this class. When instantiated, the construction function will be called and the data structures mentioned above will be initialized. When the function named train is called, and a set of training data is passed to it, it will use the back-propagation algorithm to alter the weights of synapses and the thresholds of activation functions.

The following of this section is the detailed realization of back-propagation algorithm.

- Member function train:

```
void BPNeuralNetwork::train(vector<vector<double>>
TrainData,vector<vector<double>> TrainResult)
{
    double BPErr_output[OUTPUTNODE];
    double BPErr_hide[HIDENODE];
    for (int i=0; i<TrainData.size(); i++) {

        //do front propagation
        double HideNodeOutput[HIDENODE];
        for (int j=0; j<HIDENODE; j++) {
            double sum=0;
            for (int k=0; k<INPUTNODE; k++) {
                sum+=TrainData[i][k] * weight_Input_Hide[k][j];
            }
            HideNodeOutput[j]=sigmoid(sum);
        }

        double outputResult[OUTPUTNODE];
        for (int j=0; j<OUTPUTNODE; j++) {
            double sum=0;
            for (int k=0; k<HIDENODE; k++) {

sum+=HideNodeOutput[k]*weight_Hide_Output[HIDENODE][OUTPUTNODE];
            }
            outputResult[j]=sigmoid(sum);
        }

        //calculate the err before refreshing
        double error=0;
        for (int j=0; j<OUTPUTNODE; j++) {
            error+=pow((TrainResult[i][j]-outputResult[j]),2);
        }
        cout<<"err:"<<error<<endl;

        for (int j=0 ; j<OUTPUTNODE ; j++) {
            BPErr_output[j]=(TrainResult[i][j]-
outputResult[j])*outputResult[j]*(1-outputResult[j]);  //find the
err

            //refresh the weight between hidden layer and output
layer
            for (int k=0; k<HIDENODE; k++) {

weight_Hide_Output[k][j]+=LEARNING_RATE_WEIGHT_HIDEEN_OUTPUT*BPErr_o
```

```
utput[j]*HideNodeOutput[k];
}
        }

        for (int j=0; j<HIDENODE; j++) {
            BPErr_hide[j]=0;
            //back propagate the err to the hidden layer
            for (int k=0; k<OUTPUTNODE; k++) {

BPErr_hide[j]+=BPErr_output[k]*weight_Hide_Output[j][k];
            }
            BPErr_hide[j]*=(HideNodeOutput[j]*(1-
HideNodeOutput[j]));
            //refresh weights between input layer and hidden layer
            for (int k=0; k<INPUTNODE; k++) {

//cout<<"weight2:"<<"["<<k<<"]"<<weight_Input_Hide[k][j]<<"  ";

weight_Input_Hide[k][j]+=LEARNING_RATE_WEIGHT_INPUT_HIDDEN*BPErr_hid
e[j]*HideNodeOutput[j];

//cout<<"weight2,after:"<<"["<<k<<"]"<<weight_Input_Hide[k][j]<<endl
;
            }
        }
        //refresh nodes' threshold
        for (int j=0; j<OUTPUTNODE; j++) {

threshold_Output[j]+=LEARNING_RATE_NODE_OUTPUT*BPErr_output[j];
        }
        for (int j=0; j<HIDENODE; j++) {

threshold_Hide[j]+=LEARNING_RATE_NODE_HIDDEN*BPErr_hide[j];
        }
    }
}
```

The first thing to do for the function is using the training data set to derive output values. Then, calculate the error of the output values with the true values provided, and passes the error backwards to the hidden layer and calculates new weights of the output layer and hidden layer with the macro LEARNING_RATE and error parameter. The delta method is used in the refreshing stage. Then, it comes to the weight refreshing between input layer and the hidden layer. It is similar with the procedure mentioned above. First, calculate the errors of output of hidden layer cells. Distribute the errors to each input, and then calculate new weight values.

The refreshing of thresholds is simpler. Only a calculation which rely on the learning rate and error values is needed.

D. Test & Results

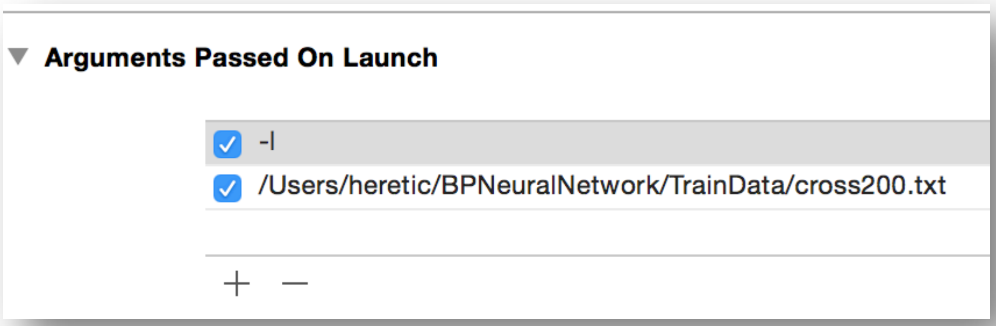
- Setting the parameters of the neural network:

```
#define INPUTNODE 2
#define HIDENODE  3
#define OUTPUTNODE 1

#define COUST_P 2

#define LEARNING_RATE_WEIGHT_INPUT_HIDDEN 0.9
#define LEARNING_RATE_WEIGHT_HIDEEN_OUTPUT 0.9
#define LEARNING_RATE_NODE_HIDDEN 0.9
#define LEARNING_RATE_NODE_OUTPUT 0.9
```

- Debugging the program with the arguments "-t" and the training data's URL.



- Documenting the output of the program(The following data is just a part of the output):

```
Successfully read
data from file!
200 piece(s) train
data!
err:0.656133
err:0.197367
err:0.113442
err:0.0687779
err:0.0616707
err:0.0262436
err:0.0273173
err:0.0366753
err:0.0284743
err:0.0291759
err:0.0152805
err:0.0258847
err:0.00970116
err:0.00732955
err:0.00730821
err:0.019178
err:0.0177405
err:0.0156265
err:0.00448648
err:0.0238983
err:0.0155424
err:0.00542113
err:0.00765365
err:0.0153072
err:0.0111428
err:0.00469829
err:0.00313865
err:0.00641036
err:0.004381
err:0.00368085
err:0.00273055
err:0.0100028
```

```
err:0.00916209
err:0.00197056
err:0.00180144
err:0.0042779
err:0.00171552
err:0.00315969
err:0.00761517
err:0.00468298
err:0.00323145
err:0.0145299
err:0.00110791
err:0.00176276
err:0.0123414
err:0.00135644
err:0.00184865
err:0.00266994
err:0.0122898
err:0.00754018
err:0.000955969
err:0.0072311
err:0.0094461
err:0.0030312
err:0.00596496
```

```
err:0.00755015
err:0.0068001
err:0.00105791
err:0.00519702
err:0.000715829
err:0.00726645
err:0.00565057
err:0.00319776
err:0.00974975
err:0.00160783
err:0.00685229
err:0.00104812
err:0.000953882
err:0.00420222
err:0.00104155
err:0.00115846
err:0.000747635
err:0.00495602
err:0.00589586
err:0.00429428
err:0.00164504
err:0.00145737
err:0.00120451
```

```
err:0.00748066
err:0.00243949
err:0.000460332
err:0.000550581
err:0.00521523
err:0.000801217
err:0.00765335
err:0.00108737
err:0.00068083
err:0.00138934
err:0.00198268
err:0.0080022
err:0.000852171
err:0.00472502
err:0.000446791
err:0.00603427
err:0.00706722
err:0.00804237
err:0.00171627
err:0.00288999
err:0.00204507
err:0.000975053
```

We find that the error is becoming less and less. We can conclude that the neural network learned from the training data. However, the error data become abnormal when the training data come to class 2:

```
err:0.946108
err:0.824514
err:0.93411
err:0.785306
err:0.876644
err:0.855191
err:0.901487
err:0.695058
err:0.818095
```

```
err:0.648267
err:0.541891
err:0.411216
err:0.283929
err:0.261404
err:0.341995
err:0.422996
err:0.592963
err:0.740373
```

```
err:0.858843
err:0.889346
err:0.915382
err:0.853066
err:0.948927
err:0.928827
err:0.958219
```

The error become less and less, but suddenly become more and more larger. I think that it may because the data is a little abnormal compared with the previous training. If we need to achieve a lower error value, larger training data amount is needed.

Appendix

Assignment 1 --- Neural Network

- Main.cpp

```
//
//  main.cpp
//  BPNeuralNetwork
//
//  Created by Holden Wu on 4/8/15.
//  MIT LICENCE
//

#include <iostream>
#include <vector>
#include <String>

#include "BPNeuralNetwork.h"

using namespace std;

int main(int argc, const char * argv[]) {
    string path;
    if (!strcmp(argv[1],"-l")) {
        if (argv[2]) {
            path = argv[2];
        }else{
            path = argv[0];
            path+="/traindata.txt";
        }
        FILE *stream;
        if ((stream=::fopen(path.c_str(),
"r"))==NULL) {
            cout<<"Fail to read train
data!"<<endl;
            exit(1);
        }
        vector<vector<double>> TrainData;
        vector<vector<double>> TrainResult;
```

```
        vector<double> input,output;
        double finput,foutput;
        int count=0;
        while (!feof(stream)) {
            for (int i=0; i<INPUTNODE; i++) {
                fscanf(stream,"%lf",&finput);
                input.push_back(finput);
            }
            TrainData.push_back(input);
            for (int i=0; i<OUTPUTNODE; i++) {
                fscanf(stream,"%lf",&foutput);
                output.push_back(foutput);
            }
            TrainResult.push_back(output);
            input.clear();
            output.clear();
            count++;
        }

//          for(int i=0;i<TrainData.size();i++){
//              cout<<"(";
//              for (int b=0;
b<TrainData[i].size(); b++) {
//                  cout<<TrainData[i][b]<<",";
//              }
//              cout<<")"<<endl;
//          }


if(TrainData.size()!=TrainResult.size()){
            cout<<"Data Error!"<<endl;
            exit(1);
        }

        cout<<"Successfully read data from
file!"<<count<<" piece(s) train data!"<<endl;
```

```
        vector<double> classifyData;
        double a=1.119,b=-1.388;
        classifyData.push_back(a);
        classifyData.push_back(b);
        double outputResult[OUTPUTNODE];
        BPNeuralNetwork network;
        network.train(TrainData,TrainResult);

        network.classify(classifyData,
outputResult);
    }

    return 0;
}
```

- BPNeuralNetwork.h

```
//
//  BPNeuralNetwork.h
//  BPNeuralNetwork
//
//  Created by Holden Wu on 4/8/15.
//  MIT LICENCE
//

#ifndef BPNeuralNetwork_BPNeuralNetwork_h
#define BPNeuralNetwork_BPNeuralNetwork_h

#define INPUTNODE 2
#define HIDENODE  3
#define OUTPUTNODE 1

#define COUST_P 2
```

```
#define LEARNING_RATE_WEIGHT_INPUT_HIDDEN 0.9
#define LEARNING_RATE_WEIGHT_HIDEEN_OUTPUT 0.9
#define LEARNING_RATE_NODE_HIDDEN 0.9
#define LEARNING_RATE_NODE_OUTPUT 0.9

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

class BPNeuralNetwork
{
protected:
    static int sampleNumber;
    double
weight_Input_Hide[INPUTNODE][HIDENODE];
    double
weight_Hide_Output[HIDENODE][INPUTNODE];
    double threshold_Hide[HIDENODE];
    double threshold_Output[OUTPUTNODE];
public:
    void train(vector<vector<double>>
TrainData,vector<vector<double>> TrainResult);
    double* classify(vector<double>
ClassifyData,double
OutputLayerOutput[OUTPUTNODE]);
    BPNeuralNetwork();
    virtual ~BPNeuralNetwork();
};


#endif
```

- BPNeuralNetwork.cpp

```
//
//  BPNeuralNetwork.cpp
//  BPNeuralNetwork
//
//  Created by Holden Wu on 4/8/15.
//  MIT LICENCE
//

#include "BPNeuralNetwork.h"

void random_Init(double array[],int n){
    for (int i=0; i<n; i++) {
        array[i]=2*((double)rand()/RAND_MAX)-1;
    }
}

double sigmoid(double input){
    return 1.0/(1.0+exp(-input/COUST_P))+1;
}

BPNeuralNetwork::BPNeuralNetwork()
{
    srand((unsigned)time(NULL));
    random_Init((double*)weight_Input_Hide,
INPUTNODE*HIDENODE);

random_Init((double*)weight_Hide_Output,HIDENODE*OUTPUTNODE);
    random_Init(threshold_Hide, HIDENODE);
    random_Init(threshold_Output, OUTPUTNODE);
}

BPNeuralNetwork::~BPNeuralNetwork()
{

}

double* BPNeuralNetwork::classify(vector<double>
ClassifyData,double OutputLayerOutput[OUTPUTNODE]){
    double HideNodeOutput[HIDENODE];
    for (int i=0; i<HIDENODE; i++) {
        double sum=0;
        for (int j=0; j<INPUTNODE; j++) {
            sum+=ClassifyData[j] * weight_Input_Hide[j][i];
        }
        HideNodeOutput[i]=sigmoid(sum);
```

```
    }

    for (int i=0; i<OUTPUTNODE; i++) {
        double sum=0;
        for (int j=0; j<HIDENODE; j++) {

sum+=HideNodeOutput[j]*weight_Hide_Output[HIDENODE][OUTPUTNODE
];
        }
        OutputLayerOutput[i]=sigmoid(sum);
    }

    for (int i=0; i<OUTPUTNODE; i++) {
        cout<<"Output Result is:"<<OutputLayerOutput[i]<<endl;
    }

    return OutputLayerOutput;
}

void BPNeuralNetwork::train(vector<vector<double>>
TrainData,vector<vector<double>> TrainResult)
{
    double BPErr_output[OUTPUTNODE];
    double BPErr_hide[HIDENODE];
    for (int i=0; i<TrainData.size(); i++) {

        //do front propagation
        double HideNodeOutput[HIDENODE];
        for (int j=0; j<HIDENODE; j++) {
            double sum=0;
            for (int k=0; k<INPUTNODE; k++) {
                sum+=TrainData[i][k] *
weight_Input_Hide[k][j];
            }
            HideNodeOutput[j]=sigmoid(sum);
        }

        double outputResult[OUTPUTNODE];
        for (int j=0; j<OUTPUTNODE; j++) {
            double sum=0;
            for (int k=0; k<HIDENODE; k++) {

sum+=HideNodeOutput[k]*weight_Hide_Output[HIDENODE][OUTPUTNODE
];
            }
            outputResult[j]=sigmoid(sum);
        }

        //calculate the err before refreshing
```

```
        double error=0;
        for (int j=0; j<OUTPUTNODE; j++) {
            error+=pow((TrainResult[i][j]-outputResult[j]),2);
        }
        cout<<"err:"<<error<<endl;

        for (int j=0 ; j<OUTPUTNODE ; j++) {
            BPErr_output[j]=(TrainResult[i][j]-
outputResult[j])*outputResult[j]*(1-outputResult[j]);  //find
the err

            //refresh the weight between hidden layer and
output layer
            for (int k=0; k<HIDENODE; k++) {

//cout<<"weight1:"<<"["<<k<<"]"<<weight_Hide_Output[k][j]<<"
";

weight_Hide_Output[k][j]+=LEARNING_RATE_WEIGHT_HIDEEN_OUTPUT*B
PErr_output[j]*HideNodeOutput[k];

//cout<<"weight1,after:"<<"["<<k<<"]"<<weight_Hide_Output[k][j
]<<endl;
            }
        }

        for (int j=0; j<HIDENODE; j++) {
            BPErr_hide[j]=0;
            //back propagate the err to the hidden layer
            for (int k=0; k<OUTPUTNODE; k++) {

BPErr_hide[j]+=BPErr_output[k]*weight_Hide_Output[j][k];
            }
            BPErr_hide[j]*=(HideNodeOutput[j]*(1-
HideNodeOutput[j]));
            //refresh weights between input layer and hidden
layer
            for (int k=0; k<INPUTNODE; k++) {

//cout<<"weight2:"<<"["<<k<<"]"<<weight_Input_Hide[k][j]<<"
";

weight_Input_Hide[k][j]+=LEARNING_RATE_WEIGHT_INPUT_HIDDEN*BPE
rr_hide[j]*HideNodeOutput[j];

//cout<<"weight2,after:"<<"["<<k<<"]"<<weight_Input_Hide[k][j]
<<endl;
            }
        }
```

```
        //refresh nodes' threshold
        for (int j=0; j<OUTPUTNODE; j++) {

threshold_Output[j]+=LEARNING_RATE_NODE_OUTPUT*BPErr_output[j]
;
        }
        for (int j=0; j<HIDENODE; j++) {

threshold_Hide[j]+=LEARNING_RATE_NODE_HIDDEN*BPErr_hide[j];
        }
    }
}
```