编程作业-c语言实现BP神经网络

**开发环境**：visual studio 2015

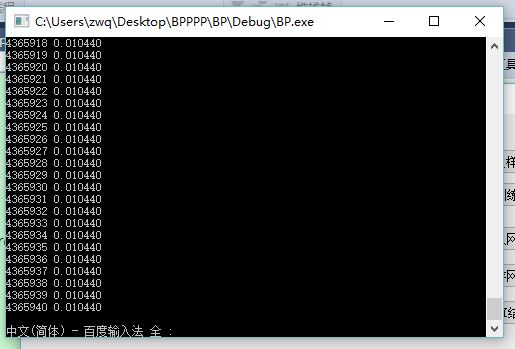
**使用说明**：

1、点击

**目标函数**：

**训练结果**：

共耗时一夜，训练迭代了400多万次，平均误差



**调试心得**：

1. 激活函数一开始全都用的是sigmoid函数，后来网上有用y = x作为输出层激活函数的，确实效果要好一些，可能是因为没有平坦区；
2. 但是如果更改输出层激活函数的话学习率应该减小几个数量级，因为sigmoid函数的输出只能在0~1之间而改成y = x的话会超过1，从而反向传播调整权值时很容易震荡；
3. 同理，输入样本个数也关乎到学习率的调整，样本数量级增大时，学习率应该减小数量级；
4. 第3条经验仅适用于所有样本都算出一轮权重调整值求和再调整权重的方式，还有一种方法是每个样本算出权重调整值后立刻就调整当前权重，这种情况下学习率就需要扩大样本数量倍。而且这种方式效果还好一些，但是最后没有用因为和公式写的不一样啊！
5. 关于改进方法，试了增加动量项好像没有明显的效果就去掉了。不过在自适应调整学习率上新提出了一点小小的改进，就是如果训练一次发现误差变大或者误差下降的太慢就随机调整学习率，调整范围是使当前学习率变为设定学习率的50%~150%。略有效果。不过再改进也抵不过随机初始化权重时赶上好的初始值来得明显；
6. 本品为纯手工制品。虽然有所参考，但BP算法真的是照着书上的公式一点一点攒出来的以至于书上公式写错了多了个负号导致调了好几天（其实本应立刻能发现的因为误差会很快增大到无穷，但是因为隐层的激活函数一开始也漏写了个负号，结果居然凑合能用，就是误差一直降不到0.05以下所以一直调不出来）（回去看了下老师的课件发现写的是对的，果然孙老师良心出品）

控诉一下这本书！

附录：核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BPANN.h\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma once

#include <math.h>

#include <time.h>

#include <conio.h>

using namespace std;

#define INPUT\_DIM 2 //输入变量个数

#define OUTPUT\_DIM 1 //输出变量个数

#define HIDE\_DIM 45 //隐含节点个数

#define SAMPLE\_NUM 1000 //训练样本总数

struct structNode

{

double weight[50]; //当前节点对每一个输入的权重

double dweight[50]; //每次训练后每个权重需要改变的量

double delta; //计算每层节点权重改变量时用到的系数

double sum\_dw; //计算隐含层节点权重改变量时用到的系数

double Output\_x;//当前节点输入之和

double Output\_y; //当前节点输出

double Output\_dy; //当前节点输出的导数

};

typedef struct structNode strctNode; //节点结构体

struct structData

{

double TrainData[SAMPLE\_NUM]; //数据：原始的→归一化后的

double TrainData\_Max; //数据最大值，归一化用

double TrainData\_Min; //数据最小值，归一化用

double TrainData\_MaxMin; //数据最大值和最小值之差，归一化用

};

typedef struct structData strctData; //数据（输入或输出）结构体

class clsANN

{

public:

double error\_limit; //误差限，误差小于它则停止训练

double error; //当前误差

double error\_max; //当前最大误差

double rate; //学习率

bool is\_trained; //仿真验证的时候用的，是不是已经归一化过了

strctNode NodeOutput[OUTPUT\_DIM]; //输出节点

strctNode NodeHide[HIDE\_DIM]; //隐含层节点

//输入节点直接就是输入

strctData TrainDataIn[INPUT\_DIM]; //储存输入训练数据

strctData TrainDataOut[OUTPUT\_DIM]; //储存输出训练数据

double ResultOut[OUTPUT\_DIM][SAMPLE\_NUM]; //储存神经网络计算出的结果

void do\_train(double rate, double error\_limit, int max\_iteration); //训练函数

void calc\_result(); //仿真验证函数

double calc\_test(double x[]); //测试函数

void init\_weight(); //使用随机数初始化权重

void findmaxmin(); //找最大最小值

void input\_normalization(int num\_sample); //输入数据归一化

void output\_normalization(int num\_sample); //输出数据归一化

void output\_anti\_normalization(int num\_sample); //神经网络输出反归一化

void calc\_output(int group\_sample); //计算当前输出

void calc\_dweight(int group\_sample); //计算当前权重需要改变的量

void adjust\_weight(); //调整权重

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BPANN.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "stdafx.h"

#include "BPANN.h"

void clsANN::init\_weight() //初始化权重为-1~1之间的随机数，初始化权重变化量都是0

{

int i, j;

for (i = 0; i < OUTPUT\_DIM; i++)

{

for (j = 0; j < HIDE\_DIM; j++)

{

NodeOutput[i].dweight[j] = 0;

NodeOutput[i].weight[j] = (rand() \* 2.0 / RAND\_MAX) - 1; //rand()生成0~RAND\_MAX之间的随机数

}

}

for (i = 0; i < HIDE\_DIM; i++)

{

for (j = 0; j < INPUT\_DIM; j++)

{

NodeHide[i].dweight[j] = 0;

NodeHide[i].weight[j] = (rand() \* 2.0 / RAND\_MAX) - 1;//rand()生成0~RAND\_MAX之间的随机数

NodeHide[i].sum\_dw = 0;

}

}

}

void clsANN::findmaxmin() //找输入输出的最大值和最小值，计算最大值和最小值之差

{

int i, j;

for (i = 0; i < INPUT\_DIM; i++)

{

TrainDataIn[i].TrainData\_Max = TrainDataIn[i].TrainData[0];

TrainDataIn[i].TrainData\_Min = TrainDataIn[i].TrainData[0];

for (j = 0; j < SAMPLE\_NUM; j++)

{

if (TrainDataIn[i].TrainData[j] > TrainDataIn[i].TrainData\_Max) TrainDataIn[i].TrainData\_Max = TrainDataIn[i].TrainData[j];

if (TrainDataIn[i].TrainData[j] < TrainDataIn[i].TrainData\_Min) TrainDataIn[i].TrainData\_Min = TrainDataIn[i].TrainData[j];

}

TrainDataIn[i].TrainData\_MaxMin = TrainDataIn[i].TrainData\_Max - TrainDataIn[i].TrainData\_Min + 1;

}

for (i = 0; i < OUTPUT\_DIM; i++)

{

TrainDataOut[i].TrainData\_Max = TrainDataOut[i].TrainData[0];

TrainDataOut[i].TrainData\_Min = TrainDataOut[i].TrainData[0];

for (j = 0; j < SAMPLE\_NUM; j++)

{

if (TrainDataOut[i].TrainData[j] > TrainDataOut[i].TrainData\_Max) TrainDataOut[i].TrainData\_Max = TrainDataOut[i].TrainData[j];

if (TrainDataOut[i].TrainData[j] < TrainDataOut[i].TrainData\_Min) TrainDataOut[i].TrainData\_Min = TrainDataOut[i].TrainData[j];

}

TrainDataOut[i].TrainData\_MaxMin = TrainDataOut[i].TrainData\_Max - TrainDataOut[i].TrainData\_Min + 1;

}

}

void clsANN::input\_normalization(int num\_sample) //归一化输入

{

int i, j;

for (i = 0; i < INPUT\_DIM; i++)

{

for (j = 0; j < num\_sample; j++)

{

TrainDataIn[i].TrainData[j] = (TrainDataIn[i].TrainData[j] - TrainDataIn[i].TrainData\_Min + 1) / TrainDataIn[i].TrainData\_MaxMin;

}

}

}

void clsANN::output\_normalization(int num\_sample) //归一化输出

{

int i, j;

for (i = 0; i < OUTPUT\_DIM; i++)

{

for (j = 0; j < num\_sample; j++)

{

TrainDataOut[i].TrainData[j] = (TrainDataOut[i].TrainData[j] - TrainDataOut[i].TrainData\_Min + 1) / TrainDataOut[i].TrainData\_MaxMin;

}

}

}

void clsANN::output\_anti\_normalization(int num\_sample) //网络输出去归一化

{

int i;

for (i = 0; i < OUTPUT\_DIM; i++)

{

ResultOut[i][num\_sample] = NodeOutput[i].Output\_y \* TrainDataOut[i].TrainData\_MaxMin + TrainDataOut[i].TrainData\_Min - 1;

}

}

void clsANN::calc\_output(int group\_sample) //计算网络输出

{

int i, j;

for (i = 0; i < HIDE\_DIM; i++) //计算隐含层节点输出，隐含层激活函数使用sigmoid函数

{

NodeHide[i].Output\_x = 0;

for (j = 0; j < INPUT\_DIM; j++)

{

NodeHide[i].Output\_x += NodeHide[i].weight[j] \* TrainDataIn[j].TrainData[group\_sample];

}

NodeHide[i].Output\_y = 1.0 / (1.0 + exp((-1.0) \* NodeHide[i].Output\_x));

NodeHide[i].Output\_dy = 1.0 \* NodeHide[i].Output\_y \* (1.0 - NodeHide[i].Output\_y);

}

for (i = 0; i < OUTPUT\_DIM; i++) //计算输出层节点输出，输出层激活函数直接使用y = x

{

NodeOutput[i].Output\_x = 0;

for (j = 0; j < HIDE\_DIM; j++)

{

NodeOutput[i].Output\_x += NodeOutput[i].weight[j] \* NodeHide[j].Output\_y;

}

NodeOutput[i].Output\_y = NodeOutput[i].Output\_x; //1.0 / (1.0 + exp((-1.0) \* NodeOutput[i].Output\_x)); //

NodeOutput[i].Output\_dy = 1.0 \* NodeOutput[i].Output\_y \* (1.0 - NodeOutput[i].Output\_y);

error += fabs((TrainDataOut[i].TrainData[group\_sample] - NodeOutput[i].Output\_y) / TrainDataOut[i].TrainData[group\_sample]);

//if (error > error\_max) error\_max = error;

}

}

void clsANN::calc\_dweight(int group\_sample) //反向传播法计算权重调整值

{

int i, j;

double delta;

for (i = 0; i < OUTPUT\_DIM; i++) //按照公式计算输出层节点的权重调整值并累计每个隐层节点输出的delta\*weight之和用于计算隐层节点的权重调整值

{

delta = (TrainDataOut[i].TrainData[group\_sample] - NodeOutput[i].Output\_y);//\* NodeOutput[i].Output\_dy;

for (j = 0; j <HIDE\_DIM; j++)

{

NodeHide[j].sum\_dw += delta \* NodeOutput[i].weight[j];

NodeOutput[i].dweight[j] += rate \* delta \* NodeHide[j].Output\_y;

}

}

for (i = 0; i < HIDE\_DIM; i++) //按照公式计算隐层节点的权重调整值

{

delta = NodeHide[i].sum\_dw \* NodeHide[i].Output\_dy;

NodeHide[i].sum\_dw = 0;

for (j = 0; j <INPUT\_DIM; j++)

{

NodeHide[i].dweight[j] += rate \* delta \* TrainDataIn[j].TrainData[group\_sample];

}

}

}

void clsANN::adjust\_weight() //调整权重

{

int i, j;

for (i = 0; i < OUTPUT\_DIM; i++)

{

for (j = 0; j < HIDE\_DIM; j++)

{

NodeOutput[i].weight[j] += NodeOutput[i].dweight[j];

NodeOutput[i].dweight[j] = 0;

}

}

for (i = 0; i < HIDE\_DIM; i++)

{

for (j = 0; j < INPUT\_DIM; j++)

{

NodeHide[i].weight[j] += NodeHide[i].dweight[j];

NodeHide[i].dweight[j] = 0;

}

}

}

void clsANN::do\_train(double r, double e, int max\_iteration) //训练函数

{

\_cprintf("train\r\n");

double error\_pre = 0;

int i, j;

srand((unsigned)time(NULL));

rate = r;

error\_limit = e;

init\_weight(); //初始化权重

findmaxmin(); //找最大最小值

input\_normalization(SAMPLE\_NUM); //归一化输入

output\_normalization(SAMPLE\_NUM); //归一化输出

for (i = 0; 0 < max\_iteration; i++) //正常来说应该设置最大迭代次数，但是因为误差一直达不到要求所以最大迭代次数设成了无穷

{

error = 0; //正常来说应该判断误差最大值，但用误差最大值训练了一晚上也没降到百分之五一下最后只好用平均值

for (j = 0; j < SAMPLE\_NUM; j++) //每一组样本都计算输出，使用BP算法计算出误差调整值进行累计

{

calc\_output(j);

calc\_dweight(j);

}

error = error / SAMPLE\_NUM; //误差平均值

if (error <= error\_limit) break; //平均值小于百分之五训练停止

if (error > 1) //自己脑补了一个调整学习率的改进方法

{

if (error > error\_pre || error\_pre - error < 0.005) //如果训练完后误差反而变大了或者误差下降的太慢则调整学习率

rate = r \* (rand() \* 1.0 / RAND\_MAX + 0.5); //随机生成一个范围在1~1.5的因子乘以设定的学习率，即随机的增大或者减小学习率

}

else rate = r; //如果误差已经降到1以下就不再折腾，安心训练

error\_pre = error;

\_cprintf("%d %lf\r\n", i, error);

adjust\_weight(); //调整一波权重

}

is\_trained = true; //训练过了，标志置1

}

void clsANN::calc\_result() //将训练样本带回到网络中计算结果进行验证

{

\_cprintf("simulate\r\n");

if (is\_trained == false)

{

findmaxmin();

input\_normalization(SAMPLE\_NUM);

}

for (int j = 0; j < SAMPLE\_NUM; j++)

{

calc\_output(j);

output\_anti\_normalization(j);

if (is\_trained == false) \_cprintf("real:%lf result:%lf\r\n", TrainDataOut[0].TrainData[j], ResultOut[0][j]);

else \_cprintf("real:%lf result:%lf\r\n", TrainDataOut[0].TrainData[j] \* TrainDataOut[0].TrainData\_MaxMin + TrainDataOut[0].TrainData\_Min - 1, ResultOut[0][j]);

}

}

double clsANN::calc\_test(double x[]) //输入两个数，进行验证

{

double y;

for (int i = 0; i < INPUT\_DIM; i++)

{

x[i] = (x[i] - TrainDataIn[i].TrainData\_Min + 1) / (TrainDataIn[i].TrainData\_Max - TrainDataIn[i].TrainData\_Min + 1);

}

for (int i = 0; i < HIDE\_DIM; i++)

{

NodeHide[i].Output\_x = 0;

for (int j = 0; j < INPUT\_DIM; j++)

{

NodeHide[i].Output\_x += NodeHide[i].weight[j] \* x[j];

}

NodeHide[i].Output\_y = 1.0 / (1.0 + exp((-1.0) \* NodeHide[i].Output\_x));

NodeHide[i].Output\_dy = 1.0 \* NodeHide[i].Output\_y \* (1.0 - NodeHide[i].Output\_y);

}

for (int i = 0; i < OUTPUT\_DIM; i++)

{

NodeOutput[i].Output\_x = 0;

for (int j = 0; j < HIDE\_DIM; j++)

{

NodeOutput[i].Output\_x += NodeOutput[i].weight[j] \* NodeHide[j].Output\_y;

}

NodeOutput[i].Output\_y = NodeOutput[i].Output\_x; //1.0 / (1.0 + exp((-1.0) \* NodeOutput[i].Output\_x));

}

y = NodeOutput[0].Output\_y \* (TrainDataOut[0].TrainData\_Max - TrainDataOut[0].TrainData\_Min + 1) + TrainDataOut[0].TrainData\_Min - 1;

return y;

}