BPNN CPP 实现

目　　录

第一章 引言

随着近年来计算能力的提高，神经网络异常火爆，在许多方面都有应用，如语言识别、图像识别与理解、计算机视觉、智能机器人故检测等，而 Mnist 手写字体识别是其经典入门实例，通过此次实验希望可以更深刻的理解神经网络，通过实验中的公式推导继而自己编程实现，来体会科研过程。

第二章 实验内容

2.1 问题描述

神经网络算法的实现，并用手写数字进行实验。

2.2 数据集介绍

经典手写字体数据集 Mnist，其中训练集 60000 张，测试集 10000 张，png 格式，每张像素大小 20X20。

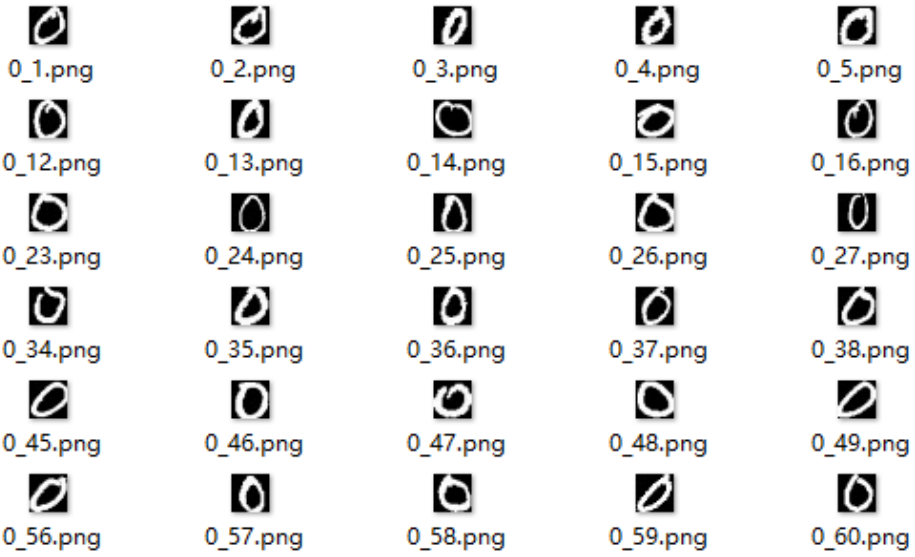

图 1 数据集剪影

第三章　具体实验

3.1 算法原理理解

3.1.1 基本结构

神经网络层与层之间是映射的过程，如 400 个输入 10 个输出，即将特征（400）映射到类别（10）的过程，为了映射准确，神经网络需知道足够信息，增加隐层结点数，增加联结数。

训练过程即神经网络学习从 400 到 10 如何映射的过程，是拟合（非线性）的过程。为了达到拟合，这里用到常用的一种最优化方法-梯度下降法。

当拟合度到达一定精度，满足要求，即可停止训练，这时得到的结果就是一堆参数，即神经网络的结构信息。神经网络的使用即加载参数，前向传递，进行回归或预测的过程。

3.1.2 调参

神经网络很重要的工作是调参，如

1. 每层神经元的个数：这里按经验公式进行尝试，又自己进行了尝试

$$m=\sqrt{n+l}+\alpha$$
$$m=\log_2 n$$
$$m=\sqrt{nl}$$

m：隐含层节点数
n：输入层节点数
l：输出层节点数
α：1--10 之间的常数。

图 2 经验公式

2. 如何初始化 Weights 和 biases：这里随机初始化权值，范围[0,0.5]，之后权值归一化，除输出层外每层都有一个偏置。
3. loss 函数选择哪一个：这里选用平方误差和
4. 选择何种 Regularization？L1,L2：这里未采用正则化
5. 激励函数如何选择：这里选用 sigmoid
6. 训练集多大比较合适：这里将 60000 分 60 批，即每个训练集 1000 个样本
7. 学习率多少合适：这里初始学习率 rate 为 0.1，随迭代次数 epoch 增加，

按照公式 rate/(1+d*epoch)，学习率逐渐下降。

8. 何时停止 Epoch 训练：这里设定训练次数，到达次数即停止训练

3.3 项目结构

这里使用 C++编写，项目结构及简单的主文件，神经网络头文件和实现文件。

3.4 代码设计

3.4.1 数据结构设计

3.4.1.1 节点结构

```
//node type
//include input weight,net value and output value
struct Perceptron {
    int inputWeightNum;//该节点的输入个数
    double* inputWeights;//输入权值数组
    double netValue,output,delta;//结点值，节点输出值，该节点之后网络的损失值之和
    Perceptron() {

    }
    Perceptron(int n)
    {
        double x,sum=0;
        inputWeightNum = n;
        inputWeights = new double[n];
        for (int i = 0; i < n; i++) {
            x = (double)rand() / (double)RAND_MAX;//随机初始化，范围在[0,0.5]
            inputWeights[i] = (x);
            sum += x;
        }
        //权值归一化
        for (int i = 0; i < n; i++) {
            inputWeights[i] /= sum;
        }
```

```
    }
    ~Perceptron(){

    }
};
```

3.4.1.2 层结构

```
//layer type
//include: a group of node
struct Layer {
    int perceptronNum; //该层节点个数
    Perceptron *perceptrons; //结点数组
    Layer() {

    }
    //初始化层 参数：结点数，前一层节点数，是否加偏置
    Layer(int n,int perN,bool partialNum)
    {
        perceptronNum = n;
        srand((unsigned)time(NULL));
        perceptrons = new Perceptron[n+1];
        int i;
        //每个节点的具体信息
        for (i = 0; i < n; i++) {
            Perceptron perceptron(perN);
            perceptrons[i] = perceptron;
        }
        //加偏置
        if (partialNum) {
            perceptronNum++;
            Perceptron perceptron(0);
            perceptron.netValue = 1;
            perceptron.output = 1;
            perceptrons[i] = perceptron;
```

```
        }
    }
    ~Layer()
    {

    }
};
```

3.4.1.3 样本结构

```
//a sample data
struct Sample{
    int featureNum; //特征向量大小
    double* feature;//特征向量
    double* label;//标签向量
    Sample() {

    }
    //初始化样本 参数：特征向量大小
    Sample(int n) {
        featureNum = n;
        feature = new double[featureNum];
        label = new double[featureNum];
    }
    //重载赋值运算符
    struct Sample& operator=(const struct Sample& s) {
        this->featureNum = s.featureNum;
        for (int i = 0; i < featureNum; i++) {
            this->feature[i] = s.feature[i];
            this->label[i] = s.label[i];
        }
        return *this;
    }
};
```

3.4.1.4 数据集结构

```
struct Data {
    int sampleNum; //数据集中样本个数
    struct Sample *sample;//样本集
    struct Data() {
        sampleNum = 0;
    }
    //初始化数据集 参数：样本数，特征数
    struct Data(int sampleNum,int featureNum) {
        this->sampleNum = sampleNum;
        this->sample = new struct Sample[sampleNum];
        for (int i = 0; i < sampleNum; ++i) {
            this->sample[i].feature = new double[featureNum];
            this->sample[i].label = new double[featureNum];
        }
    }
};
```

3.4.1.5 神经网络结构

```
//ANN type
//include: a group of layer
class ANN
{
public:

    // 结构相关
    ANN();
    //新建ANN，设置参数(输入层数，隐层数，输出层数，以及各层的结点个数)
    ANN(vector<int>& layer);
    ~ANN();
    void setParameter(double learningRate, int step,double e) { //设定参数
        this->learningRate = learningRate;
        this->step = step;
```

```
    this->e = e;
}
int getNumLayer() {
    return this->hiddenLayerNum + 2;
}
struct Layer* getLayers() { return layers; }//获取层参数信息
void setLearningRate(double learningRate) {
    this->learningRate = learningRate;
}

// 前向传播
double activeFunction(double netValue, int func); // 求经过激活函数的结果
double weightedSum(int layerNum,int perceptronNum); // 加权和
void forward(); //前向传播
double loss(int func);
double square_error();//计算对应节点均方误差
double getSquareError() { //获取均方误差
    return this->curSquareError;
}

// 后向传播
void backward(); //反向传播
double activeFunctionD(double netValue, int func); // 求经过激活函数的导数
double Loss_functionD(int func,double target,double out);//损失函数的导数
void updateWeights(int layerNum); //更新权值
double computeA(int layerNum, int perceptronNum);//计算此节点为总误差的贡献率

void train();

// 预测
double* prediction(struct Sample& sample);
int judgeClassification(double* v);

// 数据
void addData(int num);//载入数据
```

```
    void addTestData(int num);//载入测试数据

    // 参数存储 、 加载
    bool saveANN(string fileName);
    bool loadANN(string fileName);

    // 测试
    double test();
private:
    struct Layer* layers;//input hidden（每个隐层自动加一个偏置） output
    int hiddenLayerNum;// the number of hidden layer

    double learningRate;//learning rate
    int step;//迭代步数
    double curSquareError;//当前误差率
    double e;//精度

    struct Sample* sample;//sample
    struct Data* data;//data
    struct Data* testData;//data

};
```

3.4.2 关键代码分析

```
// 权值更新
void ANN::updateWeights(int layerNum){

    double t_o, o_n, n_w,t_w;
    //隐含层---->输出层的权值更新
    if(layerNum == this->hiddenLayerNum+1){
        int numP= this->layers[layerNum].perceptronNum,numW;
        for (int i = 0; i < numP; i++) {
            //总误差对输出层节点偏导 E_total / out01
            t_o = Loss_functionD(SE,this->sample->label[i],
```

```
                                this->layers[layerNum].perceptrons[i].output);

            //输出层输出值对求输出层节点值求偏导 out01 / net01
            o_n = activeFunctionD(this->layers[layerNum].perceptrons[i].output,
                                  SIGMOID);

            this->layers[layerNum].perceptrons[i].delta = t_o * o_n;
            //对outi的所有权值进行修正
            numW = this->layers[layerNum].perceptrons[i].inputWeightNum;
            for (int j = 0; j < numW; j++) {
                //节点值对权值求偏导 net01 / w
                n_w = this->layers[layerNum-1].perceptrons[j].output;
                //三者相乘 即整体误差对权值的偏导值 X
                t_w = this->layers[layerNum].perceptrons[i].delta * n_w;
                //更新权重值 w = w -a*X
                this->layers[layerNum].perceptrons[i].inputWeights[j]
                    -= this->learningRate * t_w;
            }
        }
    }
    //隐含层---->隐含层的权值更新 out(h1) net(h1) w1
    else if(layerNum >= 1 && layerNum < this->hiddenLayerNum + 1){
        int numP = this->layers[layerNum].perceptronNum, numW;
        for (int i = 0; i < numP; i++) {
            //总误差对输出层节点偏导 E_total / out(h1)
            //= E01 / out(h1) + E02 / out(h1) 误差相加
            computeA(layerNum, i);

            //对hidden i的所有权值进行修正
            numW = this->layers[layerNum].perceptrons[i].inputWeightNum;
            for (int j = 0; j < numW; j++) {
                //节点值对权值求偏导 net(h1) / w
                n_w = this->layers[layerNum-1].perceptrons[j].output;
                //三者相乘 即整体误差对权值的偏导值 X
                t_w = this->layers[layerNum].perceptrons[i].delta * n_w;
```

```
                //更新权重值 w = w -a*X
                this->layers[layerNum].perceptrons[i].inputWeights[j]
                        -= this->learningRate * t_w;
            }
        }
    }
    else {
        std::cerr << "update parameter wrong \n";
        return;
    }
}
```

```
// 计算某结点贡献的误差
double ANN::computeA(int layerNum, int perceptronNum)
{
    this->layers[layerNum].perceptrons[perceptronNum].delta = 0;
    int pointNum = this->layers[layerNum + 1].perceptronNum;
    //某节点之前的误差用delta表示，等于上一层每个结点的delta与对应权值乘积之和 再乘上本节点的损失
    for (int i = 0; i < pointNum; i++) {
        this->layers[layerNum].perceptrons[perceptronNum].delta
            += this->layers[layerNum + 1].perceptrons[i].delta
            * this->layers[layerNum + 1].perceptrons[i].inputWeights[perceptronNum];
    }
    this->layers[layerNum].perceptrons[perceptronNum].delta
        *= activeFunctionD(this->layers[layerNum].perceptrons[perceptronNum].output, SIGMOID);
    return 0;
}
```

第四章 测试数据及程序运行情况

4.1 训练过程

4.1.1 数据预处理

考虑到数字识别只看数字信息，而不必看颜色深浅，所以将图片二值化（只要灰度值大于 0 就设为 1），再向量化（20X20 矩阵化为 1X400 的向量）以满足神经网络输入层 400 个节点需要。为解决训练数据量过大不能一起输入训练的问题，这里将 60000 个样本分为 60 份，每份从 10 类数字中选取 100 个然后随机打

乱，为标识类别，这里在每个向量开头说明类别。

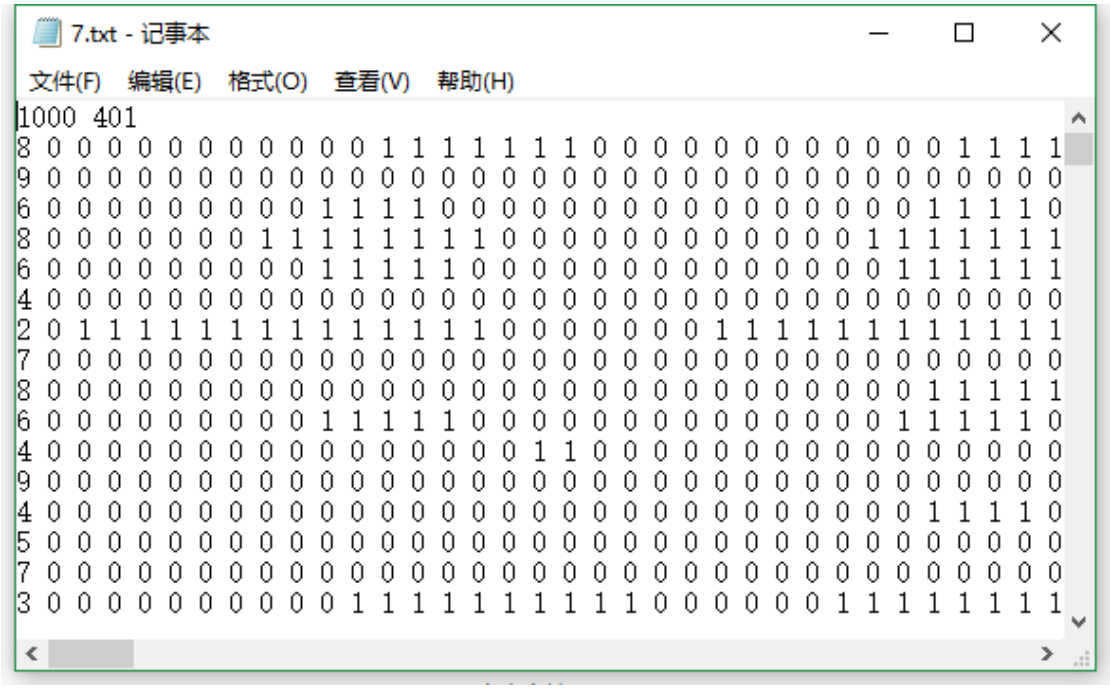

图 3 第 7 个样本集组织结构

如上图，60 份样本集中的第 7 个样本集，共包含 1000 个样本，每个样本开头标明类别，后面 400 个数字是图片向量化后的结果。

4.1.2 训练详情

```
int main()
{
    srand((unsigned)time(NULL));
    int layerArr[] = {400,100,10};
    vector<int> layerSS(layerArr, layerArr + sizeof(layerArr)/sizeof(int));
    //初始化神经网络
    ANN a(layerSS);

    //设置参数
    a.setParameter(0.1, 100, 0.01);
    double accuracy = 0;
    string fileName = "4_300_100";
```

```
    //若已存在网络，加载文件参数
    a.loadANN("./ann_"+ fileName +".parameter");
    int epoch = 100;

    //训练epoch代
    for(int i=0;i<epoch;i++){
        //一代 60 个数据集
        for (int j = 1; j < 61; j++) {
            //加载1个数据集
            a.addData(j);
            //训练
            a.train();
            //学习率随迭代次数增长而下降
            a.setLearningRate(0.1 / (1 + i * 0.001));
        }
        //存储网络参数
        a.saveANN("./ann_"+fileName+".parameter");
    }
    return 0;
}

// 训练过程即遍历数据集，对每个样本进行前向反向传播
void ANN::train()
{
    int sampleNum = (*this->data).sampleNum;
    int con = 0;
    this->sample = new struct Sample(this->data->sample->featureNum);
    for (int i =0; i<sampleNum;i++) {
        *this->sample = (*this->data).sample[i];
        //前向传播 反向传播交替
        forward();
        backward();
    }
}
```

训练结果就是一个参数文件

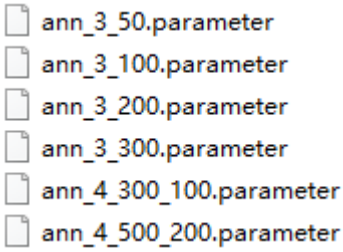

图 4 参数文件

4.2 测试结果

表 1 不同结构神经网络的测试结果（准确率）

结构	400 50 10	400 100 10	400 200 10	400 300 10
1	89.22%	89.46%	88.53%	86.84%
2	89.46%	92.63%	90.85%	89.08%
3	92.29%	94.22%	92.75%	92.12%
4	93.66%	95.07%	93.92%	93.32%
5	94.39%	95.43%	94.97%	93.13%
6	94.79%	95.52%	95.65%	93.85%
7	94.86%	95.66%	95.99%	94.67%
8	94.88%	95.70%	95.81%	95.47%
9	94.93%	95.72%	95.91%	95.35%
10	94.97%	95.79%	95.85%	95.79%
11	94.99%	95.84%	96.10%	95.62%
12	94.91%	95.87%	96.11%	95.56%
13	94.90%	95.78%	96.21%	95.81%
14	95.12%	95.77%	96.16%	95.81%
15	95.22%	95.79%	96.10%	96.18%
16	95.23%	95.83%	96.22%	96.29%
17	95.13%	95.91%	96.16%	95.81%
18	95.13%	95.88%	96.23%	96.38%
19	95.15%	95.87%	96.27%	96.13%
20	95.03%	96%	96.26%	96.56%
21	94.96%	96.10%	96.34%	96.38%
22	94.93%	96.04%	96.42%	96.61%

23	94.94%	96.07%	96.47%	96.50%
24	94.83%	96.13%	96.50%	96.65%
25	94.83%	96.15%	96.53%	96.54%
26	94.82%	96.13%	96.62%	96.53%
27		96.16%	96.60%	96.60%
28		96.19%	96.58%	96.18%
29		96.25%	96.61%	96.64%
30		96.30%	96.40%	96.44%
31		96.37%	96.53%	96.70%
32		96.33%	96.57%	96.58%
33		96.35%	96.55%	96.63%
34		96.38%	96.65%	96.08%
35		96.37%	96.68%	96.84%
36		96.44%	96.72%	96.84%
37		96.42%	96.74%	96.91%
38		96.42%	96.77%	96.94%
39		96.39%	96.73%	97.07%
40		96.41%	96.81%	96.96%
41		96.45%	96.75%	96.97%
42		96.47%	96.79%	96.88%
43		96.47%	96.85%	96.80%
44		96.44%	96.81%	96.85%
45		96.46%	96.86%	96.90%
46		96.49%	96.89%	96.90%
47		96.48%	96.86%	96.93%
48		96.46%	96.85%	96.95%
49		96.43%	96.83%	96.98%
50		96.42%	96.83%	97.02%
51		96.41%	96.89%	97.12%
52			96.85%	97.12%
53			96.88%	97.09%
54			96.89%	97.05%
55			96.95%	97.00%
56			96.98%	96.96%
57			97%	97%

58		97%	97%
59		97.03%	97.03%
60		97%	97%
61		97.03%	97.06%
62		97%	97%
63		97.02%	97.05%
64		97.02%	97.11%
65		97.01%	97.11%
66		97%	97%
67		97.01%	97.19%
68			97.15%
69			97.14%
70			97.17%
71			97.16%
72			97.12%
73			97.11%
74			97.16%
75			97.13%
76			97.12%

上述学习率均为 0.1，且随代数增加而减少。节点数 400 300 10 时，训练 67 轮时间约 2H，准确率为 97.19% 。

第五章 问题及心得体会

遇到一个非常棘手的问题，训练速度特别慢，就连前向传递过程都十分慢，想了好久都找不到问题究竟在哪。在将所有 C++ STL 结构改为指针，进行动态内存分配后，速度提升大约 60 倍。之前的代码之所以慢是因为 STL 数据结构 push_back 的过程，当 push_back 时所需的空间大于已经分配的空间时，会重新分配比原来大 1.5 倍的内存，不断重新分配内存的过程开销是十分大的。

通过本次实验看到了自己很多薄弱点，该要踏实努力。

第六章 参考文献

[1] 一文弄懂神经网络反向传播算法（http://www.cnblogs.com/charlotte77/p/5629865.html）

[2] 神经网络中权值初始化的方法 (https://blog.csdn.net/u013989576/article/details/76215989)

[3] 神经网络之激活函数(Activation Function) (https://blog.csdn.net/cyh_24/article/details/50593400)

[4] A Step by Step Backpropagation Example (https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/)

[5] Principles of training multi-layer neural network using backpropagation (http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html)