

西安交通大学

论文

使用 MNIST 手写数据集训练基于 bp 算法的神经网络

作者：徐子越

班级：计算机 72 班

学号：2174410455

2019 年 12 月

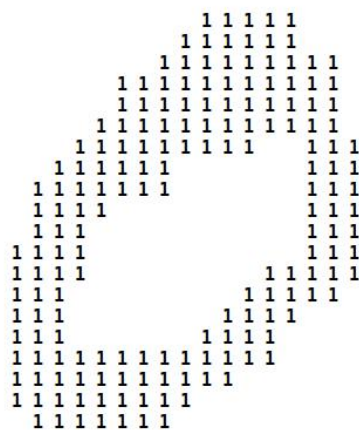
摘 要

本文基于人工智能课上所学，使用神经网络中的 bp 算法来训练 MNIST 手写数据集的分类器。

关键词：人工智能；神经网络；bp 算法；MNIST 数据集

1、问题描述

MNIST 数据集来自美国国家标准与技术研究所，National Institute of Standards and Technology (NIST)。训练集 (training set) 由来自 250 个不同人手写的数字构成，其中 50% 是高中学生，50% 来自人口普查局 (the Census Bureau) 的工作人员。测试集 (test set) 也是同样比例的手写数字数据。数据图片以字节的形式进行存储，读取之后的单个图片样例如下。



BP (Back Propagation) 网络是 1986 年由 Rumelhart 和 McClelland 为首的科学家小组提出，是一种按误差逆传播算法训练的多层前馈网络，是应用最广泛

的神经网络模型之一。BP 网络能学习和存贮大量的输入-输出模式映射关系，而无需事前揭示描述这种映射关系的数学方程。它的学习规则是使用最速下降法，通过反向传播来不断调整网络的权值和阈值，使网络的误差平方和最小。BP 神经网络模型拓扑结构包括输入层 (input)、隐层(hidden layer)和输出层(output layer)。

BP 算法的一般流程：

BP 算法是一个迭代算法，它的基本思想为：(1) 先计算每一层的状态和激活值，直到最后一层（即信号是前向传播的）；(2) 计算每一层的误差，误差的计算过程是从最后一层向前推进的（这就是反向传播算法名字的由来）；(3) 更新参数（目标是误差变小），迭代前面两个步骤，直到满足停止准则（比如相邻两次迭代的误差的差别很小）

2、算法分析

本文构建 bp 神经网络，分为输入层，输出层和隐藏层。输入层为 MNIST 数据集的图片向量格式，每一次训练输入一张图。输出层为每一张图片对应的数字的选择。隐藏层从输入层读入数据，每一隐藏层的每一个结点取前一层的所有结点的加权和，采用可微的激活函数加以输出。一次训练后，在输出和输入图片的期望之间计算欧式距离的值作为误差。在误差的反向传播过程中使用误差对权值求偏分可以计算出权值的变化量，一层一层的向前计算直到隐藏层第一层的权值组合。一张图当做一次训练，可以通过设定训练步长来限制一次训练的下降幅度。在输入了全部的 MNIST 训练集之后用 MNIST 测试集加以测试，如果达到我们要求的正确率就停止训练，否则重新输入 MNIST 训练集继续迭代训练。

3、C++代码实现

```
#include <bits/stdc++.h>
#define input_init 0
#define mid_init 1
#define output_init 2
using namespace std ;
vector<double>labels;

vector<vector<double>>>images;//训练集
vector<double>labels1;

vector<vector<double>>>images1;//测试集

const int train_number=60000;//训练样本数

const int test_number=5000;//测试样本数

const int image_bit = 785;

const int mid_layer_num = 2;//中间层个数

const int mid_node_num = 200;//中间层 node 个数

const int out=10;

const double alpha=0.10;//学习率

const double e = 0.0001;

static int now = 1;

//均匀分布给出权值随机数, 范围在(0~1/)

//s 设置不同层的权值

static double average(int s){
    int temp = ((rand())%100)-50;
    if(s == input_init)
        return double(temp)/double(140);
```

```

        else
            return double(temp)/double(90);
    }

    //西格玛函数  $o=f(net)=1/(1+\exp(-net))$ 
    double sigema(double temp){
        return 1/(1+exp(-temp));
    }

    //没有输入层,输入层就是 vector<double>images/images1
    class mid_Node{
    public:
        double input = 0;
        double output = 0;
        vector<double> weight;
        vector<double> delta_weight;

        //获得改变值,in 应该为上一中间层的
        vector<double> getdelta_weight(vector<double > &w,
vector<double > &s, vector<double > &upper_o){

            double temp1 = 0;
            for(int i = 0; i<w.size();i++)
                temp1 += w[i]*s[i];
            for(int i = 0; i<upper_o.size();i++)
                this->delta_weight[i] =
((this->output*(1-this->output)*(temp1)*upper_o[i]));
            for(int i = 0; i<this->delta_weight.size();i++)
                this->weight[i] += this->delta_weight[i];
            return this->delta_weight;
        }

        //计算这个单元的输出
        void setinput(vector<double> &in){

```

```

        this->input = 0;
        for(int i = 0; i<in.size(); i++){
            this->input += in[i]*this->weight[i];
        }
        this->output = sigema(this->input);
        return;
    }
    mid_Node(vector<double> &in, vector<double> &weight_temp){
        this->weight = weight_temp;
        this->delta_weight.resize(mid_node_num);
        this->setinput(in);
    }
};

class output_Node{
public:
    double value;
    double target;
    double output;
    vector<double> delta_weight;
    vector<double> weight;

    //获得权重的改变值
    vector<double> getdelta_weight(vector<double> > &in,
vector<double> > &upper_o){
        this->delta_weight.resize(this->weight.size());
        for(int i = 0 ;i<this->weight.size();i++) {
            this->delta_weight[i] = (alpha * (output * (1 - output)
* (target - output)) * upper_o[i]);
        }
        return this->delta_weight;
    }

    //根据权值改变值更新权值

```

```

void change_weigt(vector<double> &change){
    for(int i = 0; i<change.size();i++)
        this->weigt[i] += change[i];
    return;
}

//从上一层获得 value

void getvalue(vector<double > &in){
    this->value = 0;
    for(int i=0; i<this->weigt.size(); i++)
        this->value += in[i]*this->weigt[i];
    return;
}

//初始化

output_Node(vector<double> &in, double temp, vector<double>
&weigt_temp){
    this->target = temp;
    this->weigt = weigt_temp;
    this->getvalue(in);
}

};

class mid_Layer{
public:
    vector<mid_Node> mid_node;
    vector<vector<double>> mid_weight;
    vector<double> mid_layer_out;

    void setlayerout(){//本层的输出
        for(int i = 0; i<this->mid_node.size();i++){
            this->mid_layer_out[i] = (this->mid_node[i].output);
        }
        return;
    }
};

```

```

    }
    void train(vector<double> &in_every_layer) {
        for (int i = 0; i < this->mid_node.size(); i++)
            this->mid_node[i].setinput(in_every_layer);
        return;
    }
    //初始化
    mid_Layer(vector<double> &in, vector<vector<double>>
&weight_temp){
        this->mid_weight = weight_temp;
        this->mid_layer_out.resize(mid_node_num);
        mid_Node temp(in, this->mid_weight[0]);
        for(int i = 0; i<mid_node_num; i++){
            temp = mid_Node(in, weight_temp[i]);
            this->mid_node.push_back(temp);
        }
        //for()
    }
};

class total_Layer{
public:
    vector<vector<double>>>weight;
    vector<mid_Layer> mid_layer;
    vector<output_Node> out_Node;
    vector<double>outputlayer_output;
    double target;
    double dif(){
        double temp = 0;
        for(int i = 0; i<out; i++){
            if(i == this->target)
                temp +=
0.5*double(this->outputlayer_output[i]-1)*double(this->outputl

```



```

ayer_output[i]-1);
        else
            temp +=
0.5*(this->outputlayer_output[i])*(this->outputlayer_output[i])
;
    }
    return temp;
}

//设置目标值 y[out], 即输出层
void getoutput(){
    this->outputlayer_output.resize(out);
    double t = 0;
    for(int i = 0; i<out_Node.size(); i++)
        t+=out_Node[i].value;
    for(int i = 0; i<out; i++){
        this->out_Node[i].output =
sigma(this->out_Node[i].value/t);
        this->outputlayer_output[i] =
(this->out_Node[i].output);
    }
    return;
}

//设置输出层
void setoutputlayer(int x, vector<double> &in, double tar,
vector<double>&weight_temp){
    this->target = tar;
    output_Node temp(in, 0, weight_temp);
    if(x == tar)
        temp = output_Node(in, tar, weight_temp);
    else
        temp = output_Node(in, 0, weight_temp);
}

```

```

        this->out_Node[x] = (temp);
        return;
    }

    void train1(vector<double> &in_every_layer, double tar){
        vector<double> temp, temp1;
        temp1 = in_every_layer;
        for(int i = 0; i<this->mid_layer.size(); i++){
            this->mid_layer[i].train(temp1);
            this->mid_layer[i].setlayerout();
            temp1 = this->mid_layer[i].mid_layer_out;
        }
        for(int i = 0; i < out ; i++)
            setoutputlayer(i, in_every_layer, tar,
this->weight[this->weight.size()-10+i]);
        getoutput();
        return;
    }

    double test1(vector<double> &in_every_layer, double tar){
        this->out_Node.clear();
        vector<double> temp = in_every_layer;
        for(int i = 0; i<this->mid_layer.size(); i++){
            this->mid_layer[i].train(temp);
            this->mid_layer[i].setlayerout();
            temp = this->mid_layer[i].mid_layer_out;
        }
        for(int i = 0; i<out; i++)
            setoutputlayer(i, in_every_layer, tar,
this->weight[this->weight.size()-10+i]);
        getoutput();
        return this->dif();
    }

```

```

total_Layer(vector<vector<double>>&weight_temp, double tar){

    this->weight = weight_temp;//权值矩阵在 main 给出

    //初始化中间层,最开始的
    vector<double> in_every_layer = images[0];
    for(int i = 0; i<mid_layer_num; i++){
        mid_Layer temp(in_every_layer, this->weight);
        this->mid_layer.push_back(temp);
        temp.setlayerout();
        in_every_layer = temp.mid_layer_out;
    }
    //初始化 out_Node
    output_Node temp(in_every_layer, tar,
this->weight[this->weight.size()-10]);
        for(int i = 0; i<out; i++)
            this->out_Node.push_back(temp);
        for(int i = 0; i < out ;i++)
            setoutputlayer(i, in_every_layer, tar,
this->weight[this->weight.size()-10+i]);
        getoutput();
    }
};

int ReverseInt(int i)
{
    unsigned char ch1, ch2, ch3, ch4;
    ch1 = i & 255;
    ch2 = (i >> 8) & 255;
    ch3 = (i >> 16) & 255;
    ch4 = (i >> 24) & 255;
    return(((int)ch1 << 24) + ((int)ch2 << 16) + ((int)ch3 << 8) +
ch4;
}

```

```

void read_Mnist_Label(string filename, vector<double>& labels)
{
    ifstream file;
    file.open(filename, ios::binary);
    if (file.is_open())
    {
        int magic_number = 0;
        int number_of_images = 0;
        file.read((char*)&magic_number, sizeof(magic_number));
        file.read((char*)&number_of_images,
sizeof(number_of_images));
        magic_number = ReverseInt(magic_number);
        cout<<magic_number<<endl;
        number_of_images = ReverseInt(number_of_images);
        cout<<number_of_images<<endl;
        for (int i = 0; i < number_of_images; i++)
        {
            unsigned char label = 0;
            file.read((char*)&label, sizeof(label));
            labels.push_back((double)label);
        }
    }
}

void read_Mnist_Images(string filename,
vector<vector<double> >&images)
{
    ifstream file(filename, ios::binary);
    if (file.is_open())
    {
        int magic_number = 0;
        int number_of_images = 0;
        int n_rows = 0;

```

```

    int n_cols = 0;
    file.read((char*)&magic_number, sizeof(magic_number));
    file.read((char*)&number_of_images,
sizeof(number_of_images));
    file.read((char*)&n_rows, sizeof(n_rows));
    file.read((char*)&n_cols, sizeof(n_cols));
    magic_number = ReverseInt(magic_number);
    number_of_images = ReverseInt(number_of_images);
    n_rows = ReverseInt(n_rows);
    n_cols = ReverseInt(n_cols);
    for (int i = 0; i < number_of_images; i++)
    {
        vector<double> tp;
        for (int r = 0; r < n_rows; r++)
        {
            for (int c = 0; c < n_cols; c++)
            {
                unsigned char image = 0;
                file.read((char*)&image, sizeof(image));
                tp.push_back(image);
                tp.back() /= 255;
            }
        }
        tp.push_back(1);
        images.push_back(tp);
    }
}

vector<vector<double>> init(){
    //初始化权值
    srand((unsigned int)time(NULL));

```

```

vector<vector<double>>>dad(mid_layer_num*mid_node_num+out);
vector<double>son;

//+1 是将顺便将偏置加入 weight

//中间层第一层与输入层的权值链接
son.resize(image_bit);
for(int j = 0; j<mid_node_num;j++){

    for(int i = 0; i<image_bit; i++){

        double temp = average(input_init);
        son[i] = temp;
    }

    //中间层第二层开始
    dad[j] = son;
}
son.resize(mid_node_num);
for(int z = mid_node_num; z<dad.size()-out; z++){
    for(int i = 0; i<mid_node_num; i++){
        double temp = average(input_init);
        son[i] = (average(temp));
    }
    dad[z] = son;
}

//输出层与最后一层的权值链接
for(int k = dad.size()-out; k<dad.size(); k++) {
    for (int i = 0; i < mid_node_num; i++) {
        double temp = average(input_init);
        son[i] = average(temp);
    }
}

```

```

        dad[k] = son;
    }
    return dad;
}

void train(total_Layer &dnn){
    vector<double> temp;
    for(int i = 0; i<dnn.out_Node.size();i++){
        temp =
dnn.out_Node[i].getdelta_weight(dnn.outputlayer_output,
dnn.mid_layer[dnn.mid_layer.size()-1].mid_layer_out);
        dnn.out_Node[i].change_weight(temp);

    }

//输出层到倒数第二层,后网

    for(int i = dnn.mid_layer.size()-1;
i>=(dnn.mid_layer.size()-1); i--){
        for(int j = 0; j<dnn.mid_layer[i].mid_node.size();j++){
            vector<double> w, s, upper_o;
            for(int k = 0;
k<dnn.mid_layer[i].mid_node[j].weight.size();k++){
                w.clear(); s.clear(); upper_o.clear();
                for(int z = 0; z<out; z++) {
                    w.push_back(dnn.out_Node[z].weight[j]); //w为下
一层到本节点的w

s.push_back(dnn.out_Node[z].delta_weight[j]); //s为下一层到本节点的
delta_w

                }
            }
            for(int k = 0;

```

```

k<dnn.mid_layer[i-1].mid_node.size();k++){

upper_o.push_back(dnn.mid_layer[i-1].mid_node[k].output);//upper_o 为上一层对本节点的输出

    }

dnn.mid_layer[i].mid_node[j].getdelta_weight(w,s,upper_o);
    }
}

//除了后网和前网的中间网

    for(int i = dnn.mid_layer.size()-2; i>=0; i--){//i 为中间层序号

        for(int j = 0; j<dnn.mid_layer[i].mid_node.size();
j++){//j 为中间层节点序号

            vector<double> w, s, upper_o;
            for(int k = 0;
k<dnn.mid_layer[i].mid_node[j].weight.size();k++){//k 为中间层的权重的序号

                w.clear();
                s.clear();
                upper_o.clear();
                for(int z = 0; z<mid_node_num; z++) {

w.push_back(dnn.mid_layer[i+1].mid_node[z].weight[j]);//w 为下一层到本节点的w

s.push_back(dnn.mid_layer[i+1].mid_node[z].delta_weight[j]);//

```


s 为下一层到本节点的 delta_w

```
        }
    }
    if(i == 0)
        upper_o = images[now];
    else
        for(int z = 0; z<dnn.mid_layer[i-1].mid_node.size();
z++){
```

```
upper_o.push_back(dnn.mid_layer[i-1].mid_node[z].output); //upp
```

er_o 为上一层对本节点的输出

```
        }
        if(i == 0){
            dnn.mid_layer[i].mid_node[j].getdelta_weight(w, s,
images[now]);
        }
        else
```

```
dnn.mid_layer[i].mid_node[j].getdelta_weight(w,s,upper_o);
    }
}
now++;
dnn.train1(images[now], labels[now]);
return;
}
```

```
double test(total_Layer &tnn, int now1){
    tnn.test1(images1[now1],labels1[now1]);
}
int main()
{
```

```

    read_Mnist_Label("/home/xzy/文档
/MINST/train-labels.idx1-ubyte", labels);

    read_Mnist_Images("/home/xzy/文档
/MINST/train-images.idx3-ubyte", images);

    read_Mnist_Label("/home/xzy/文档
/MINST/t10k-labels.idx1-ubyte", labels1);

    read_Mnist_Images("/home/xzy/文档
/MINST/t10k-images.idx3-ubyte", images1); //读取mnist数据集

    vector<vector<double>> init_weight=init();
    total_Layer wheel(init_weight, labels[0]);
    cout<<wheel.dif()<<"difdfidfi"<<endl;
    //    for(int i = 0; i<wheel.out_Node.size();i++)
    //        cout<<wheel.out_Node[i].output<<" ";

    double temp = 0;
    while(1){
        temp = 0;
        while(now<train_number){
            train(wheel);
            cout<<"next picture:"<<endl;
            for(int i = 0; i<wheel.outputlayer_output.size();i++){
                cout<<wheel.outputlayer_output[i]<<" ";
            }
            cout<<wheel.dif()<<endl;
        }
        cout<<"train-finish"<<endl;
        for(int i = 0 ;i<test_number; i++){
            temp += test(wheel, i);
        }
        cout<<"test"<<temp<<endl;
        now = 1;
        if(temp<0.03){

```

```

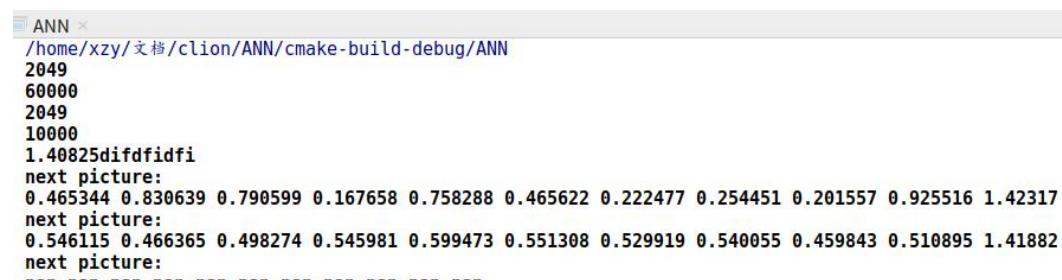
        cout<<"final_test"<<temp<<endl;
        break;
    }
}

//输出 image 测试
//    for(int i = 1;i<2;i++){
//        for(int j = 0;j<images[i].size();j++){
//            if(images[i][j]==0)
//                cout<<" "<<" ";
//            else
//                cout<<images[i][j]<<" ";
//            if(j%28==0)cout<<"| "<<endl;
//        }
//    }
return 0;
}

```

4、程序运行结果

运行之后每一张图片的输出以及误差如下：



```

ANN x
/home/xzy/文档/clion/ANN/cmake-build-debug/ANN
2049
60000
2049
10000
1.40825d1fd1d1
next picture:
0.465344 0.830639 0.790599 0.167658 0.758288 0.465622 0.222477 0.254451 0.201557 0.925516 1.42317
next picture:
0.546115 0.466365 0.498274 0.545981 0.599473 0.551308 0.529919 0.540055 0.459843 0.510895 1.41882
next picture:
xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx

```

因为 C++ 使用了较多 vector，运行较慢。在经过较长时间的训练之后终于达到了作者所要求的 97% 的正确率。

final_test! diff:0.0299!

参考文献

鲍军鹏, 张选平. 人工智能导论[M]. 西安交通大学. 机械工业出版社