中山大学数据科学与计算机学院

移动信息工程专业-人工智能

本科生实验报告

（2017-2018 学年秋季学期）

课程名称：**Artificial Intelligence**

[redacted]			
[redacted]			

一、 实验题目

Back Propagation Neural Network

二、 实验内容

1. 算法原理

（1）BPNN 是一种按照误差反向向传播训练的多层前馈神经网络。它的基本思想是梯度下降法，利用梯度搜索技术，期望使得网络的实际输出和期望输出值的误差均方差最小。

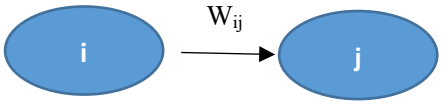
（2）BPNN 算法：

BPNN 分为两个过程：① 工作信号正向传播；② 误差信号反向传播。

假设在 BP 神经网络中，有 m 个输入，有 n 个输入，在输入输出层之间有 h 个隐藏层。

- 正向传播

设结点 i 和结点 j 之前的权值为 W_{ij} ,结点 j 的阈值为 b_j,



每个结点的输出值是根据上层所有结点的输出值、当前结点与上层所有结点的权值、当前结点的阈值以及激活函数来实现的。

每个结点的输入值：$I_j = \sum_{i=0} W_{ij}\, x_i + b_j$

输出值：$O_J = f(I_j)$

其中 f 为激活函数.

- 反向传播

假设输出层每个结点的输出结果为d_j,误差函数为：$E_j = \frac{1}{2}\sum_{j=0}^{n-1}(d_j - y_j)^2$

神经网络的主要目的是反复修正权值和阈值，使得误差函数达到最小。

根据梯度下降法，权值矢量的修正正比于当前位置上 Error 的梯度，所以对于第 j 个输出结点有：$.\Delta W_j = -\eta \frac{\partial \mathrm{E}}{\partial \mathrm{Wj}}$

根据链式求导，可得：

$$\frac{\partial \mathrm{E}}{\partial W_j} = \frac{\partial}{\partial W_j}\frac{1}{2}(d_j - y_j)^2$$

$$y_j = f(h) \quad and \quad h = \sum_j w_j x_j$$

$$\frac{\partial \mathrm{E}}{\partial W_j} = -(d_j - y_j)\frac{\partial d_j}{\partial W_j} = -(d_j - y_j)f'(h)\frac{\partial \sum_j w_j x_j}{\partial W_j}$$

$$= -(d_j - y_j)f'(h) x_j$$

令 $\delta = (d_j - y_j)f'(h)$

所以 $W_j = W_j + \eta\,\delta\, x_i$

其他层的权值更新步长也是以同样的方法进行计算。

（3）BPNN 的优缺点：

- 优点：

  ① BP 神经网络具有较强的非线性映射能力。因为 BP 神经网络实质上实现了一个从输入到输出的映射功能，数学理论证明三层的神经网路就能够以任意精度逼近任何非线性连续函数。

  ② BP 神经网络具有高度自学习和自适应能力；

  ③ BP 神经网络具有将学习成果应用于新知识的能力；

  ④ BP 神经网络在其局部的神经元受到破坏后对全局的训练结果不会造成很大的影响，具有一定的容错能力。

- 缺点：

  ① 训练次数多，学习速度很慢，收敛速度慢；

  ② 因为 BP 神经网络中极小值比较多，所以容易陷入局部极小值而得不到全局最优解。

  ③ 网络层数、神经元个数的选择没有相应的理论指导；

2. 伪代码

```
void trainning(int iteration){
    初始化输入层到隐藏层的权重为 0~1 之间的随机数；
    初始化隐藏层到输出层的权重为 0~1 之间的随机数；
    while(迭代 iteration 次){
        for train 中的每一个训练样本 sample
            //向前传播
            for 隐藏层或输出层的每个单元 j
                计算输入：I_j = ∑_i w_i x_i + θ_j
                计算输出：O_j = 1/(1+e^{-I_j})
```

//向后传播，同时更新权重

误差 Error=$\frac{1}{2}*(O_j - rightoutput_j)^2$

计算输出结点的误差梯度$\delta^o = (y-\hat{y})f'(z)$ 这里 $z=\sum_j W_j a_j$ 是输出节点的输入。

for 隐藏层的每个单元 j 对应的权重W_j

权重步长：$\Delta W_j = \Delta W_j + \delta^o a_j$

更新权重：$W_j = W_j + \eta \Delta W_j / m$

误差传播到隐藏层时误差梯度：$\delta_j^h = \delta^o W_j f'(h_j)$

for 输入层的每个单元对应的权重W_{ij}

权重步长：$\Delta w_{ij} = \Delta w_{ij} + \delta_j^h a_i$

更新权重：$w_{ij} = w_{ij} + \eta \Delta w_{ij} / m$

}

}

3. 关键代码截图（带注释）

（1）设置隐藏层结点数，输出结点数，输入结点数，隐藏层层数、学习速率。

```
#define innode 12     //输入结点数
#define hidenode 5    //隐藏层结点数
#define hidelayer 1   //隐藏层层数
#define outnode 1     //输出结点数
#define alpha 0.1     //学习速率
```

（2）初始化输入层到隐藏层的权重和隐藏层到输出层的权重。

```
//初始化输入层到隐藏层的w 和 隐藏层到输出层的w
void initialize_weight(double weight) {
    for (int i = 0;i < innode+1;i++) {
        vector<double> v;
        for (int j = 0;j < hidenode;j++) {
            v.push_back(get_random());
        }
        input_w.push_back(v);
    }
    for (int i = 0;i < hidenode;i++) {
        hidden_w.push_back(get_random());
    }
```

（3）让训练集中的每一个训练样本向前传播。

计算出隐藏层的输入：

```
vector<double> hidden_in;
for (int i = 0;i < hidenode;i++) {
    double sum = 0;
    for (int j = 0;j < innode+1;j++) {
        sum += sample[j] * input_w[j][i];
    }
    hidden_in.push_back(sum);
}
```

计算隐藏层的输出：

```
hidden_out.clear();
for (int i = 0;i < hidenode;i++) {
    double res = sigmoid(hidden_in[i]);
    hidden_out.push_back(res);
}
```

```
double sigmoid(double x)
{
    double res = 1.0 / (1 + exp(-x));
    return res;
}
```

计算输出层的输入：

```
//输出层的输入
double output_in=0;
for (int i = 0;i < hidenode;i++) {
    output_in += 1.0*hidden_out[i] * hidden_w[i];
}
```

计算输出层的输出：

```
//输出层的输出
output=output_in;
```

（4）让训练集的每一个样本向后传播，同时更新输入层到隐藏层的权重和隐藏层到输出层的权重。

计算出隐藏层->输出层到的权重更新步长：

```
double hw = 1.0*(rightout- output);
for (int i = 0;i < hidenode;i++) {
    double hw1 = hw*hidden_out[i];
    renew_hw.push_back(hw1);
}
```

计算出输入层->隐藏层的权重更新步长并更新权重：

```
for (int i = 0;i < innode+1;i++) {
    for (int j = 0;j < hidenode;j++) {
        double iw = 1.0*hw*hidden_w[j] * hidden_out[j] * (1 - hidden_out[j])*sample[i];
        input_w[i][j] = input_w[i][j] + 1.0*alpha*iw / (innode+1);
    }
}
```

根据隐藏层->输出层的权重更新步长更新权重：

```
for (int i = 0;i < hidenode;i++) {
    hidden_w[i] = hidden_w[i] + 1.0*alpha*renew_hw[i]/hidenode;
}
```

4. 创新点&优化

（1）尝试使用不同的激活函数——tanh 函数：

```
double tanh(double x)
{
    double res = 1.0*(exp(x) - exp(-x)) / (1.0*exp(x) + exp(-x));
    return res;
}
```

更新权重步长的计算变为：

```
double hw = (output - rightout);
for (int i = 0;i < hidenode;i++) {
    double hw1 = hw*hidden_out[i];
    renew_hw.push_back(hw1);
}

for (int i = 0;i < innode+1;i++) {
    for (int j = 0;j < hidenode;j++) {
        double iw = 1.0*hw*hidden_w[j]  * (1 - hidden_out[j]* hidden_out[j])*sample[i];
        input_w[i][j] = input_w[i][j] + 1.0*alpha*iw / (innode+1);
    }
}
```

（2）对 train 进行归一化：

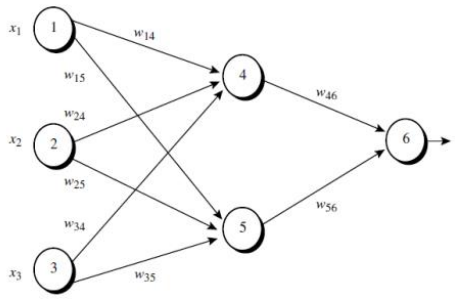
```
void normalization(vector<vector<double> > v){
    int row=v.size(),column=v[0].size();
    double max=0,min=1000000;
    for(int i=0;i<column;i++){
        for(int j=0;j<row;j++){
            if(v[j][i]>max) max=v[j][i];
            if(v[j][i]<min) min=v[j][i];
        }
        for(int j=0;j<row;j++){
            v[j][i]= 1.0*(v[j][i]-min)/(max-min);
        }
    }
}
```

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

小数据测试模型构建是否正确：

一个训练元组 X=（1,0,1），标签为 1,学习率 alpha=0.9



初始化 weight 和 bias:

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2

那么隐藏层和输出层的输入输出如下图：

Unit j	Net Input	Output
4	0.2+0-0.5-0.4=-0.7	$\frac{1}{1+e^{0.7}} = 0.332$
5	-0.3+0+0.2+0.2=0.1	$\frac{1}{1+e^{-0.1}} = 0.525$
6	(-0.3)(-.332)-0.2*(0,525)= -0.205	$\frac{1}{1+e^{0.205}} = 0.449$

```
隐藏层的输入：
h0=-0.7
h1=0.1
隐藏层的输出：
0.331812
0.524979
hidden=-0.3
hidden=-0.2
输出层的输入：
-0.20454
输出层的输出：
0.449043
```

Weight 和 bias 更新值：

W_{46}	W_{56}	θ_4	θ_5	W_{14}	W_{15}	W_{24}	W_{25}	W_{34}	W_{35}
-0.045	-0.072	0.00907	0.006798	0.00907	0.006798	0	0	0.00907	0.006798

```
隐藏层->输出层的权值更新值：
-0.0452289
-0.0715592
输入层->隐藏层的权值更新值：
0.00906642
0.00679842
0.00906642
0.00679842
0
0
0.00906642
0.00679842
```

更新后的 W：

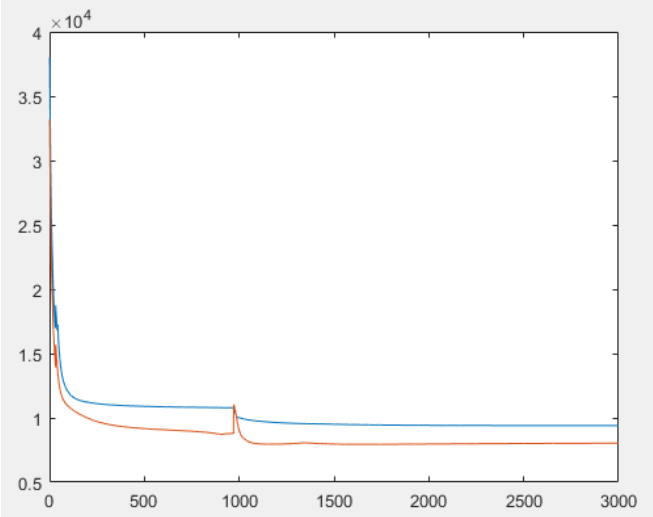
W_{46}	W_{56}	θ_4	θ_5	W_{14}	W_{15}	W_{24}	W_{25}	W_{34}	W_{35}
-0.31	-0.2162	-0.397	0.2015	0.20204	-0298	0.4	0.1	-0.497	0.2015

```
隐藏层->输出层更新后的权值：
-0.320353
-0.232202
输入层->隐藏层更新后的权值：
-0.39796
0.20153
0.20204
-0.29847
0.4
0.1
-0.49796
0.20153
```

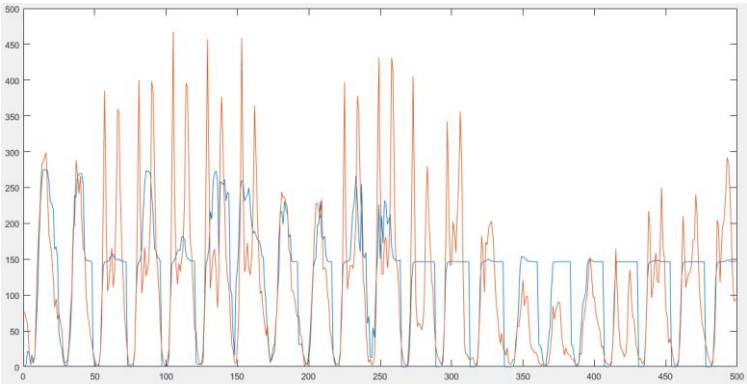
2. 评测指标展示即分析（如果实验题目有特殊要求，否则使用准确率）

（1）使用 sigmoid：

下图为 loss 的下降过程，蓝色为 train_loss 过程，红色为 vali_loss 过程。

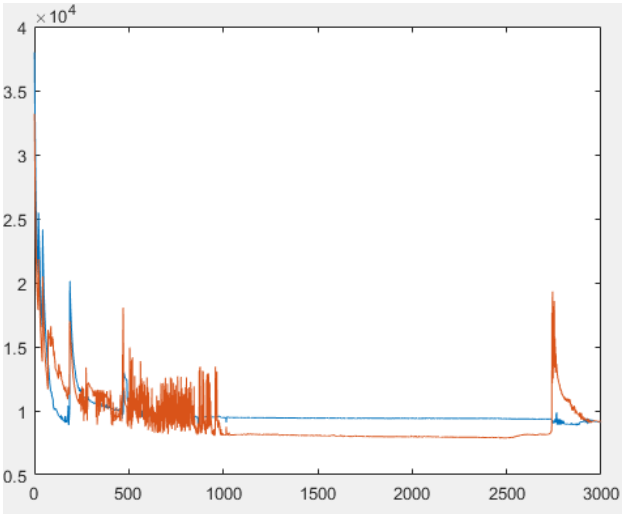


下图是数据集中最后 20 天的数据的预测值和真实值的对比图：(红色为真实值，蓝色为预测值)

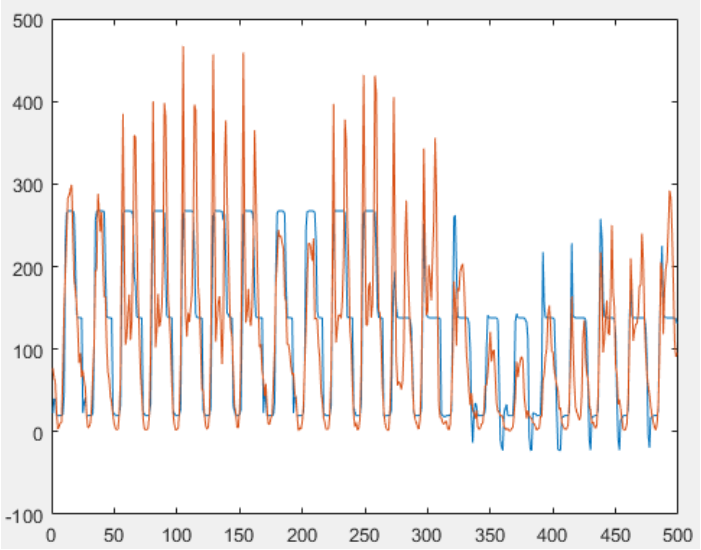


（2）使用 tanh：

下图为 loss 的下降过程，蓝色为 train_loss 过程，红色为 vali_loss 过程。



下图是数据集中最后 20 天的数据的预测值和真实值的对比图：(红色为真实值，蓝色为预测值)



四、 思考题

1. 尝试说明下其他激活函数的优缺点。

答：常用激活函数：

- Sigmoid 函数 $f(x) = \frac{1}{1+e^{-x}}$：
  - 优点：

    Sigmoid 函数能够把输入的连续实值压缩到 0~1 之间，避免因数值相差太多而造成影响。
  - 缺点：

    Sigmoid 函数很容易饱和，当输入非常大或非常小的时候，神经元的梯度就接近 0 了，这会造成在反向传播时反向传播接近于 0 的梯度，导致最终权重基本没什么更新。为了避免饱和，我们要注意参数的初始值的设置。而且 Sigmoid 函数的输出不是 0 均值的，这会导致后层的神经元的输入是非 0 均值的信号，会对梯度产生影响。假设后层神经元的输入都为正，那么对 W 求局部梯度则都为正，这样在反向传播的过程中 W 要么都往正方向更新，要么都往负方向更新，导致有一种捆绑的效果，使得收敛缓慢。
- Tanh 函数 $\tanh(x) = 2\text{sigmoid}(2x) - 1 = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ ：
  - 优点：

    tanh 是 0 均值的，能够将数据压缩到-1 到 1 之间；
  - 缺点：

    tanh 也容易饱和，当梯度非常大或非常小的时候，神经元的梯度接近于 0。

tanh 的收敛速度比较慢，计算比较复杂。

- ReLU 函数 $f(x) = \max(0, x)$：
  - 优点：

    ReLU 是线性函数，而且梯度不会饱和。

    收敛速度比 sigmoid 和 tanh 快；

    ReLU 的计算复杂度较低，因为 Sigmoid 和 tanh 需要计算指数等，计算复杂度高，而 ReLU 只需要一个阈值就可以得到激活值；
  - 缺点：

    ReLU 在训练的时候很脆弱，有可能导致神经元坏死。例如：因为 ReLU 在 x<0 时梯度为 0，那么就会导致负的梯度在这个 ReLU 被置零，而且这个神经元可能再也不会被任何数据激活，那么这个神经元的梯度就永远为 0 了，即这个 ReLU 神经元坏死了。实际操作中，当学习率设置过大时，就容易导致神经元坏死，所以我们要注意学习率的设置，选择一个合适的较小的学习率。
- Maxout 函数 $f(x) = max_{j\in[1,k]}Z_{ij}$：

  假设 W 是二维，那么$f(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$
  - 优点：

    ReLU 是 Maxout 的变形，Maxout 具有 ReLU 的优点。Maxout 函数的拟合能力非常强，可以拟合任意的凸函数；
  - 缺点：

    把参数 double 了，造成参数增多；

2.有什么方法可以实现传递过程中不激活所有节点？

答：给节点的梯度更新步长设置一个极小的阈值，当节点的梯度更新步长小于该阈值时就不激活该节点。

3. 梯度消失和梯度爆炸是什么？可以怎么解决?

答：梯度消失是指梯度趋于 0，梯度爆炸是指梯度趋于无穷大。在神经网络训练时，如果使用反向传播算法，那么在计算梯度时就会进行链式求导，所以计算每一层的梯度时会有连乘的情况。如果连乘的式子小于 1 的话，当连乘的数量增多时，乘积就会越来越小，甚至趋近于 0，那么就会导致梯度消失；如果连乘的式子大于 1 的话，当连乘的数量增多时，乘积就会越来越大，趋近于正无穷，那么就会导致梯度爆炸。

解决方法：

（1）梯度消失：

改用 ReLU 函数作为激活函数，因为它在 x>0 时导数一直为 1，一定程度上能解决梯度消失。

（2）梯度爆炸：

设置梯度阈值，当梯度大于这个阈值的时候就将梯度设置为该阈值，防止梯度继续增大。