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**

教学班级	**1515**	专业（方向）	**软件工程**
学号	**15352334**	姓名	**吴佳卫**

一、 实验题目

BP 神经网络算法

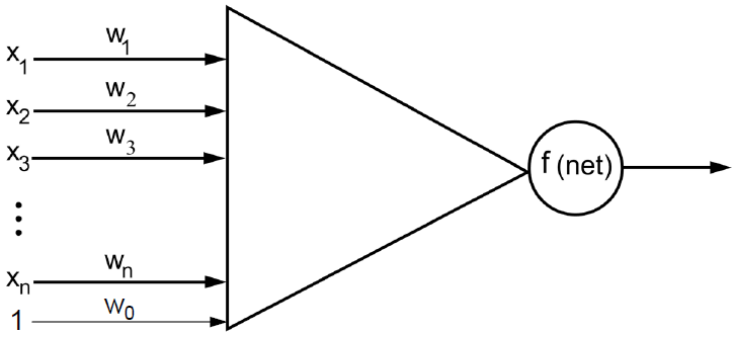
二、 实验内容

1. 算法原理

综述：

神经网络算法的基本原理是让输入的数据x_i通过一个有多层中间节点的网络结构，经过这样的非线性变换以后，预测出输出y_i。通过调整节点之间的连接强度w_{ij}使得误差沿梯度下降，最后到达一定阈值后视为收敛。

节点输出模型：



神经网络算法中的每一个节点都连接了多个输入x_i，以及一个 bias 项。节点的输入的计算公式为$\sum_i w_i x_i$，然后输入会通过一个激活函数并且作为节点的输出。

激活函数：

神经网络算法中的激活函数，其作用就是引入非线性。而在这里我们使用的函数是 sigmoid 函数$\mathrm{sigmod(x)} = \frac{1}{1+e^x}$。

sigmoid 的优点主要有两个，第一是输出的范围是(-1,1)，所以数据在计算的过程中不容易发散，也就是更容易收敛。还有就是 sigmoid 函数的倒数也很好表达$f'(x) = f(x)(1 - f(x))$

BP：

本次修正误差的方法依旧是梯度下降法。

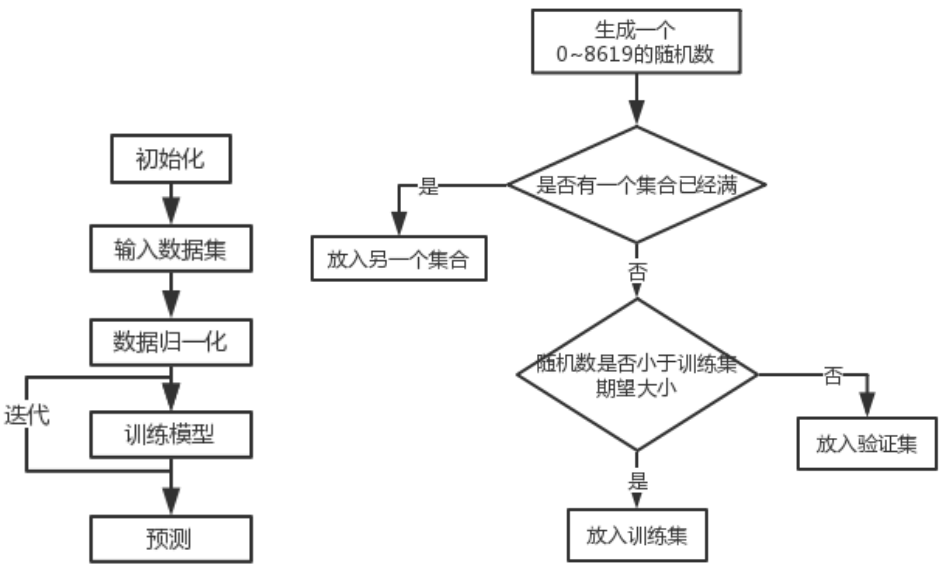
首先我们定义了误差函数$\mathrm{E} = \frac{1}{2}(y - f(\sum w_i\, x_i))^2$，由于我们使用的激活函数为 sigmoid 函数，此时就可以很方便的求出其导数了$f'(x) = f(x)(1 - f(x))$。

我们要计算的是 E 最小时w_i的值，在下降过程中其梯度为$\frac{\partial \mathrm{E}}{\partial w_i}$，带入 E 后化简运算。

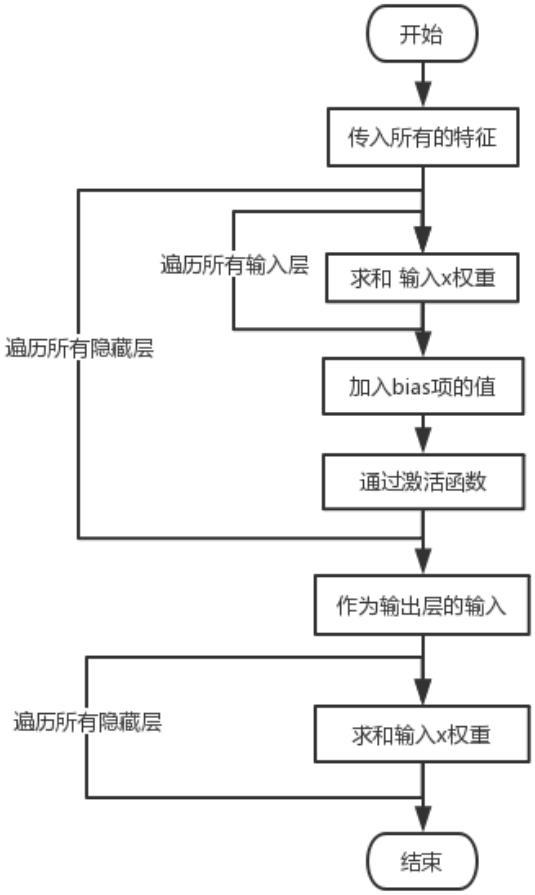
$\frac{\partial \mathrm{E}}{\partial w_i} = -(y - \tilde{y})f'(h)\frac{\partial}{\partial w_i}\sum w_i\, x_i = -(y - \tilde{y})f'(h)x_i$，因此梯度为$(y - \tilde{y})f'(h)$，更新$w_i$的公式为$w_i = w_i + \eta\delta x_i$。其中$\eta$为步长$\delta$为梯度。BP 实现的就是这一个过程。

2. 伪代码

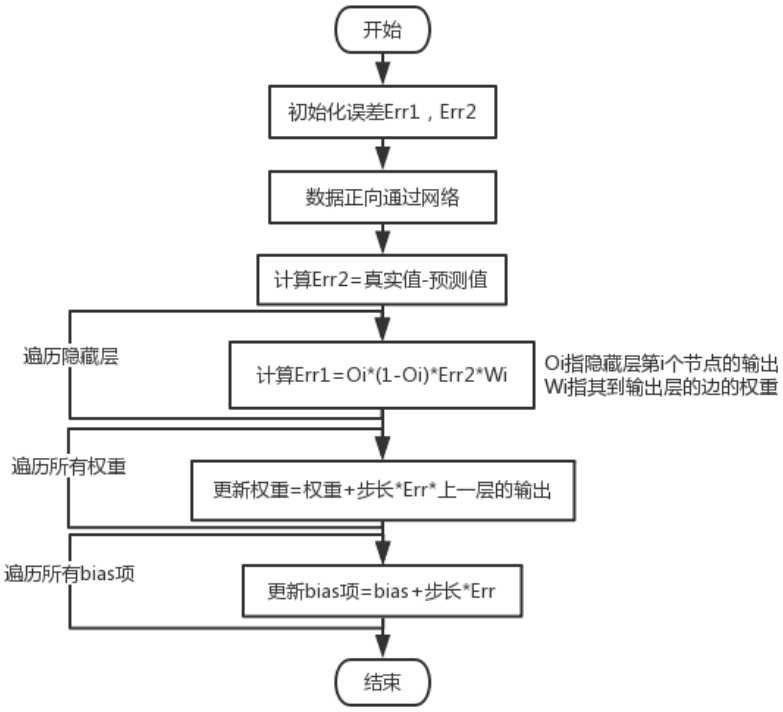
左侧为主函数，右侧为随机分配训练验证集：



数据正向通过神经网络：



反向计算误差并更新权重和 bias 项：



3. 关键代码截图

在为权重 w 赋值的时候使用了随机赋值的方法，通过一个简单的 rand( )函数就可以实现；

```
double random_init() //随机生成0.01-0.1的数字
{
    double b = (rand() % 10) + 1;
    double result = b / 100 ;
    return result;
}
```

同样的，在划分训练集和验证集的时候也是使用了一个随机函数。随机将当前输入的数据划分进验证集或训练集。这种算法避免了原始数据某些特征分布过于集中导致的划分不均匀；

```
int random_cin() //随机分配输入
{
    int a = rand() % 8619;
    if(train_cnt >= train_size) return 1;
    if(val_cnt >= val_size) return 0;
    if(a < train_size) return 0;
    else return 1;
}
```

数据的归一化在这次试验中也很重要，因此这次的归一化同时对训练集和验证集进行；

```
double maxnum = 0;
double minnum = 4000;
//找到训练集和验证集中某一特征的最大最小值
for(int j = 0; j < train_size; j ++)
{
    if(train_set[j].info[i] > maxnum) maxnum = train_set[j].info[i];
    if(train_set[j].info[i] < minnum) minnum = train_set[j].info[i];
}
for(int j = 0; j < val_size; j ++)
{
    if(val_set[j].info[i] > maxnum) maxnum = val_set[j].info[i];
    if(val_set[j].info[i] < minnum) minnum = val_set[j].info[i];
}
if(maxnum != minnum)
{
    for(int j = 0; j < train_size; j ++)
    {
        train_set[j].info[i]=(train_set[j].info[i]-minnum)/(maxnum-minnum);
    }
    for(int j = 0; j < val_size; j ++)
    {
        val_set[j].info[i]=(val_set[j].info[i]-minnum)/(maxnum-minnum);
    }
}
```

传入某一项数据，让其正向通过神经网络；

```
double cal_net(vector<double> info) //正向通过网络输出
{
    for(int i = 0; i < hidden_layer; i++)
    {
        double tmp_result = 0;
        for(int j = 0; j < 12; j ++)
        {
            tmp_result += info[j] * layer1[j][i];
        }
        tmp_result += bias1[i];
        tmp_layer[i] = func(tmp_result); //计算隐藏层数据
    }
    double final_result = 0;
    for(int i = 0; i < hidden_layer; i ++)
    {
        final_result += tmp_layer[i] * layer2[i];
    }
    final_result += bias2;
    return final_result;
}
```

迭代使用梯度下降修正模型的参数（边的权重、bias 项）

```
double Err2;
vector<double> Err1;
Err1.resize(hidden_layer);
for(int i = 0; i < train_size; i ++)
{
    double output = cal_net(train_set[i].info);
    Err2 = train_set[i].flag - output;
    for(int j = 0; j < hidden_layer; j ++)
    {
        Err1[j] = tmp_layer[j] * (1 - tmp_layer[j]) * Err2 * layer2[j];
    }
    bias2 += learning_rate * Err2;
    for(int j = 0; j < hidden_layer; j ++)
    {
        bias1[j] += learning_rate * Err1[j];
    }
    for(int j = 0; j < hidden_layer; j ++)
    {
        //更新第二层的权值
        layer2[j] += learning_rate * Err2 * tmp_layer[j];
    }
    for(int j = 0; j < 12; j ++)
    {
        for(int k = 0; k < hidden_layer; k ++)
        {
            //更新第一层的权值
            layer1[j][k] += learning_rate * Err1[k] * train_set[i].info[j];
        }
    }
}
```

4. 创新点&优化

1) 数据的预处理

第一项是在划分验证集和训练集的时候并不是简单的切分，而是在整个输入的过程中随即将输入的数据划分入训练集和验证集中。因为我们可以从原始的数据集中看出，如果直接划分出训练集，会导致某一个特征的信息熵非常的小，这是很不利于我们训练模型的。同时在处理的时候我们舍去了第一项（编号）和第二项数据（具体的日期）。

```
1, 2011/1/1, 1, 0, 1, 0, 0, 6, 0, 1, 0. 24, 0. 2879, 0. 81, 0, 16
2, 2011/1/1, 1, 0, 1, 1, 0, 6, 0, 1, 0. 22, 0. 2727, 0. 8, 0, 40
3, 2011/1/1, 1, 0, 1, 2, 0, 6, 0, 1, 0. 22, 0. 2727, 0. 8, 0, 32
4, 2011/1/1, 1, 0, 1, 3, 0, 6, 0, 1, 0. 24, 0. 2879, 0. 75, 0, 13
5, 2011/1/1, 1, 0, 1, 4, 0, 6, 0, 1, 0. 24, 0. 2879, 0. 75, 0, 1
6, 2011/1/1, 1, 0, 1, 5, 0, 6, 0, 2, 0. 24, 0. 2576, 0. 75, 0. 0896, 1
7, 2011/1/1, 1, 0, 1, 6, 0, 6, 0, 1, 0. 22, 0. 2727, 0. 8, 0, 2
8, 2011/1/1, 1, 0, 1, 7, 0, 6, 0, 1, 0. 2, 0. 2576, 0. 86, 0, 3
9, 2011/1/1, 1, 0, 1, 8, 0, 6, 0, 1, 0. 24, 0. 2879, 0. 75, 0, 8
10, 2011/1/1, 1, 0, 1, 9, 0, 6, 0, 1, 0. 32, 0. 3485, 0. 76, 0, 14
```

第二项就是对训练集和验证集进行归一化。由于神经网络算法是一个比较依赖代数运算的过程，因此如果某一个特征的数据的取值范围比较大就会倾向于在训练过程中产生更高的影响，归一化就消除了这种影响。

2) 尝试添加输出层的节点

因为在理论课上提到了“数据编码”的概念。也即是对于数字 3 可以编码为 011，这样就可以把只有一个输出层节点的神经网络升级为有三个输出节点的神经网络。

受到这种做法的启发，我尝试着将结果的个位十位和百位分别作为神经网络的三个输出节点。

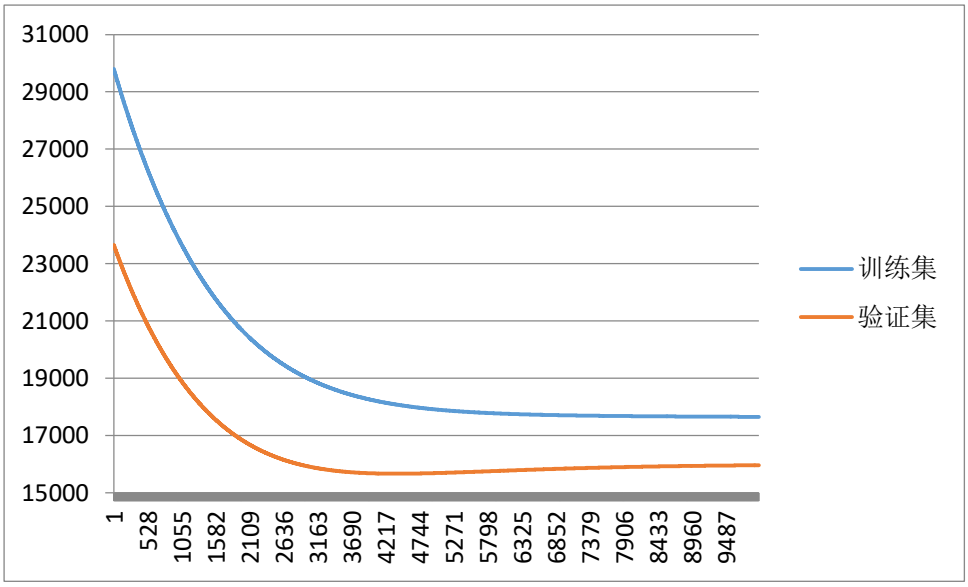
```
tmp = buffer.substr(p1,buffer.length()-p1);
int tmp_num = stod(tmp);
int num3 = tmp_num % 10;
int num2 = (tmp_num / 10) % 10;
int num1 = (tmp_num / 100) % 10;
```

相应的，在进行梯度下降修正权值和 bias 的时候也要有相应的改变：

```
for(int j = 0; j < 3; j ++)
{
    bias2[j] += learning_rate * Err2[j];
}
for(int j = 0; j < hidden_layer; j ++)
{
    bias1[j] += learning_rate * Err1[j];
}
for(int j = 0; j < hidden_layer; j ++)
{
    for(int k = 0; k < 3; k ++)
    {
        layer2[j][k] += learning_rate * Err2[k] * tmp_layer[j];
    }

}
for(int j = 0; j < 12; j ++)
{
    for(int k = 0; k < hidden_layer; k ++)
    {
        layer1[j][k] += learning_rate * Err1[k] * train_set[i].info[j];
    }
}
```

最后运行结果如下，发现相比于一个输出节点并没有更好的效果。考虑到神经网络只实现了一层，复杂度比较低自然拟合的效果也不是很满意。

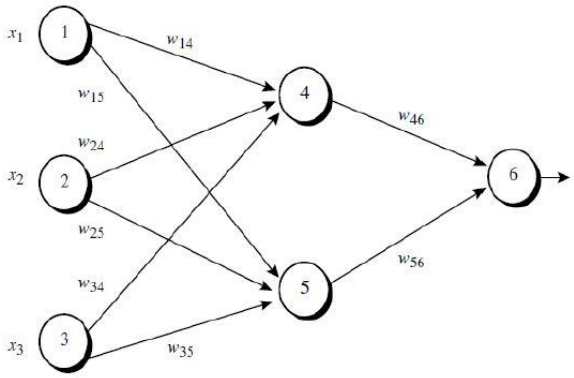


三、 实验结果及分析

1. 实验结果展示示例

为了更加直观便利的判断运算过程的准确性 ,我们使用了课件内提到一个三层神经网络的例子。

One training tuple **X**=(1,0,1), whose class label is 1.

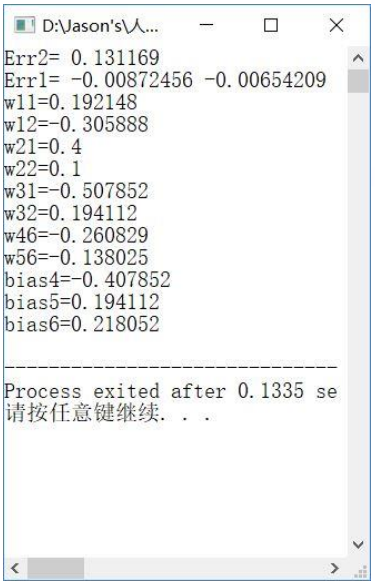


其中数据的初始化如下：

Initial input, weight, and bias values.

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	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

运行结果与答案如下：



Calculation of the error at each node.

Unit j	Err_j
6	$(0.474)(1-0.474)(1-0.474)=0.1311$
5	$(0.525)(1-0.525)(0.1311)(-0.2)=-0.0065$
4	$(0.332)(1-0.332)(0.1311)(-0.3)=-0.0087$

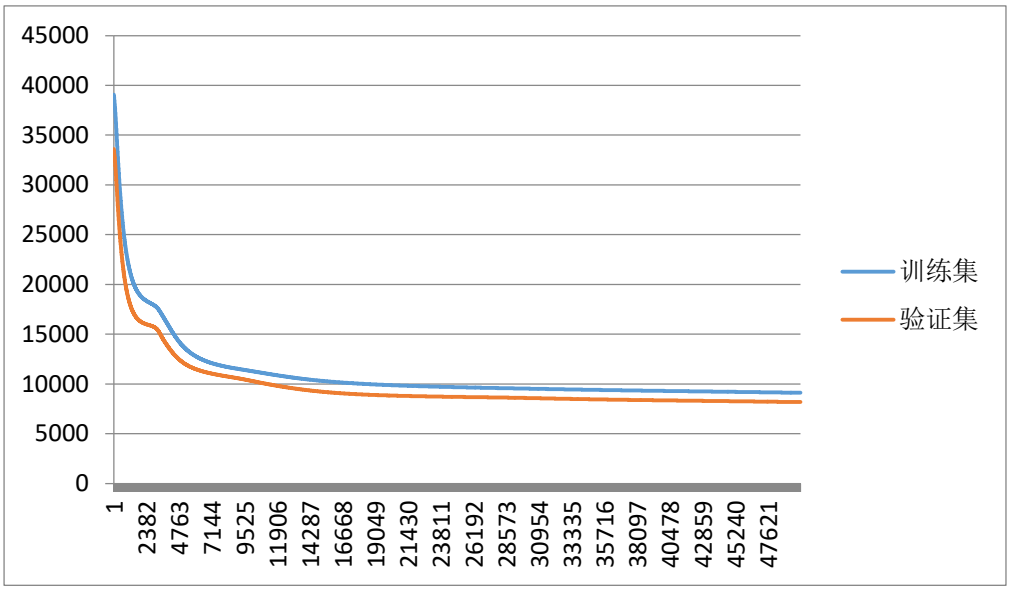
Calculations for weight and bias updating.

Weight or bias	*New value*
w_{46}	$-0.3+(0.9)(0.1311)(0.332)=-0.261$
w_{56}	$-0.2+(0.9)(0.1311)(0.525)=-0.138$
w_{14}	$0.2+(0.9)(-0.0087)(1)=0.192$
w_{15}	$-0.3+(0.9)(-0.0065)(1)=-0.306$
w_{24}	$0.4+(0.9)(-0.0087)(0)=0.4$
w_{25}	$0.1+(0.9)(-0.0065)(0)=0.1$
w_{34}	$-0.5+(0.9)(-0.0087)(1)=-0.508$
w_{35}	$0.2+(0.9)(-0.0065)(1)=0.194$
θ_6	$0.1+(0.9)(0.1311)=0.218$
θ_5	$0.2+(0.9)(-0.0065)=0.194$
θ_4	$-0.4+(0.9)(-0.0087)=-0.408$

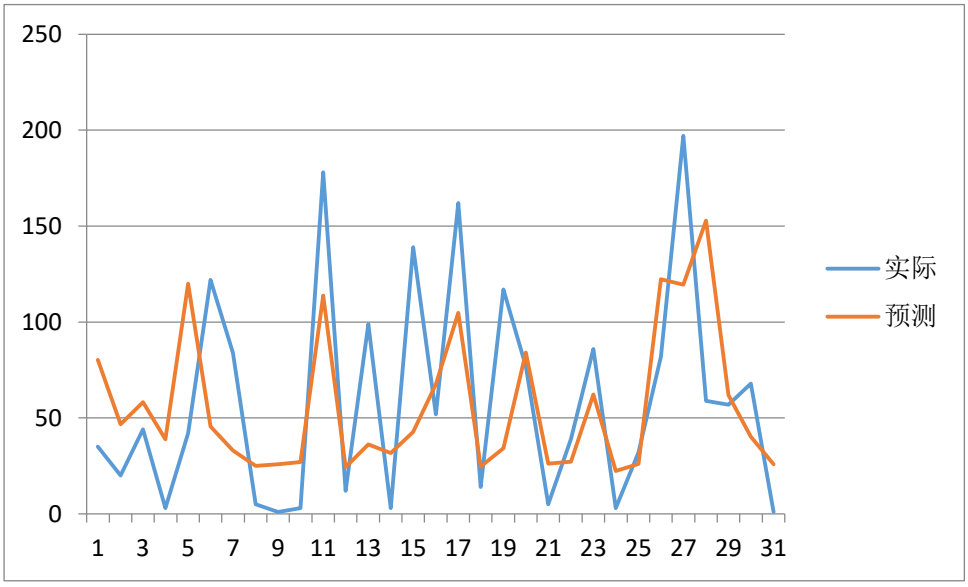
证明了算法是正确的。

2. 评测指标展示即分析

本次一共进行了两项指标的分析，第一项是 lost 函数随着迭代过程下降。



任意取 30 个验证集的样本（注：不是连续的 30 的样本），判断预测结果的准确性：



虽然有少许的误差，但是训练结果大致在令人满意的范围。

四、 思考题

1. 尝试说明下其他激活函数的优缺点。

Tanh 函数

优点：把实值得输入压缩到-1~1 的范围，因此它基本是 0 均值的。

缺点：容易梯度饱和，当输入非常大或者非常小的时候，神经元的梯度接近于 0 了。

ReLU 函数

优点：收敛速度会比 sigmoid/tanh 快很多，计算复杂度低，只需要一个阈值就可以得到激活值。

缺点：有可能导致神经元“坏死”。由于 ReLU 在 x<0 时梯度为 0，这样就导致负的梯度在这个 ReLU 被置零，而且这个神经元有可能再也不会被任何数据激活。

2. 有什么方法可以实现传递过程中不激活所有节点？

理论上来说是可以的，因为参考到聚类的算法，比如仿照 K-Means 算法，可以将特征相近数据放在一起从而忽略掉。

3. 梯度消失和梯度爆炸是什么？可以怎么解决？

梯度消失是因为如果在计算的过程中 ,如果每一层神经元对上一层的输出的偏导乘上权重结果都小于 1，那么经过足够多层传播之后，误差对输入层的偏导会趋于 0。相反的，如果每一层神经元对上一层的输出的偏导乘上权重结果都大于 1，传播以后梯度会指数上升。最近单的解决方法是让误差传递时链式求导的部分为 1，也就是 $|y_{jj}(t)'w_{jj}| = 1.0$