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	**M1**	专业（方向）	**移动信息工程（互联网方向）**
学号	**15352102**	姓名	**韩硕轩**

一、 实验题目

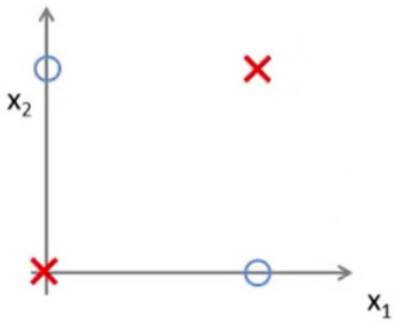
反向传播神经网络（BPNN）

二、 实验内容

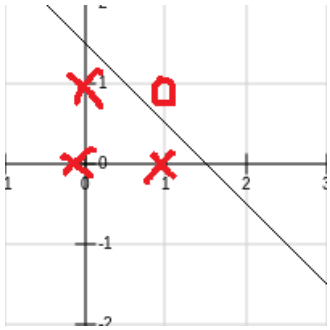
1. 算法原理

本次试验的算法包括两个部分，前向传播和反向传播。我们在 PLA 和逻辑回归的基础上理解神经网络算法。

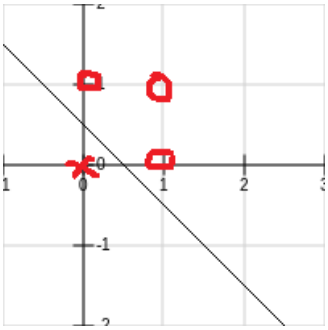
对于一个输入向量，将其与输入层到隐藏层的权重矩阵相乘，就得到了隐藏层的净输入。如果没有隐藏层直接从输入层到输出层，这就是 PLA 的模型。现在有了隐藏层，那么输入层和隐藏层的每一个神经元就构成了一个小的 PLA 模型。从隐藏层到输出层也是同理。多了一个隐藏层的好处是多一组对于数据集的线性划分。比如异或问题，用无隐藏层的神经网络是无法划分的：



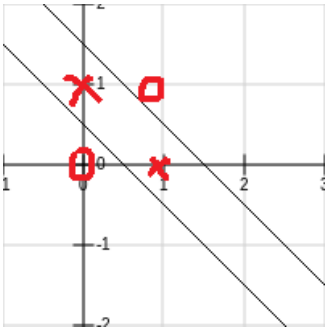
但如果我们先做 and 问题的神经网络，如下图：



再做 or 问题的神经网络，如下图：



那么将两个神经网络合起来，把 and 和 or 的输出神经元放到隐藏层，就可以划分 xor 问题了：



这是前向传播的部分。反向传播是根据误差来调整每两层之间的权重矩阵的。我们对于最终输出的预测结果集有如下函数来判断误差的大小：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

而训练的目的就是使得这个函数值最小。那么就要求它的梯度。一个神经元的误差对于每一个它输出的权重求偏导就是梯度：

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i \propto -\frac{\partial E}{\partial w_i} \longrightarrow \text{THE GRADIENT}$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

所以就能得到每一个隐藏层神经元到输出层的权重的更新公式：

$$\delta = (y - \hat{y}) f'(h)$$

$$w_i = w_i + \eta \delta x_i$$

这个和逻辑回归里面的梯度下降是同样的道理，可以参考二维的曲线来理解，曲线就是误差，而我们算出来的梯度就是曲线的导数，减去它就可以向着下降的方向靠近，每次也就更加逼近最小值。由于有很多个文本，每个文本又要经过很多神经元，所以学习率也需要设置得很合适才行。

神经网络就是按照以上原理实现。

2. 伪代码

```
for i in train_set:
        normalize

initialize eta, number_of_hidden, w_input_to_hidden, w_hidden_to_output
for i from 0 to iterations:
        initialize delta_input_to_hidden, delta_hidden_to_output
        for j in train_set.subset:
                h_hidden = sigmoid(j .* w_input_to_hidden)
                h_output = h_hidden .* w_hidden_to_output
                err_output = j.label - h_output
                err_hidden = mul(h_hidden, ones - h_hidden) .* w_hidden_to_output * err_output
                err_input[k] = j[k] * (err_hidden .* w_input_to_hidden[k])
                delta_input_to_hidden[j][k] = eta * err_input .* h_input
                delta_hidden_to_output = eta * err_hidden .* h_hidden
        w_input_to_hidden += delta_input_to_hidden
        w_hidden_to_output += delta_hidden_to_output
```

3. 关键代码截图（带注释）

首先是读入的时候对于文本做归一化处理：

```
for j in range(2, len(i)-1):
    if j == 2:
        temp.append(string.atof(i[j])/4.0)
    elif j == 4:
        temp.append(string.atof(i[j])/12.0)
    elif j == 5:
        temp.append(string.atof(i[j])/23.0)
    elif j == 7:
        temp.append(string.atof(i[j])/7.0)
    elif j == 9:
        temp.append(string.atof(i[j])/4.0)
    else:
        temp.append(string.atof(i[j]))
```

然后是我们需要用到的 sigmoid 函数：

```
def sigmoid(m):  #sigmoid函数
    return 1.0 / (1+math.pow(math.e, -m))
```

前向传递过程：

```
h1 = [0 for h in range(0, number_of_hidden)]
for k in range(0, number_of_hidden):
    h1[k] = sigmoid(np.dot(ts[j], w01[k]))  #通过隐藏层
ho = np.dot(h1, w1o)                #得到输出
```

反向传播过程：

```
    for k in range(0, number_of_hidden):
        for kk in range(0, col):
            delta_w01[k][kk] += eta * err_0[kk] * ts[j][kk]
    delta_w1o += eta * np.multiply(err_1, h1)  #累加delta
w01 += delta_w01  #更新输入层到隐藏层的权重矩阵
w1o += delta_w1o  #更新隐藏层到输出层的权重
```

```
err_o = tans[j] - ho   #输出误差
err_1 = [0 for e in range(0, number_of_hidden)]
for k in range(0, number_of_hidden):
    err_1[k] = h1[k] * (1 - h1[k]) * err_o * w1o[k] #隐藏层的误差
```

做完训练之后在验证集上测试：

```
def calc_mse():  #计算误差函数
    global vrow, number_of_hidden, vs, w01, w1o, vans
    mse = 0
    for i in range(0, vrow):
        h1 = [0 for h in range(0, number_of_hidden)]
        for k in range(0, number_of_hidden):
            h1[k] = sigmoid(np.dot(vs[i], w01[k]))  #让每一个文本通过隐藏层
        ho = np.dot(h1, w1o)    #再通过输出层
        mse += (ho - vans[i]) * (ho - vans[i]) / vrow  #计算
    return mse
```

4. 创新点&优化（如果有）

本次试验实现的优化是激活函数的优化，将 sigmoid 函数分别改成了 tanh 函数之后的代码如下：

```
def tanh(m):
    return (1 - math.pow(math.e, -2*m)) / (1 + math.pow(math.e, -2*m))
```

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）

我自己设计了小数据集进行测试。输入文本为[1,0,1]，标签为 1，隐藏层有两个节点，输出层一个节点。学习率为 1。所有的权重均为 1。

则手算结果得到，第一次迭代完隐藏层两个节点净输入为

1*1+0*1+1*1=2

经过 sigmoid 函数之后为 0.88，则输出层的输出为

0.88*1+0.88*1=1.76

误差为 1-1.76=-0.76，则隐藏层两个节点误差为

0.88*（1-0.88）*1*（-0.76）=-0.08

则新的输入层到隐藏层权重应为 1+1*1*（-0.08）=0.92，这是第一个和第三个输入节点连接的权重，而第二个点连接的权重仍为 1。从隐藏层到输出层的两个权重均变为

1+（-0.76）*0.88=0.33

所以最后的输出为 0.33×sigmoid（1.84）*2=0.57

程序运行的结果如下：

```
The output after 1 iteration(s): 1.76159415596
The output after 2 iteration(s): 0.5695461469
[Finished in 0.0s]
```

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）

读入训练集的时候，对于每一个文本都产生一个 0 到 99 的随机数，如果小于 20，就把这个文本放入验证集。测试的时候约有 1600 到 1800 条验证集。经过多次调整参数，得出最小的 mse 如下，第二行为输出 0 时候的 mse：

```
17182.2073575
35984
[Finished in 13.4s]
```

反思：sigmoid 函数在自变量值较大或较小的时候区分度太小，导致更新之后的预测结果都很相近，调参数从 3w 多降到 1w 多，但是并没有解决数量级上的差异，希望下一次 proj 中能够从根本上解决。

四、 思考题

1. 尝试说明下其他激活函数的优缺点。

答： 其他的激活函数还有 tanh 函数，ReLU 函数等。

tanh 函数的优点是相比于 sigmoid 函数，它的收敛速度更快，且由函数图像可得，它的输出以 0 为中心；缺点是无法解决梯度消失问题。

ReLU 函数的优点是收敛速度相比 tanh 更快了一些，同时实现起来更加简单，且梯度消失的问题有所缓解；而缺点是随着训练的进行，可能会出现神经元的权重无法更新的问题。

2. 有什么方法可以实现传递过程中不激活所有节点？

答：在权重矩阵中，以一定的概率将其中一些值变为 0，这就相当于去掉了某些神经元与神经元之间的连接。这种方法可以有效解决过拟合问题。

3. 梯度消失和梯度爆炸是什么？可以怎么解决？

答：在我们计算梯度的时候需要用到链式求导，在连乘的时候如果大部分的乘数小于 1，则最后的结果趋于 0，这是梯度消失；反之如果大部分的乘数大于 1，则结果趋于无穷，这是梯度爆炸。解决方法是设定一个梯度的阈值，一旦超过阈值则直接把阈值赋值给梯度。