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	**15M1/1518**	专业（方向）	**移动互联网**
学号		姓名	**Jw**

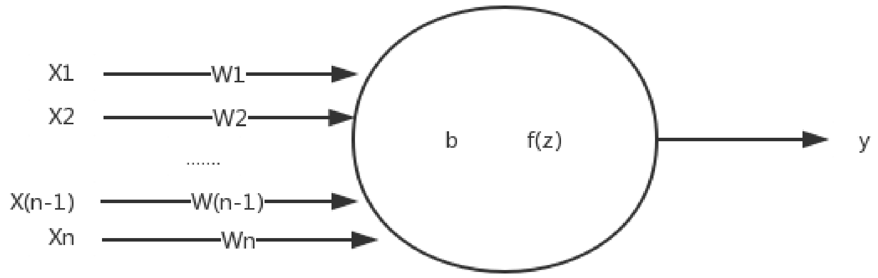
一、 实验题目

BP 神经网络(误差反向传播神经网络) ------ Back propagation nerual network

二、 实验内容

1. 算法原理

神经元模型：学习神经网络首先要知道网络的组成元素----神经元。神经网络是由许多神经元组成的网络系统，一个神经元的组成如下图：



它包括：

1.输入信号 Xi，这些信号代表来自环境的数据或其他神经元的激活

2.一组实值权重 Wi，这些权重的值代表连接强度

3.一个阈值 b

4.一个激活函数 f（z）

5.输出 y

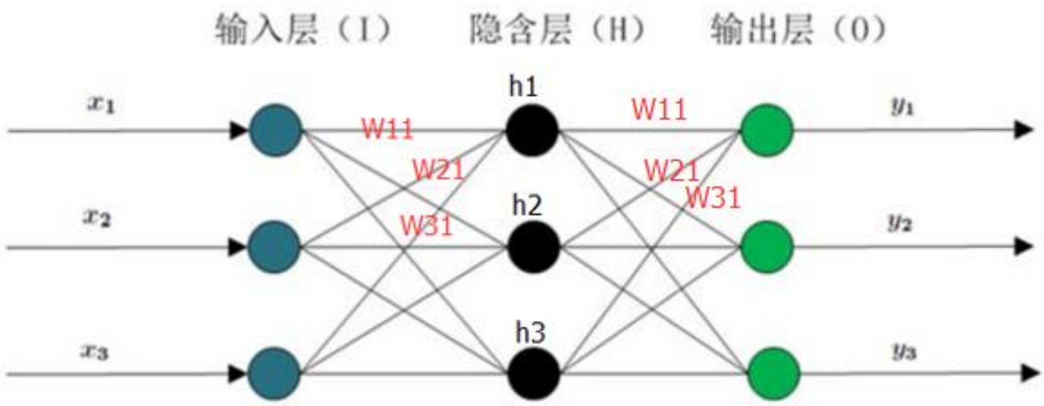
这其中：

$y = f(z)$　　　$z = \sum_{i=1}^{n} w_i x_i + b$

在实际应用中，激活函数 f 通常采用 Sigmoid 函数，将数据映射到(0，1)范围：

$$f(x) = \frac{1}{1+e^{-x}}$$

BP 神经网络：BP 神经网络是一种三层或三层以上的神经网络模型。它的训练算法采用的是误差逆向传播算法(err Back Propagation)。网络模型拓扑结构包括三层，分别是输入层（input layer）、隐藏层（hidden layer）和 输出层（output layer），其中输入层和输出层都只有一层点，但是**隐藏层可以是一层也可以是有很多层**。一个例子如下（注意输入层与隐含层之间的权重 w 和 隐含层与输出层之间的 w 是不一样的）：



BP 神经网络的每一次迭代包括两个部分、前向传播输入以及反向传播误差。也就是先从左向右先将输入传递到输出层产生结果，然后计算输出层结果与真实结果的误差，再将误差逆向传播去修改权重。重复上述过程直到输出误差收敛或者迭代到了一定的次数。

前向传播输入部分：

1. 给定隐藏层或输出层的单元 j，单位 j 的净输入I_j为

   $$I_j = \sum_i W_{ij} O_i + b_j$$

   W_{ij}是从上一层单元 i 到单元 j 的连接权重，O_i是上一层单元 i 的输出，b_j是单元 j 的阈值。

2. 给定单位 j 的净输入I_j，则对隐藏层单位 j 的输出O_j为

   $$O_j = f(I_j)$$

   对输出层单位 j 的输出O_j为

   $$O_j = I_j$$

反向传播误差部分：

我们的目标是最小化输出的误差，为了让误差以最快的速度下降，我们采用梯

度下降法作为学习策略，将误差往回传然后不断去调整权重和阈值，流程如下(推导在后面):

1. 对于输出层中的单元 k，设真实输出为Y_k误差Err_k由下式计算

$$Err_k = Y_k - O_k$$

2. 对于隐藏层单元 j 的误差为:

$$Err_j = \sum_k Err_k W_{jk}$$

3. 权重以及阈值更新公式:

$$W_{jk} = W_{jk} + \ alpha * \ f'(O_j) * Err_k O_j$$

$$b_k = b_k + alpha * f'(O_j) * Err_k$$

其中 alpha 是学习率，是一个(0,1)之间的浮点数，上式是所有情况，要注意的是由于输出层的输出=输入，此时隐藏层到输出层的更新公式中$f'(O_j)$为 1，其他情况就是激活函数的导数。

梯度下降的推导：

我们采用均方误差来衡量，为方便计算，将每一个输出点 k 误差定义为

$$Err_k = Y_k - O_k$$

将均方误差误差定义为

$$\mathrm{E} = \frac{1}{2}\sum_{k=1}^{n} {Err_k}^2$$

我们的目标是要让 E 最小，这个目标通过对权重和阈值的调整来实现。

为了调整权重W_{jk}，我们对 E 求W_{jk}的偏导得:

$$\frac{\partial \mathrm{E}}{\partial W_{jk}} = \frac{\partial \mathrm{E}}{\partial Err_k} * \frac{\partial Err_k}{\partial f} * \frac{\partial \mathrm{f}}{\partial W_{jk}}$$

$$= Err_k * \frac{\partial\left[Y_k - f\left(\sum_j W_{jk} O_j + b_k\right)\right]}{\partial f} * \frac{\partial \mathrm{f}}{\partial W_{jk}}$$

$$= Err_k * \left(-f'\left(\sum_j W_{jk} O_j + b_k\right)\right) * \frac{\partial f\left(\sum_j W_{jk} O_j + b_k\right)}{\partial W_{jk}}$$

$$= -Err_k * f'(O_k) * O_j$$

同理，为了调整阈值b_k，我们对 E 求b_k的偏导得：

$$\frac{\partial \mathrm{E}}{\partial b_k} = \frac{\partial \mathrm{E}}{\partial Err_k} * \frac{\partial Err_k}{\partial f} * \frac{\partial \mathrm{f}}{\partial b_k}$$

$$= Err_k * \frac{\partial\left[Y_k - f\left(\sum_j W_{jk} O_j + b_k\right)\right]}{\partial f} * \frac{\partial \mathrm{f}}{\partial b_k}$$

$$= Err_k * \left(-f'\left(\sum\nolimits_j W_{jk} O_j + b_k\right)\right) * \frac{\partial f\left(\sum_j W_{jk} O_j + b_k\right)}{\partial b_k}$$

$$= -Err_k * f'(O_k) * 1$$

然后权重更新公式就是：

$$W_{jk} = W_{jk} + \ alpha * \ f'\left(O_j\right) * Err_k * O_j$$

$$b_k = b_k + alpha * f'\left(O_j\right) * Err_k$$

上式是针对一个样本的情况，如果有多个样本，这上面两式+后面的部分为所有样本求出的值的均值。

2. 伪代码

训练部分：

Input： 训练集 D={(xk, yk) | 1<=k<=n}
Output： 权重矩阵 W1、W2，阈值矩阵 B1，B2
Initialize： W 和 B 全随机，设置最大迭代次数 maxStep，步长alpha
for t = 0,1…maxStep
  hiddenInput = W1*InputData + B1
  finalInput = hiddenOuput = f（hiddenInput）
  finalOuput = W2 * hiddenOutput + B2
  errK = realOuput – finalOutput
  errj = W2 * errK * f'(hiddenInput)
  W2 = W2 + alpha * errK*hiddenOutpu/Nt
  B2 = B2 + alpha * errK/N
  W1=W1+alpha*errj*InputData/N
  B1=B1+alpha*errj/N
end for

训练部分：

Input： 测试集 D={(xk, yk) | 1<=k<=n}，训练好的 W1,W2，B1，B2
Output： 预测结果 finalOutput
hiddenInput = W1*InputData + B1

finalInput = hiddenOuput = f（hiddenInput）

*finalOuput = W2 * hiddenOutput + B2*

3. 关键代码截图（带注释）

1.训练函数

```
def train(InputData, OutputData):
    inputDim = 11
    hiddenDim = 100
    outputDim = 1
    maxStep = 3000
    alpha = 0.0001
    N = InputData.shape[1]

    w1 = 0.5*np.random.rand(hiddenDim, inputDim) - 0.1  # 输入层到隐藏层的
    b1 = 0.5*np.random.rand(hiddenDim, 1) - 0.1
    w2 = 0.5*np.random.rand(outputDim, hiddenDim) - 0.1  # 隐藏层到输出层的
    b2 = 0.5*np.random.rand(outputDim, 1) - 0.1

    TrainLoss, w1List, w2List, b1List, b2List = [], [], [], [], []
    w1List.append(w1), b1List.append(b1)
    w2List.append(w2), b2List.append(b2)
    for i in range(maxStep):
        sys.stdout.write("\rPercent: %f %%" % ((i+1)/maxStep * 100))
        sys.stdout.flush()
        # 前向传播 Forward pass
        hiddenInput = np.dot(w1, InputData) + b1  # hiddenDim*N
        # hiddenOuput = sigmoid(hiddenInput)  # hiddenDim*N
        hiddenOuput = tanh(hiddenInput)
        finalOutput = np.dot(w2, hiddenOuput) + b2  # outputDim*N

        # 反向传播 Backward pass
        err2 = (OutputData - finalOutput)  # 输出层的误差 outputDim*N
        gradient2 = err2  # 输出层的误差梯度  ouputDim*N
        err1 = np.dot(w2.T, gradient2)  # 隐藏层误差 hiddenDim*outputDim
        # gradient1 = err1 * hiddenOuput * (1 - hiddenOuput)  # 隐藏层误差梯度 hiddenDim*outputDim
        gradient1 = err1 * (1 - hiddenOuput**2)

        # 更新权重
        w2 = w2 + alpha * np.dot(gradient2, hiddenOuput.T)/N
        b2 = b2 + alpha * np.dot(gradient2, np.ones((N, 1)))/N
        w1 = w1 + alpha * np.dot(gradient1, InputData.T)/N
        b1 = b1 + alpha * np.dot(gradient1, np.ones((N, 1)))/N

        MSE = np.sum(err2**2)/N/2
        TrainLoss.append(MSE)

        w1List.append(w1), b1List.append(b1)
        w2List.append(w2), b2List.append(b2)

    return TrainLoss, w1List, w2List, b1List, b2List
```

2.预测函数、输入需要预测的矩阵以及训练好的 W1、W2、B1、B2，输出标签 Y

```python
def predict(InputData, w1, w2, b1, b2):
    hiddenInput = np.dot(w1, InputData) + b1  # hiddenDim*N
    hiddenOuput = sigmoid(hiddenInput)  # hiddenDim*N
    # hiddenOuput = tanh(hiddenInput)
    finalOutput = np.dot(w2, hiddenOuput) + b2  # outputDim*N
    return finalOutput
```

4. **创新点&优化（如果有）**

1. python 向量化运算

2. 激活函数除了用了默认的 sigmoid，还尝试了双曲正切 tanh

```python
def tanh(x):
    return (np.exp(x*2) - 1) / (np.exp(x*2) + 1)
```

```python
    hiddenInput = np.dot(w1, InputData) + b1  # hiddenDim*N
    # hiddenOuput = sigmoid(hiddenInput)  # hiddenDim*N
    hiddenOuput = tanh(hiddenInput)
```

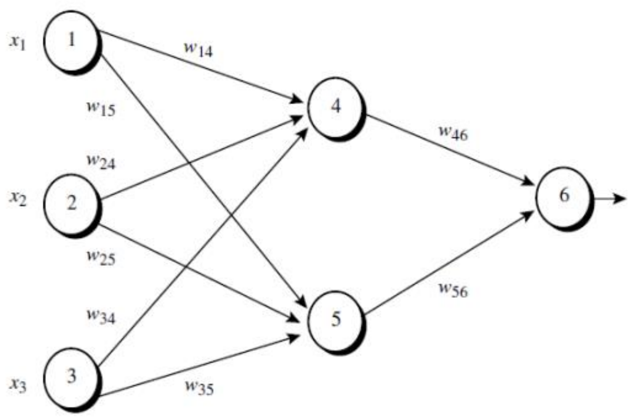
3.对数据使用了 minmax 归一化

```python
def normalize(M):
    if M.ndim > 1:
        Min = np.array([M.min(axis=1).T.tolist()]).T
        Max = np.array([M.max(axis=1).T.tolist()]).T
    else:
        Min = np.array([M.min().T.tolist()]).T
        Max = np.array([M.max().T.tolist()]).T
    M = (Max-Min)*(M-Min)/(Max-Min + 0.000001)
    return Min, Max, 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）

一个简单的小数据集：一个训练元组（X=1,0,1），标签为 1



输入数据如下：

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

使用 sigmoid 作为激活函数，迭代一次，学习率为 0.9。

手动计算结果如下：

前向传播过程：

Unit j	Net input I_J	Output Oj
4	0.2+0-0.5-0.4 = -0.7	1/（$1+e^{0.7}$）=0.332
5	-0.3 + 0 + 0.2 + 0.2 = 0.1	1/（$1+e\text{-}^{0.1}$）=0.525
6	(-0.3)(0.332)-(0.2*0.525)+0.1=-0.105	-0.105

反向传播过程：

Unit j	Err j
6	1 - （- 0.105）=1.105
5	0.525*（1-0.525）*（1.105）*（-0.2）= -0.055
4	0.332*（1-0.332）*（1.105）*（-0.3）= -0.0735

权重及阈值更新（用 b 代替θ）

W46	-0.3 + 0.9 * 1.105 * 0.332 = 0.030
W56	-0.2 + 0.9 * 1.105 * 0.525 = 0.322
W14	0.2 + 0.9 * (-0.0735) * 1 = 0.13385
W15	-0.3+0.9*（-0.055）*1 = -0.3495
W24	0.4+0.9*（-0.0735）*0=0.4
W25	0.1+0.9*（-0.055）*0 = -0.1
W34	-0.5+0.9*（-0.0735）*1= -0.56615
W35	0.2 + 0.9*（-0.055）*1 = 0.1505
B6	0.1+0.9*1.105 = 1.0945
B5	0.2+0.9*（-0.055）=1.505
B4	-0.4 + 0.9*（-0.0735）=- - 0.46615

程序计算结果如下，直接看最后的权重和阈值结果：

```
问题    输出    调试控制台    终端
w1:
[[ 0.13387953 -0.34958022]
 [ 0.4         0.1       ]
 [-0.56612047  0.15041978]]
w2:
[ 0.02984974  0.32187423]
b1:
[-0.46612047  0.15041978]
b2:
[ 1.09408556]
```

如图 W1 的 第 i 行分别对应 Wi4 和 Wi5 (1<=i<=3)

W2 的两个数字分别是 W46 和 W56

b1 的两个数字分别是 b4、b5

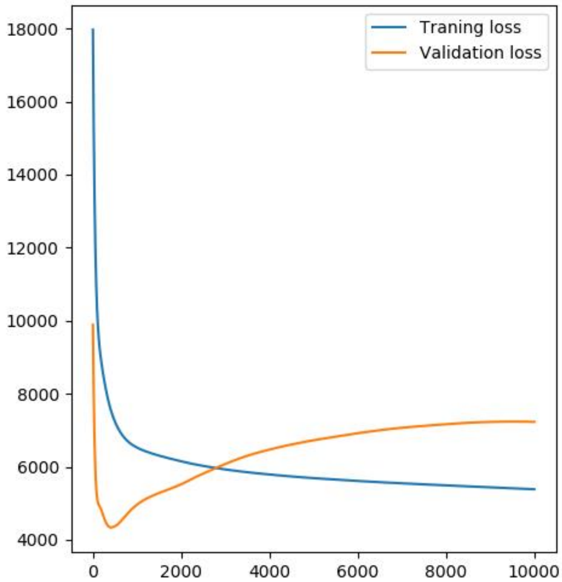
b2 的数字 是 b6

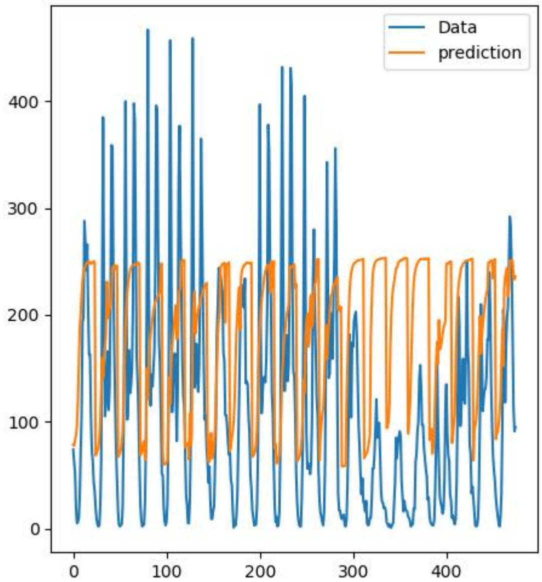
对比可知，在计算误差范围内，与手动计算结果基本一致。

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）

我将训练集数据的后 20 天即 2011 年 12 月 11 号到 2011 年 12 月 30 号的数据作为验证集（共 475 个样本），并且去掉了训练数据中的索引列、日期列、年列，因为个人觉得没什么用，前两个都是一个索引作用，年的话训练集都是 11 年也没有用。

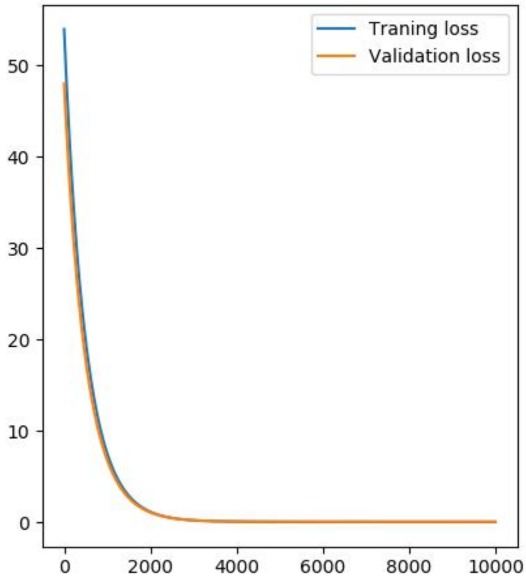
采用 sigmoid 作为激活函数、没有作归一化处理、学习率固定为 0.0001、100 个隐藏层结点且迭代 1 万次的结果如下：

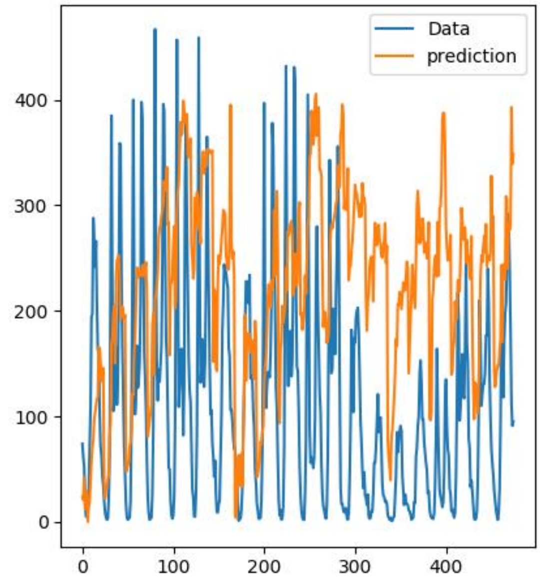




左边 Loss function 的对比图 可以看到训练集的 loss 仍在下降，但是验证集的 loss 却已经触底反弹，说明这个时候已经过拟合了，从右边输出结果对比也可以看到预测的结果不好。造成这种现象的原因可能是学习率设置太大，数据没有归一化发散性比较强，我观察了一下 12 月的真实结果都比较小，而训练集有很多挺大的结果，这就很大几率造成训练集训练出来的模型在验证集上表现很差。

采用双曲正切函数 tanh 作为激活函数，并且使用 minmax 进行归一化处理，学习率固定为 0.00002，100 个隐藏层结点且迭代 1 万次的结果如下：



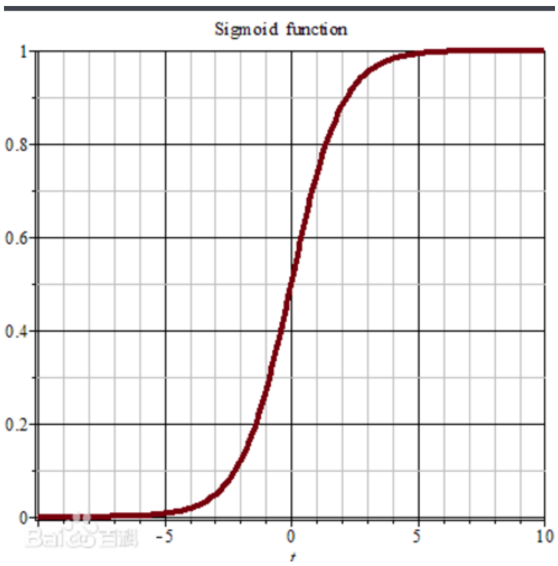


可以发现左边 loss 曲线都比较好，到最后都收敛了，纵坐标值比较小是因为做了归一化。右边输出结果对比也比之前好很多，在最后样本数 300-400 这个区间的结果差距比较大，原因也可能是因为 12 月真实标签数据较小，而训练集的标签数据大数比较多，模型不能很好地符合验证集。

四、 思考题

1.尝试说明下其他激活函数的优缺点

答：1. **Sigmoid 函数**：



优点：将数据映射到了（0,1）区间，数据不容易发散；

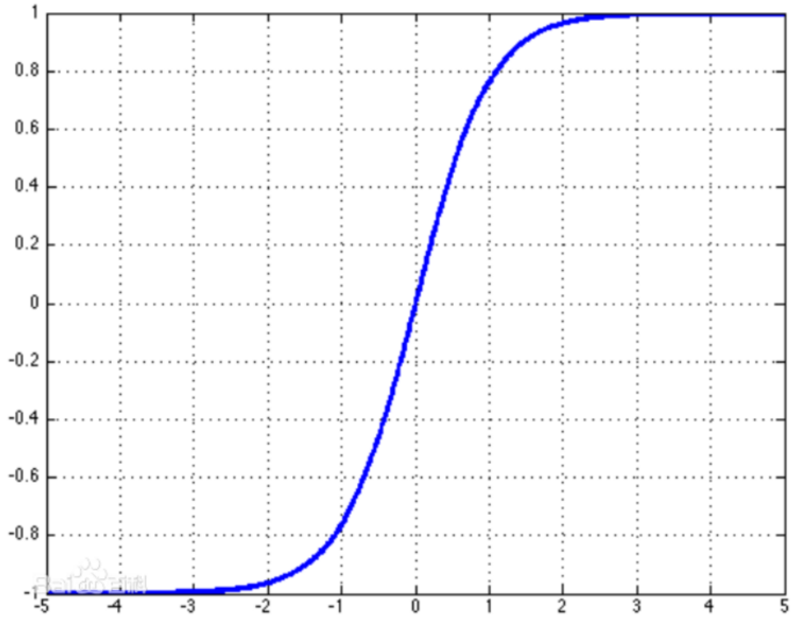
让数据的传递有非线性的形式，这样才能逼近目标函数；

是凸函数，有最优值，可导，可以采用梯度下降的优化方法

缺点：当输入过大或者过小的时候，梯度接近 0，出现过饱和现象，因此在反向传播时，这个局部梯度会与整个代价函数关于该神经元输出的梯度相乘，结果也会接近为 0；

Sigmoid 函数不是关于原点中心对称的，这样会导致后面网络层的输入也不是以 0 为均值的，后果就是假如输入数据为正，那么反向传播时 W 就会全为正或者全为负，就会导致梯度下降出现锯齿形波动。

2.tanh 函数：



优点：将数据映射到（-1,1），由图可以看出它其实是 sigmoid 的放大版，除了拥有和 sigmoid 函数一样的优点外，它是以原点为中心的。

缺点：同样存在过饱和问题使得梯度消失。

3. **ReLU 函数：**

F(x) = max(0,x)

优点：在梯度下降上比前两者有更快的收敛速度，因为它是线性非饱和的；

计算比前两者要简单快速；

缺点：大的梯度流过一个 ReLU 神经元并且更新了参数之后，这个神经元可能再也不会对任何数据有激活现象，这时这个神经元的梯度永远会是 0。一般在学习率设置较大的时候会出现这种情况。

4. **Leaky-ReLU 函数：**

F(x) =x(x>=0)　　F(x) = ax(x<0)　a 是很小的数

优点：保留了很小的负数梯度值，解决 ReLu 容易大量神经元死亡的问题。

缺点：效果不太稳定

2.有什么方法可以实现传递过程中不激活所有节点？

答：可以使用 dropout 技术，就是在每次训练的时候，随机临时删除一半神经元，这样做的好处是能提高网络的泛化能力，防止过拟合。

3. 梯度消失和梯度爆炸是什么？可以怎么解决？

答：我们知道在反向传播的过程中，梯度是会逐层相乘往前传播，那么假如每一层的梯度都是一个比 1 小的很小的数，那么经过足够多层的传播之后梯度会变成 0，例如如果每一层的梯度都是 0.9，那么反向传 n 层会有 $\lim_{n\to\infty}(0.9)^n = 0$，这样权重基本就不会再更新，这就是梯度消失现象。

梯度爆炸刚好相反，由于初始权重过大，这样误差大梯度也大，回传时候前面层的变换就会比后面层的变化更快，权重会越来越大。

解决办法：1. 初始化的权重不能过大也不要过小

2. 学习率设置要恰当，可以采用动态学习率

3. 采用 ReLu 函数代替 sigmoid 或 tanh

五、 一些补充

1. 关于隐藏层点数的确定，隐藏层点数是自己定的，但是随意定好像不是太靠谱，一般有几条经验公式：

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

$$m = \log_2 n$$

$$m = 2n + 1$$

$$m = \sqrt{nl}$$

其中 m 是隐藏层节点数、n 是输入层节点数、l 是输出层节点数、α是一个 1-10 之间的常数。这次实验的时候前三条试过都不太好，最后一条我也结合了一下逐步试验发调了一下，最终用了 100 个点。

2.学习率最好应该设为自动调整，增加 BPNN 的稳定性。

3.我们不能一味地追求训练集误差达到最小，这样很容易过拟合，比如我实验结果展示的 sigmoid 那个结果，就是这样的情况，避免过拟合一般要试出最佳训练次数、然后训练数据划分训练集和验证集，用 K 折验证会更稳定。

4.BP 网络的可解释性不强，很多参数、网络结构得凭经验去确定。