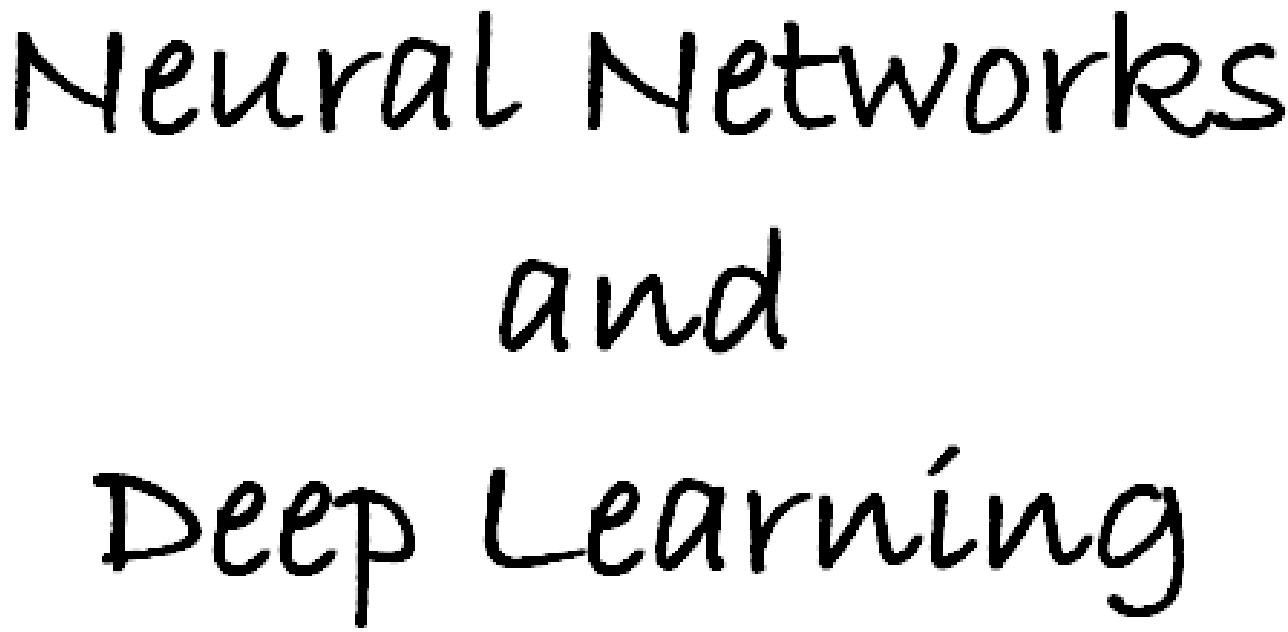


Coursera吴恩达《神经网络与深度学习》课程笔记（4）-- 浅层神经网络

红色石头 · 2 个月前

> **我的CSDN博客地址：红色石头的专栏**
> **我的知乎主页：红色石头**
> **我的知乎专栏：红色石头的机器学习之路**
> **欢迎大家关注我！共同学习，共同进步！**

上节课我们主要介绍了向量化、矩阵计算的方法和python编程的相关技巧。并以逻辑回归为例，将其算法流程包括梯度下降转换为向量化的形式，从而大大提高了程序运算速度。本节课我们将从浅层神经网络入手，开始真正的神经网络模型的学习。

1. Neural Networks Overview

首先，我们从整体结构上来大致看一下神经网络模型。

前面的课程中，我们已经使用计算图的方式介绍了逻辑回归梯度下降算法的正向传播和反向传播两个过程。如下图所示。神经网络的结构与逻辑回归类似，只是神经网络的层数比逻辑回归多一层，多出来的中间那层称为隐藏层或中间层。这样从计算上来说，神经网络的正向传播和反向传播过程只是比逻辑回归多了一次重复的计算。正向传播过程分成两层，第一层是输入层到隐藏层，用上标[1]来表示：

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

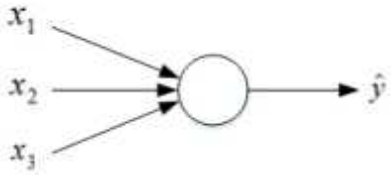
第二层是隐藏层到输出层，用上标[2]来表示：

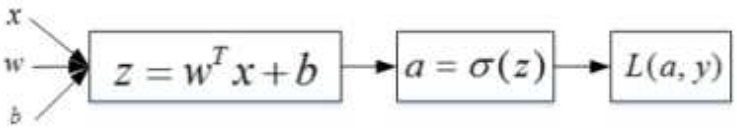
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

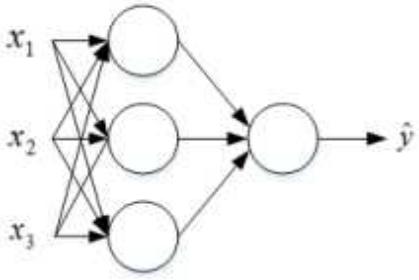
$$a^{[2]} = \sigma(z^{[2]})$$

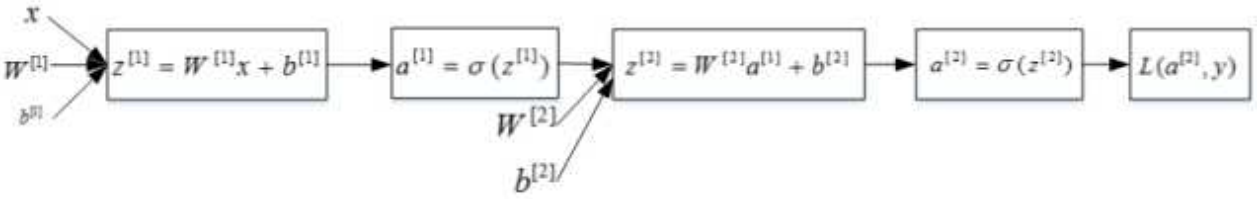
在写法上值得注意的是，方括号上标[i]表示当前所处的层数；圆括号上标(i)表示第i个样本。

同样，反向传播过程也分成两层。第一层是输出层到隐藏层，第二层是隐藏层到输入层。其细节部分我们之后再来讨论。



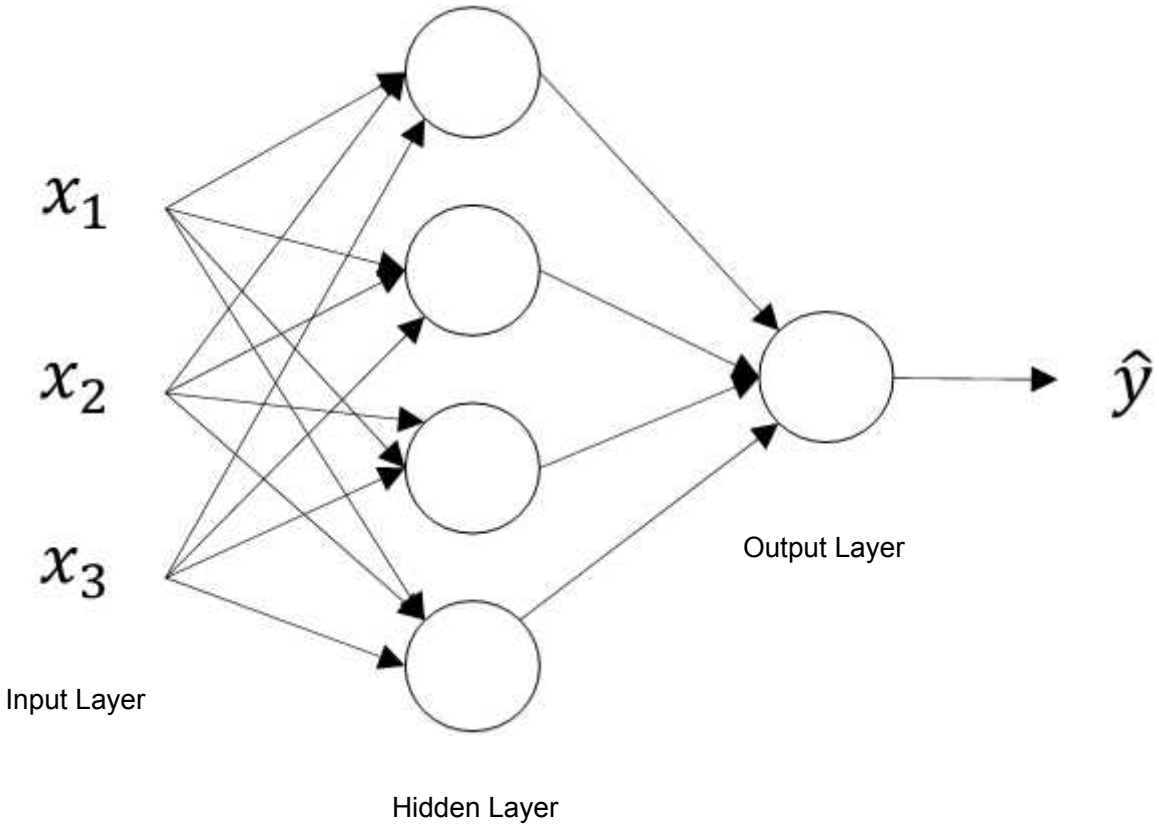






2. Neural Network Representation

下面我们以图示的方式来介绍单隐藏层的神经网络结构。如下图所示，单隐藏层神经网络就是典型的浅层（shallow）神经网络。



结构上，从左到右，可以分成三层：输入层（Input layer），隐藏层（Hidden layer）和输出层（Output layer）。输入层和输出层，顾名思义，对应着训练样本的输入和输出，很好理解。隐藏层是抽象的非线性的中间层，这也是其被命名为隐藏层的原因。

在写法上，我们通常把输入矩阵X记为 $a^{[0]}$ ，把隐藏层输出记为 $a^{[1]}$ ，上标从0开始。用下标表示第几个神经元，注意下标从1开始。例如 $a_1^{[1]}$ 表示隐藏层第1个神经元， $a_2^{[1]}$ 表示隐藏层第2个神经元,，等等。这样，隐藏层有4个神经元就可以将其输出 $a^{[1]}$ 写成矩阵的形式：

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

最后，相应的输出层记为 $a^{[2]}$ ，即 $\hat{y}$ 。这种单隐藏层神经网络也被称为两层神经网络（2 layer NN）。之所以叫两层神经网络是因为，通常我们只会计算隐藏层输出和输出层的输出，输入层是不用计算的。这也是我们把输入层层数上标记为0的原因（ $a^{[0]}$ ）。

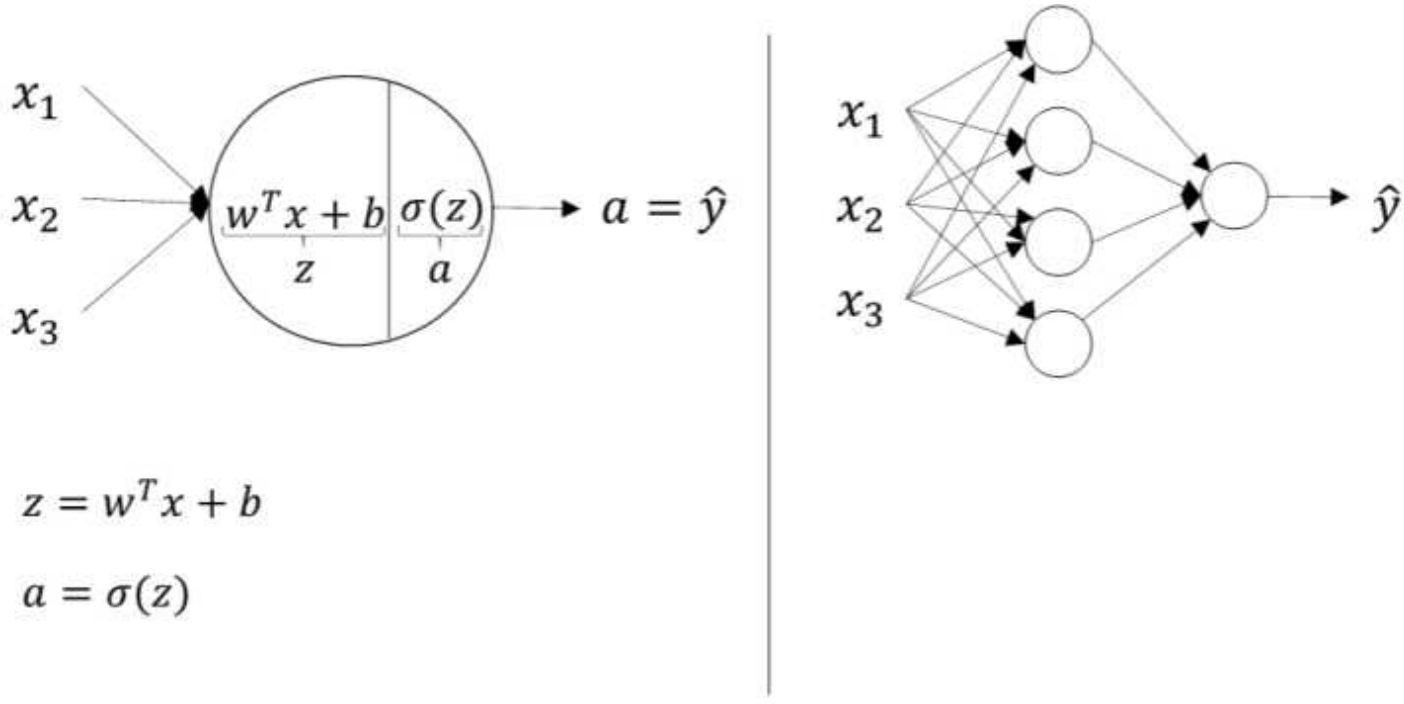
关于隐藏层对应的权重 $W^{[1]}$ 和常数项 $b^{[1]}$，$W^{[1]}$ 的维度是（4,3）。这里的4对应着隐藏层神经元个数，3对应着输入层x特征向量包含元素个数。常数项 $b^{[1]}$ 的维度是（4,1），这里的4同样对应着隐藏层神经元个数。关于输出层对应的权重 $W^{[2]}$ 和常数项 $b^{[2]}$，$W^{[2]}$ 的维度是（1,4），这里的1对应着输出层神经元个数，4对应着输出层神经元个数。常数项 $b^{[2]}$ 的维度是（1,1），因为输出只有一个神经元。总结一下，第i层的权重 $W^{[i]}$ 维度的行等于i层神经元的个数，列等于i-1层神经元的个数；第i层常数项 $b^{[i]}$ 维度的行等于i层神经元的个数，列始终为1。

3. Computing a Neural Network's Output

接下来我们开始详细推导神经网络的计算过程。回顾一下，我们前面讲过两层神经网络可以看成是逻辑回归再重复计算一次。如下图所示，逻辑回归的正向计算可以分解成计算z和a的两部分：

$$z = w^T x + b$$

$$a = \sigma(z)$$



对于两层神经网络，从输入层到隐藏层对应一次逻辑回归运算；从隐藏层到输出层对应一次逻辑回归运算。每层计算时，要注意对应的上标和下标，一般我们记上标方括号表示layer，下标表示第几个神经元。例如 $a_i^{[l]}$ 表示第l层的第i个神经元。注意，i从1开始，l从0开始。

下面，我们将从输入层到输出层的计算公式列出来：

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

然后，从隐藏层到输出层的计算公式为：

$$z_1^{[2]} = w_1^{[2]T} a^{[1]} + b_1^{[2]},\ a_1^{[2]} = \sigma(z_1^{[2]})$$

其中 $a^{[1]}$ 为：

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

上述每个节点的计算都对应着一次逻辑运算的过程，分别由计算z和a两部分组成。

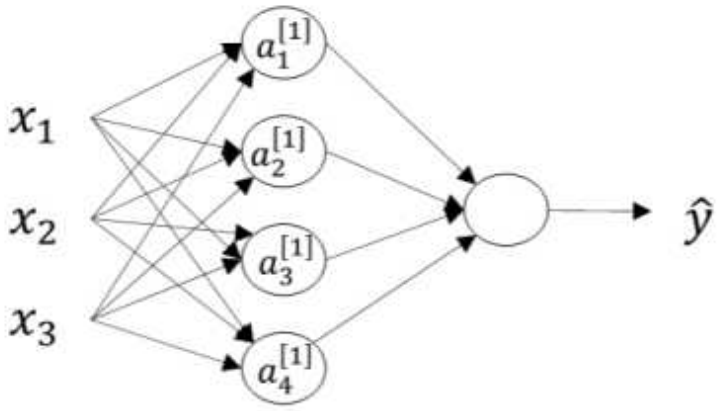
为了提高程序运算速度，我们引入向量化和矩阵运算的思想，将上述表达式转换成矩阵运算的形式：

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$



Vectorizing single example:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

之前也介绍过，这里顺便提一下， $W^{[1]}$ 的维度是（4,3），$b^{[1]}$ 的维度是（4,1），$W^{[2]}$ 的维度是（1,4），$b^{[2]}$ 的维度是（1,1）。这点需要特别注意。

4. Vectorizing across multiple examples

上一部分我们只是介绍了单个样本的神经网络正向传播矩阵运算过程。而对于m个训练样本，我们也可以使用矩阵相乘的形式来提高计算效率。而且它的形式与上一部分单个样本的矩阵运算十分相似，比较简单。

之前我们也介绍过，在书写标记上用上标(i)表示第i个样本，例如 $x^{(i)}$，$z^{(i)}$，$a^{[2](i)}$。对于每个样本i，可以使用for循环来求解其正向输出：

$$for\ i = 1\ to\ m:$$

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]} \qquad a^{[2](i)} = \sigma(z^{[2](i)})$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

不使用for循环，利用矩阵运算的思想，输入矩阵X的维度为（n_x,m）。这样，我们可以把上面的for循环写成矩阵运算的形式：

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

其中， $Z^{[1]}$ 的维度是（4,m），4是隐藏层神经元的个数； $A^{[1]}$ 的维度与 $Z^{[1]}$ 相同； $Z^{[2]}$ 和 $A^{[2]}$ 的维度均为（1,m）。对上面这四个矩阵来说，均可以这样来理解：行表示神经元个数，列表示样本数目m。

5. Explanation for Vectorized Implementation

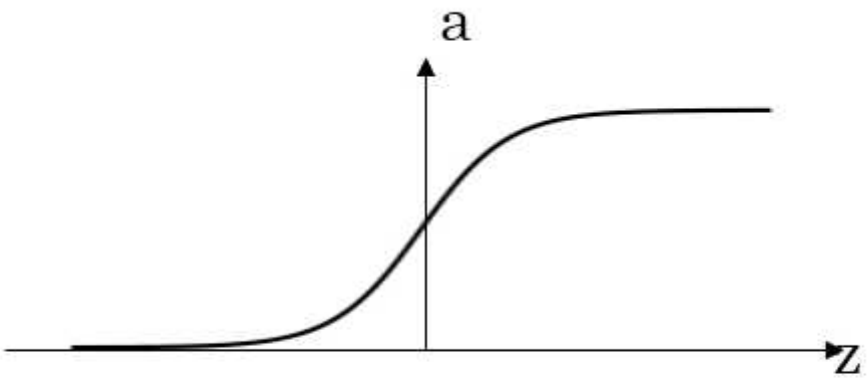
这部分Andrew用图示的方式解释了m个样本的神经网络矩阵运算过程。其实内容比较简单，只要记住上述四个矩阵的行表示神经元个数，列表示样本数目m就行了。

值得注意的是输入矩阵X也可以写成 $A^{[0]}$ 。

6. Activation functions

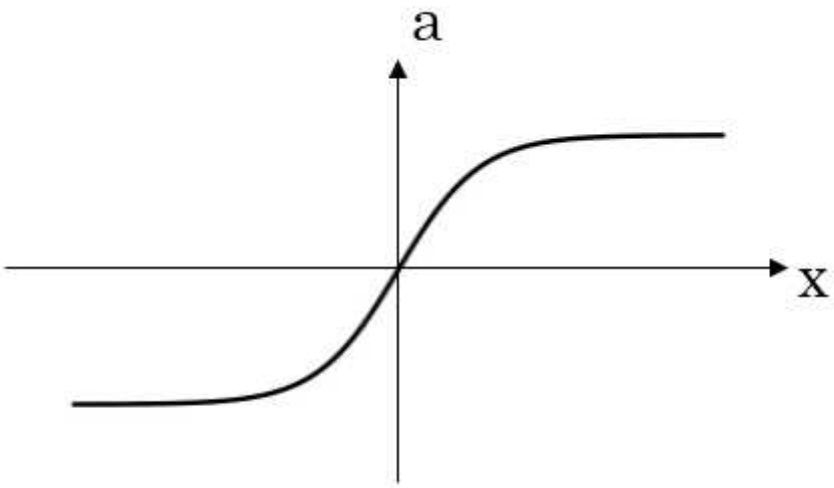
神经网络隐藏层和输出层都需要激活函数（activation function），在之前的课程中我们都默认使用Sigmoid函数 $\sigma(x)$ 作为激活函数。其实，还有其它激活函数可供使用，不同的激活函数有各自的优点。下面我们就来介绍几个不同的激活函数 $g(x)$ 。

- sigmoid函数



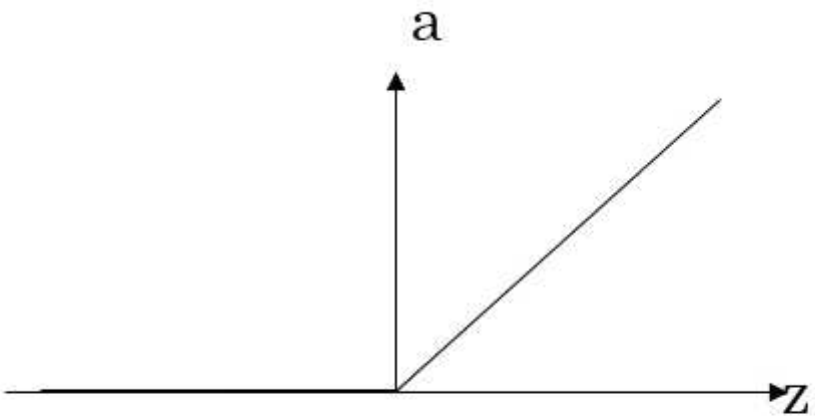
$$\text{sigmoid:} \quad a = \frac{1}{1 + e^{-z}}$$

- tanh函数



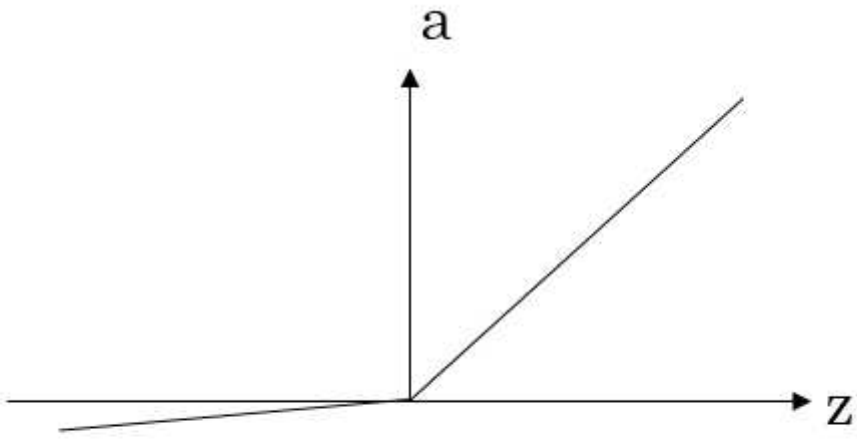
$$\text{tanh:} \quad a = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

- ReLU函数



$$\text{ReLU:} \quad a = \max(0, z)$$

- Leaky ReLU函数



$$\text{Leaky ReLU:} \quad a = \max(0.01z, z)$$

如上图所示，不同激活函数形状不同，a的取值范围也有差异。

如何选择合适的激活函数呢？首先我们来比较sigmoid函数和tanh函数。对于隐藏层的激活函数，一般来说，tanh函数要比sigmoid函数表现更好一些。因为tanh函数的取值范围在[-1,+1]之间，隐藏层的输出被限定在[-1,+1]之间，可以看成是在0值附近分布，均值为0。这样从隐藏层到输出层，数据起到了归一化（均值为0）的效果。因此，隐藏层的激活函数，tanh比sigmoid更好一些。而对于输出层的激活函数，因为二分类问题的输出取值为{0,+1}，所以一般会选择sigmoid作为激活函数。

观察sigmoid函数和tanh函数，我们发现有这样一个问题，就是当|z|很大的时候，激活函数的斜率（梯度）很小。因此，在这个区域内，梯度下降算法会运行得比较慢。在实际应用中，应尽量避免使z落在这个区域，使|z|尽可能限定在零值附近，从而提高梯度下降算法运算速度。

为了弥补sigmoid函数和tanh函数的这个缺陷，就出现了ReLU激活函数。ReLU激活函数在z大于零时梯度始终为1；在z小于零时梯度始终为0；z等于零时的梯度可以当成1也可以当成0，实际应用中并不影响。对于隐藏层，选择ReLU作为激活函数能够保证z大于零时梯度始终为1，从而提高神经网络梯度下降算法运算速度。但当z小于零时，存在梯度为0的缺点，实际应用中，这个缺点影响不是很大。为了弥补这个缺点，出现了Leaky ReLU激活函数，能够保证z小于零是梯度不为0。

最后总结一下，如果是分类问题，输出层的激活函数一般会选择sigmoid函数。但是隐藏层的激活函数通常不会选择sigmoid函数，tanh函数的表现会比sigmoid函数好一些。实际应用中，通常会会选择使用ReLU或者Leaky ReLU函数，保证梯度下降速度不会太小。其实，具体选择哪个函数作为激活函数没有一个固定的准确的答案，应该要根据具体实际问题进行验证（validation）。

7. Why do you need non-linear activation functions

我们知道上一部分讲的四种激活函数都是非线性（non-linear）的。那是否可以使用线性激活函数呢？答案是不行！下面我们就来进行简要的解释和说明。

假设所有的激活函数都是线性的，为了简化计算，我们直接令激活函数 $g(z)=z$ ，即 $a=z$ 。那么，浅层神经网络的各层输出为：

$$z^{[1]}=W^{[1]}x+b^{[1]}$$

$$a^{[1]}=z^{[1]}$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = z^{[2]}$$

我们对上式中 $a^{[2]}$ 进行化简计算：

$$\begin{aligned} a^{[2]} = z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} = W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} \\ &= (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]}) = W'x + b' \end{aligned}$$

经过推导我们发现 $a^{[2]}$ 仍是输入变量x的线性组合。这表明，使用神经网络与直接使用线性模型的效果并没有什么两样。即便是包含多层隐藏层的神经网络，如果使用线性函数作为激活函数，最终的输出仍然是输入x的线性模型。这样的话神经网络就没有任何作用了。因此，隐藏层的激活函数必须要是非线性的。

另外，如果所有的隐藏层全部使用线性激活函数，只有输出层使用非线性激活函数，那么整个神经网络的结构就类似于一个简单的逻辑回归模型，而失去了神经网络模型本身的优势和价值。

值得一提的是，如果是预测问题而不是分类问题，输出y是连续的情况下，输出层的激活函数可以使用线性函数。如果输出y恒为正值，则也可以使用ReLU激活函数，具体情况，具体分析。

8. Derivatives of activation functions

在梯度下降反向计算过程中少不了计算激活函数的导数即梯度。

我们先来看一下sigmoid函数的导数：

$$g(z) = \frac{1}{1 + e^{(-z)}}$$

g'(z) = g(z)(1 - g(z)) = a(1 - a)

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z)) = a(1 - a)$$

$g'(z) = \frac{d}{dz} g(z) = g(z)(1 - g(z)) = a(1 - a)$

对于tanh函数的导数:

$$g(z) = \frac{e^{(z)} - e^{(-z)}}{e^{(z)} + e^{(-z)}}$$

$$g'(z) = \frac{d}{dz} g(z) = 1 - (g(z))^2 = 1 - a^2$$

对于ReLU函数的导数:

$$g(z) = max(0, z)$$

$$g'(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

对于Leaky ReLU函数:

$$g(z) = max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

9. Gradient descent for neural networks

接下来看一下在神经网络中如何进行梯度计算。

仍然是浅层神经网络，包含的参数为 $W^{[1]}$， $b^{[1]}$， $W^{[2]}$， $b^{[2]}$。令输入层的特征向量个数 $n_x = n^{[0]}$，隐藏层神经元个数为 $n^{[1]}$，输出层神经元个数为 $n^{[2]} = 1$。则 $W^{[1]}$ 的维度为（ $n^{[1]}$，$n^{[0]}$），$b^{[1]}$ 的维度为（ $n^{[1]}$,1），$W^{[2]}$ 的维度为（ $n^{[2]}$，$n^{[1]}$），$b^{[2]}$ 的维度为（ $n^{[2]}$,1）。

该神经网络正向传播过程为：

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g(Z^{[2]})$$

其中， $g(\cdot)$ 表示激活函数。

反向传播是计算导数（梯度）的过程，这里先列出来Cost function对各个参数的梯度：

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdim = True)$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

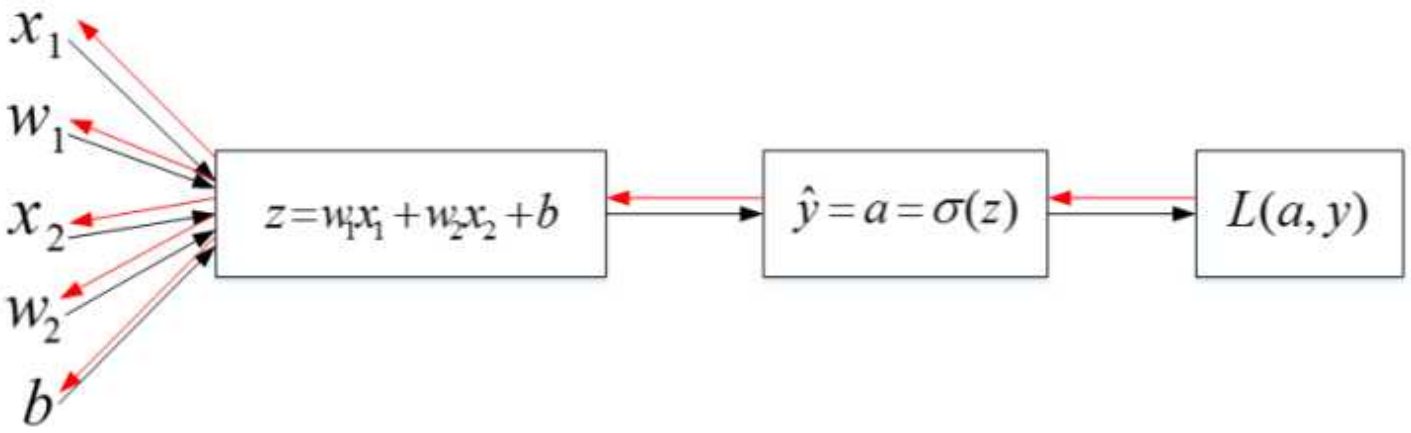
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdim = True)$$

反向传播的具体推导过程我们下一部分再进行详细说明。

10. Backpropagation intuition(optional)

我们仍然使用计算图的方式来推导神经网络反向传播过程。记得之前介绍逻辑回归时，我们就引入了计算图来推导正向传播和反向传播，其过程如下图所示：

Logistic regression gradients



由于多了一个隐藏层，神经网络的计算图要比逻辑回归的复杂一些，如下图所示。对于单个训练样本，正向过程很容易，反向过程可以根据梯度计算方法逐一推导。

$$dz^{[2]} = a^{[2]} - y$$

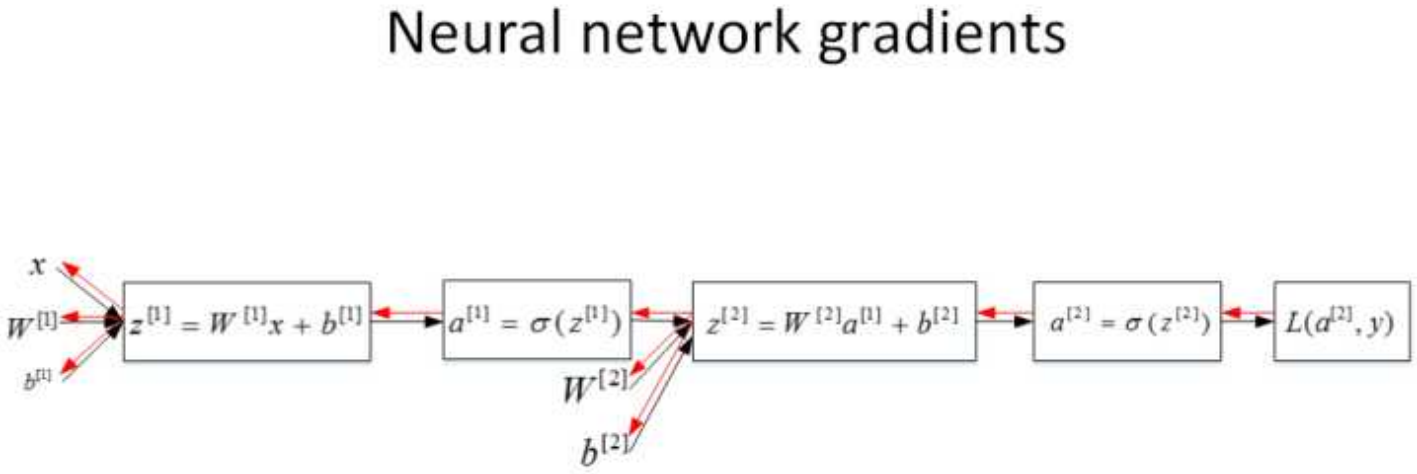
$$dW^{[2]} = dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}} = dz^{[2]} \cdot 1 = dz^{[2]}$$

$$dz^{[1]} = dz^{[2]} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} = W^{[2]T} dz^{[2]} * g'(z^{[1]})$$

$$dW^{[1]} = dz^{[1]} \cdot \frac{\partial z^{[1]}}{\partial W^{[1]}} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}} = dz^{[1]} \cdot 1 = dz^{[1]}$$



总结一下，浅层神经网络（包含一个隐藏层），m个训练样本的正向传播过程和反向传播过程分别包含了6个表达式，其向量化矩阵形式如下图所示：

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]}a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]}A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]'}(\mathrm{z}^{[1]})$	$dZ^{[1]} = W^{[2]T}dZ^{[2]} * g^{[1]'}(\mathrm{Z}^{[1]})$
$dW^{[1]} = dz^{[1]}x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]}X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

11. Random Initialization

神经网络模型中的参数权重W是不能全部初始化为零的，接下来我们分析一下原因。

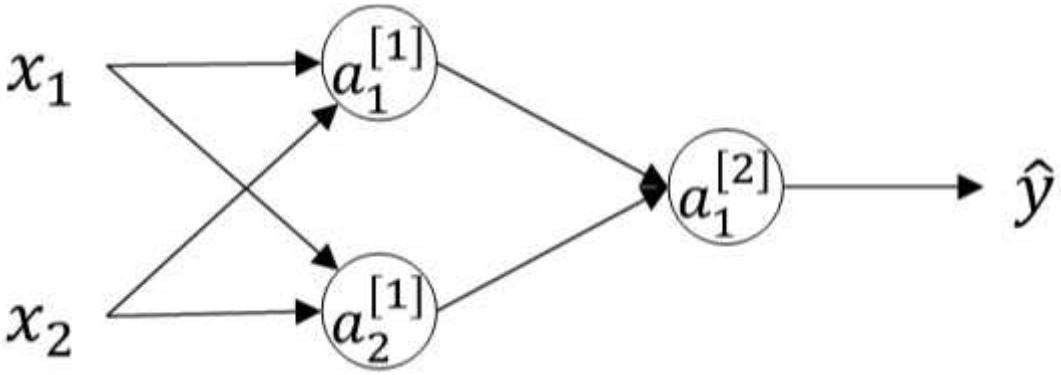
举个简单的例子，一个浅层神经网络包含两个输入，隐藏层包含两个神经元。如果权重 $W^{[1]}$ 和 $W^{[2]}$ 都初始化为零，即：

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$W^{[2]} = [0 \quad 0]$$

这样使得隐藏层第一个神经元的输出等于第二个神经元的输出，即 $a_1^{[1]} = a_2^{[1]}$ 。经过推导得到 $dz_1^{[1]} = dz_2^{[1]}$ ，以及 $dW_1^{[1]} = dW_2^{[1]}$ 。因此，这样的结果是隐藏层两个神经元对应的权重行向量 $W^{[1]}$ 和 $W^{[1]}$ 每次迭代更新都会得到完全相同的结果，$W^{[1]}$ 始终等于 $W^{[1]}$ ，完全对称，这样隐藏层

$w_1^{[1]}$ 和 $w_2^{[1]}$ 每次迭代更新都会得到完全相同的结果，$w_1^{[1]}$ 始终等于 $w_2^{[1]}$，完全对称。这样隐藏层设置多个神经元就没有任何意义了。值得一提的是，参数b可以全部初始化为零，并不会影响神经网络训练效果。



我们把这种权重W全部初始化为零带来的问题称为symmetry breaking problem。解决方法也很简单，就是将W进行随机初始化（b可初始化为零）。python里可以使用如下语句进行W和b的初始化：

```
W_1 = np.random.randn((2,2))*0.01
b_1 = np.zero((2,1))
W_2 = np.random.randn((1,2))*0.01
b_2 = 0
```

这里我们将 $W_1^{[1]}$ 和 $W_2^{[1]}$ 乘以0.01的目的是尽量使得权重W初始化比较小的值。之所以让W比较小，是因为如果使用sigmoid函数或者tanh函数作为激活函数的话，W比较小，得到的|z|也比较小（靠近零点），而零点区域的梯度比较大，这样能大大提高梯度下降算法的更新速度，尽快找到全局最优解。如果W较大，得到的|z|也比较大，附近曲线平缓，梯度较小，训练过程会慢很多。

当然，如果激活函数是ReLU或者Leaky ReLU函数，则不需要考虑这个问题。但是，如果输出层是sigmoid函数，则对应的权重W最好初始化到比较小的值。

12. Summary