



deeplearning.ai

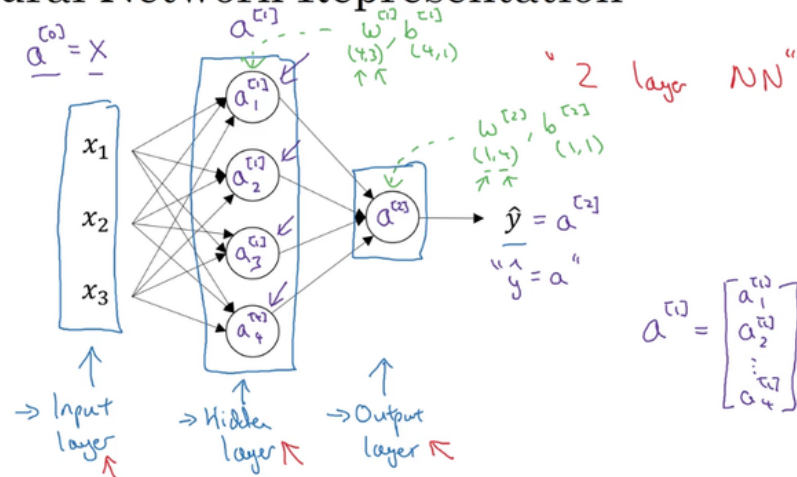
## 吴恩达 DeepLearning.ai 课程提炼笔记 (1-3) 神经网络和深度学习 --- 浅层神经网络

以下为在Coursera上吴恩达老师的DeepLearning.ai课程项目中，第一部分《神经网络和深度学习》第三周课程部分关键点的笔记。笔记并不包含全部小视频课程的记录，如需学习笔记中舍弃的内容请至Coursera 或者 网易云课堂。同时在阅读以下笔记之前，强烈建议先学习吴恩达老师的视频课程。

### 1. 神经网络表示

简单神经网络示意图：

### Neural Network Representation



Andrew Ng

神经网络基本的结构和符号可以从上面的图中看出，这里不再复述。

主要需要注意的一点，是层与层之间参数矩阵的规格大小：

- 输入层和隐藏层之间
  - $w^{[1]} \rightarrow (4, 3)$ ：前面的4是隐层神经元的个数，后面的3是输入层神经元的个数；
  - $b^{[1]} \rightarrow (4, 1)$ ：和隐藏层的神经元个数相同；
- 隐藏层和输出层之间
  - $w^{[2]} \rightarrow (1, 4)$ ：前面的1是输出层神经元的个数，后面的4是隐层神经元的个数；
  - $b^{[2]} \rightarrow (1, 1)$ ：和输出层的神经元个数相同；

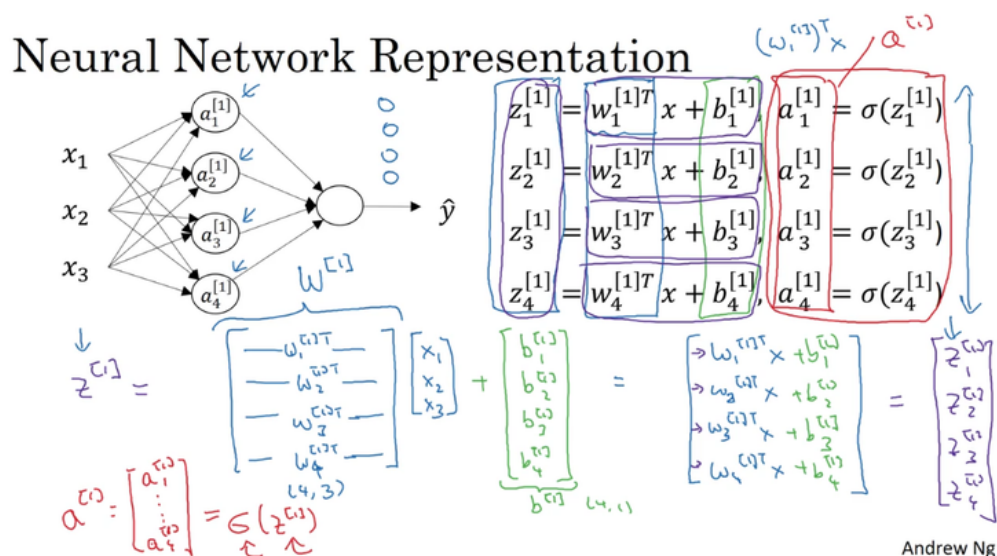


由上面我们可以总结出，在神经网络中，我们以相邻两层为观测对象，前面一层作为输入，后面一层作为输出，两层之间的 $w$ 参数矩阵大小为  $(n_{out}, n_{in})$ ， $b$ 参数矩阵大小为  $(n_{out}, 1)$ ，这里是作为  $z = wX + b$  的线性关系来说明的，在神经网络中， $w^{[i]} = w^T$ 。

在logistic regression中，一般我们都会用  $(n_{in}, n_{out})$  来表示参数大小，计算使用的公式为：  
 $z = w^T X + b$ ，要注意这两者的区别。

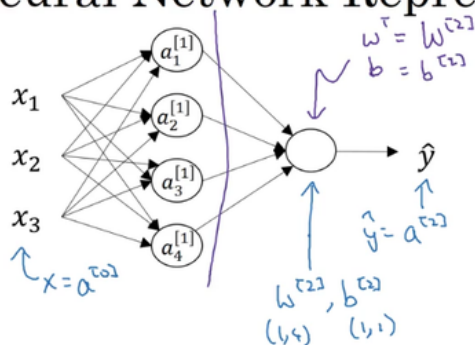
## 2. 计算神经网络的输出

除输入层之外每层的计算输出可由下图总结出：



其中，每个结点都对应这两个部分的运算， $z$ 运算和 $a$ 运算。在编程中，我们使用向量化去计算神经网络的输出：

## Neural Network Representation learning



Given input  $x$ :

$$\rightarrow z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

$(4,1)$      $(4,3)$   $(3,1)$      $(4,1)$

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

$(4,1)$      $(4,1)$

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

$(1,1)$      $(1,4)$   $(4,1)$      $(1,1)$

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

$(1,1)$      $(1,1)$

在对应图中的神经网络结构，我们只用Python代码去实现右边的四个公式即可实现神经网络的输出计算。

## 3. 向量化实现

假定在 $m$ 个训练样本的神经网络中，计算神经网络的输出，用向量化的方法去实现可以避免在程序中使用for循环，提高计算的速度。

下面是实现向量化的解释：

## Justification for vectorized implementation

Diagram illustrating the justification for vectorized implementation. It shows the calculation of  $z^{(1)}$  for multiple samples. Individual calculations are shown at the top:  $z^{(1)}(1) = w^{(1)} x^{(1)} + b^{(1)}$ ,  $z^{(1)}(2) = w^{(1)} x^{(2)} + b^{(1)}$ , and  $z^{(1)}(3) = w^{(1)} x^{(3)} + b^{(1)}$ . Below, these are combined into matrix notation.  $W^{(1)}$  is a  $3 \times 1$  column vector,  $X$  is a  $3 \times m$  matrix of input vectors, and  $b^{(1)}$  is a scalar. The matrix multiplication  $W^{(1)} X$  results in a  $3 \times m$  matrix of  $z^{(1)}$  values. Adding  $b^{(1)}$  to each row yields the final output vector  $z^{(1)}$ .

Andrew Ng

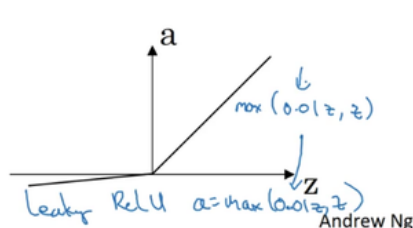
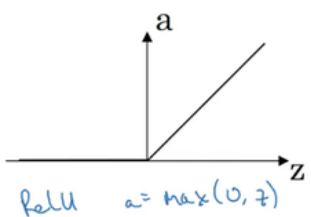
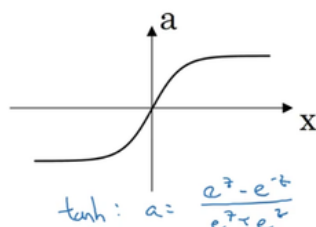
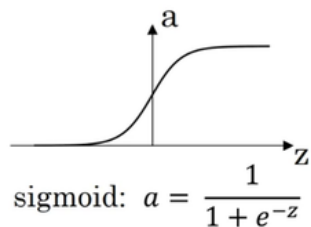
由图可以看出，在  $m$  个训练样本中，每次计算都是在重复相同的过程，均得到同样大小和结构的输出，所以利用向量化的思想将单个样本合并到一个矩阵中，其大小为  $(x_n, m)$ ，其中  $x_n$  表示每个样本输入网络的神经元个数，也可以认为是单个样本的特征数， $m$  表示训练样本的个数。

通过向量化，可以更加便捷快速地实现神经网络的计算。

## 4. 激活函数的选择

几种不同的激活函数  $g(x)$ ：

### Pros and cons of activation functions



- sigmoid:  $a = \frac{1}{1 + e^{-z}}$ 
  - 导数:  $a' = a(1 - a)$
- tanh:  $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ 
  - 导数:
- ReLU (修正线性单元):  $a = \max(0, z)$
- Leaky ReLU:  $a = \max(0.01z, z)$

激活函数的选择:

sigmoid函数和tanh函数比较:

- 隐藏层：tanh函数的表现要好于sigmoid函数，因为tanh取值范围为  $[-1, +1]$ ，输出分布在0值的附近，均值为0，从隐藏层到输出层数据起到了归一化（均值为0）的效果。
- 输出层：对于二分类任务的输出取值为  $\{0, 1\}$ ，故一般会选择sigmoid函数。

然而sigmoid和tanh函数在当  $|z|$  很大的时候，梯度会很小，在依据梯度的算法中，更新在后期会变得很慢。在实际应用中，要使  $|z|$  尽可能的落在0值附近。

ReLU弥补了前两者的缺陷，当  $z > 0$  时，梯度始终为1，从而提高神经网络基于梯度算法的运算速度。然而当  $z < 0$  时，梯度一直为0，但是实际的运用中，该缺陷的影响不是很大。

Leaky ReLU保证在  $z < 0$  的时候，梯度仍然不为0。

在选择激活函数的时候，如果在不知道该选什么的时候就选择ReLU，当然也没有固定答案，要依据实际问题在交叉验证集合中进行验证分析。

## 5. 神经网络的梯度下降法

以本节中的浅层神经网络为例，我们给出神经网络的梯度下降法的公式。

- 参数：  $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$  ；
- 输入层特征向量个数：  $n_x = n^{[0]}$  ；
- 隐藏层神经元个数：  $n^{[1]}$  ；
- 输出层神经元个数：  $n^{[2]} = 1$  ；
- $W^{[1]}$  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  $(n^{[2]}, 1)$  ；

下面为该例子的神经网络反向梯度下降公式（左）和其代码向量化（右）：

### Summary of gradient descent

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

Andrew Ng

## 6. 随机初始化

如果在初始时，两个隐藏神经元的参数设置为相同的大小，那么两个隐藏神经元对输出单元的影响也是相同的，通过反向梯度下降去进行计算的时候，会得到同样的梯度大小，所以在经过多次迭代后，两个隐藏层单位仍然是对称的。无论设置多少个隐藏单元，其最终的影响都是相同的，那么多个隐藏神经元就没有了意义。

在初始化的时候，  $W$  参数要进行随机初始化，  $b$  则不存在对称性的问题它可以设置为0。以2个输入，2个隐藏神经元为例：

```
W = np.random.rand((2,2))* 0.01  
b = np.zeros((2,1))
```

这里我们将W的值乘以0.01是为了尽可能使得权重W初始化为较小的值，这是因为如果使用sigmoid函数或者tanh函数作为激活函数时，W比较小，则  $Z = WX + b$  所得的值也比较小，处在0的附近，0点区域的附近梯度较大，能够大大提高算法的更新速度。而如果W设置的太大的话，得到的梯度较小，训练过程因此会变得很慢。

ReLU和Leaky ReLU作为激活函数时，不存在这种问题，因为在大于0的时候，梯度均为1。

---

编辑于 2017-11-08 08:53

[深度学习 \(Deep Learning\)](#)   [神经网络](#)   [吴恩达 \(Andrew Ng\)](#)