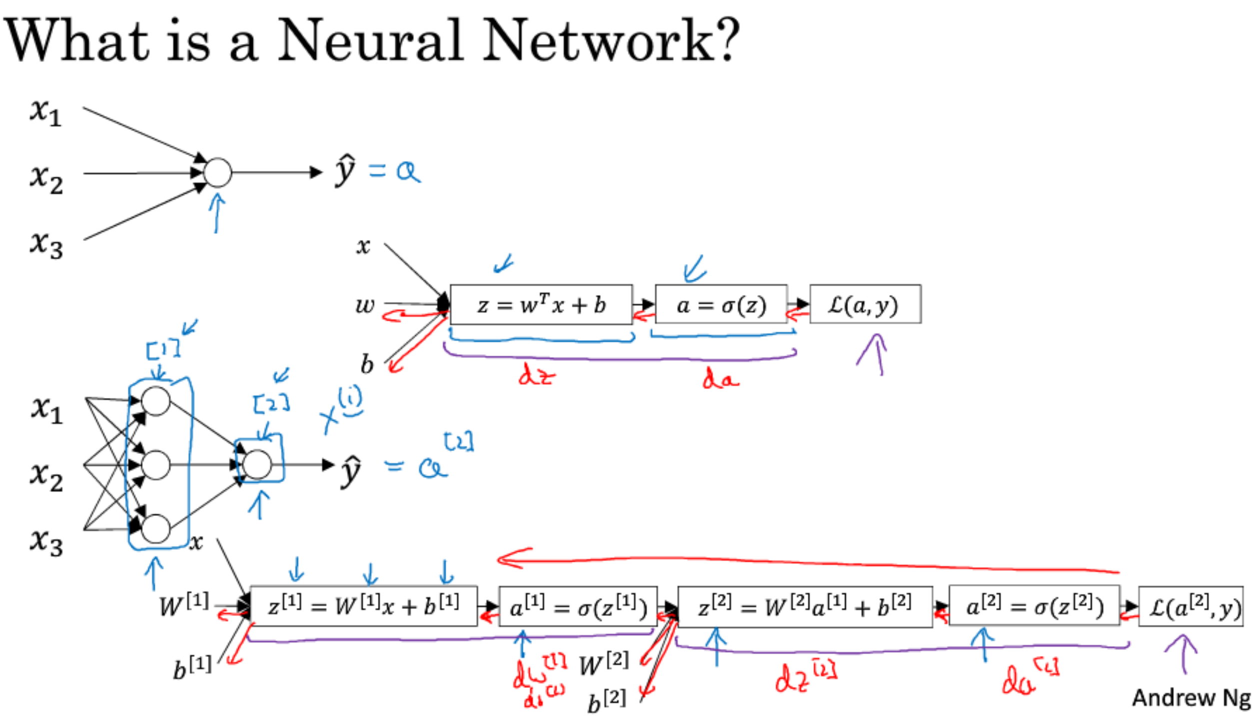
神经网络和深度学习--浅层圣经网络

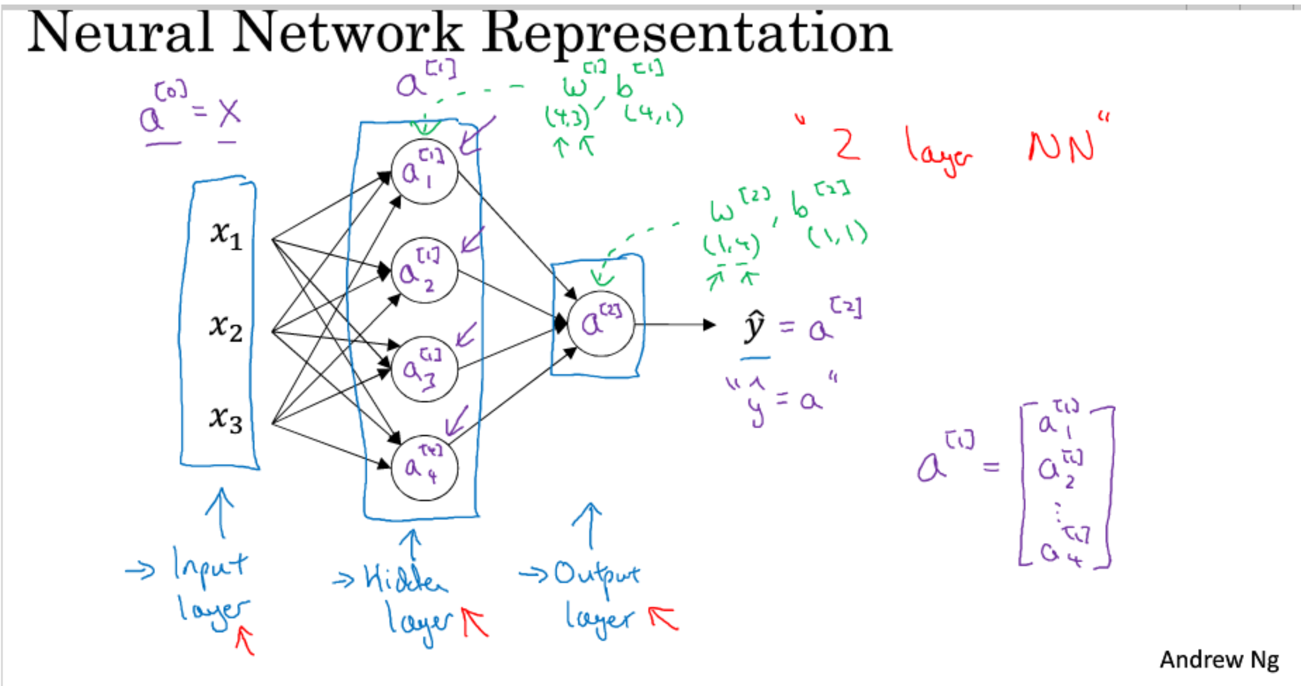
3.1 神经网络概述

单个神经元



3.2 神经网络表示

简单神经网络示意图：



输入层和隐藏层之间

- w[1]−>(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]−>(4,1)：和隐藏层的神经元个数相同；

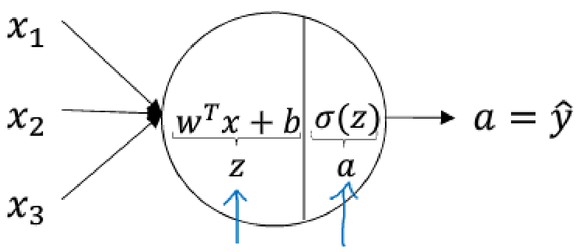
隐藏层和输出层之间

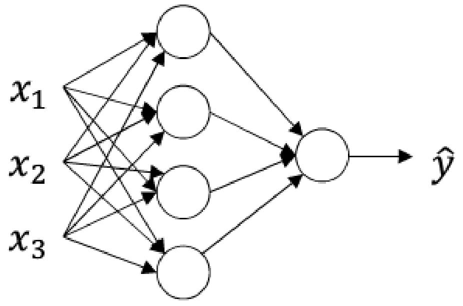
- w[1]−>(1,4)：前面的1是输出层神经元的个数，后面的4是隐层神经元的个数；
- b[1]−>(1,1)：和输出层的神经元个数相同。

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(nout,nin)，b参数矩阵大小为(nout,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]=wT。

3.3 神经网络的输出

Neural Network Representation

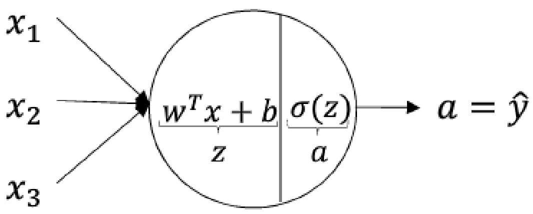


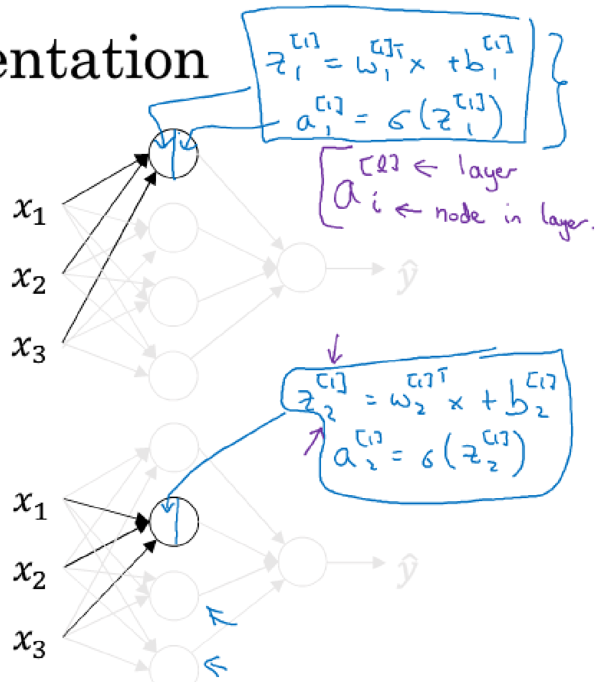


$z = w^T x + b$

$a = \sigma(z)$

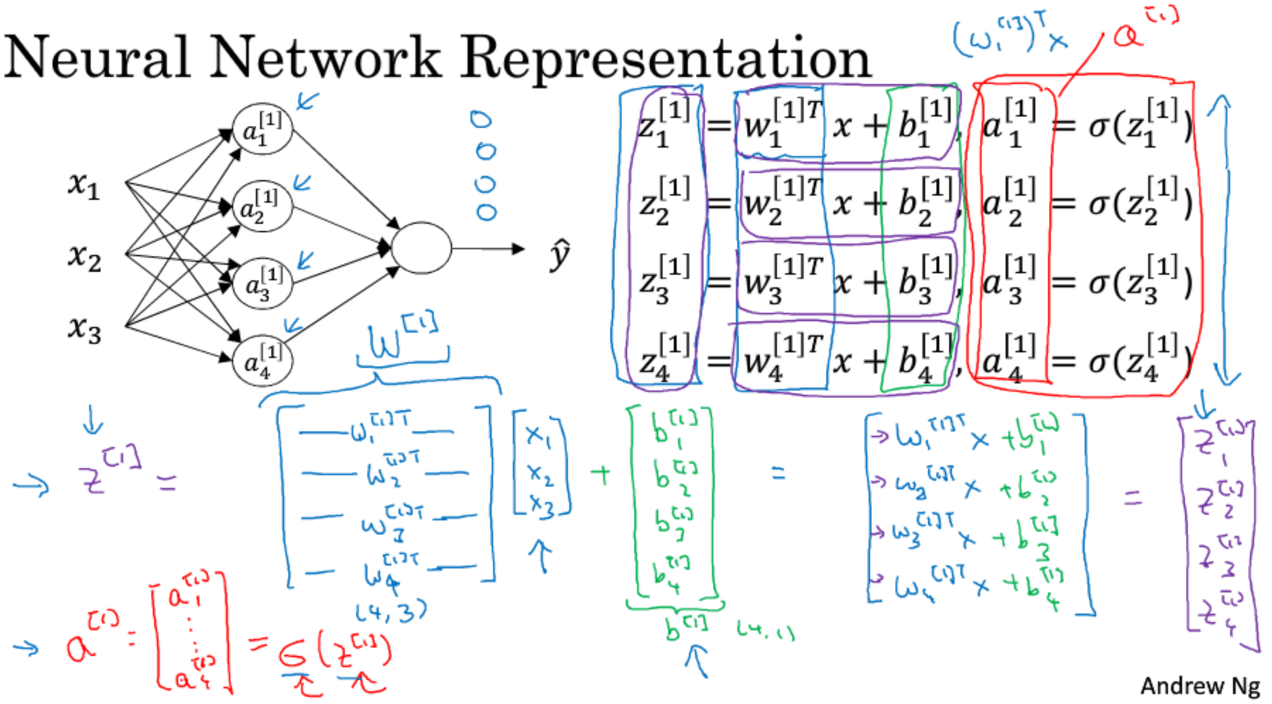
Neural Network Representation



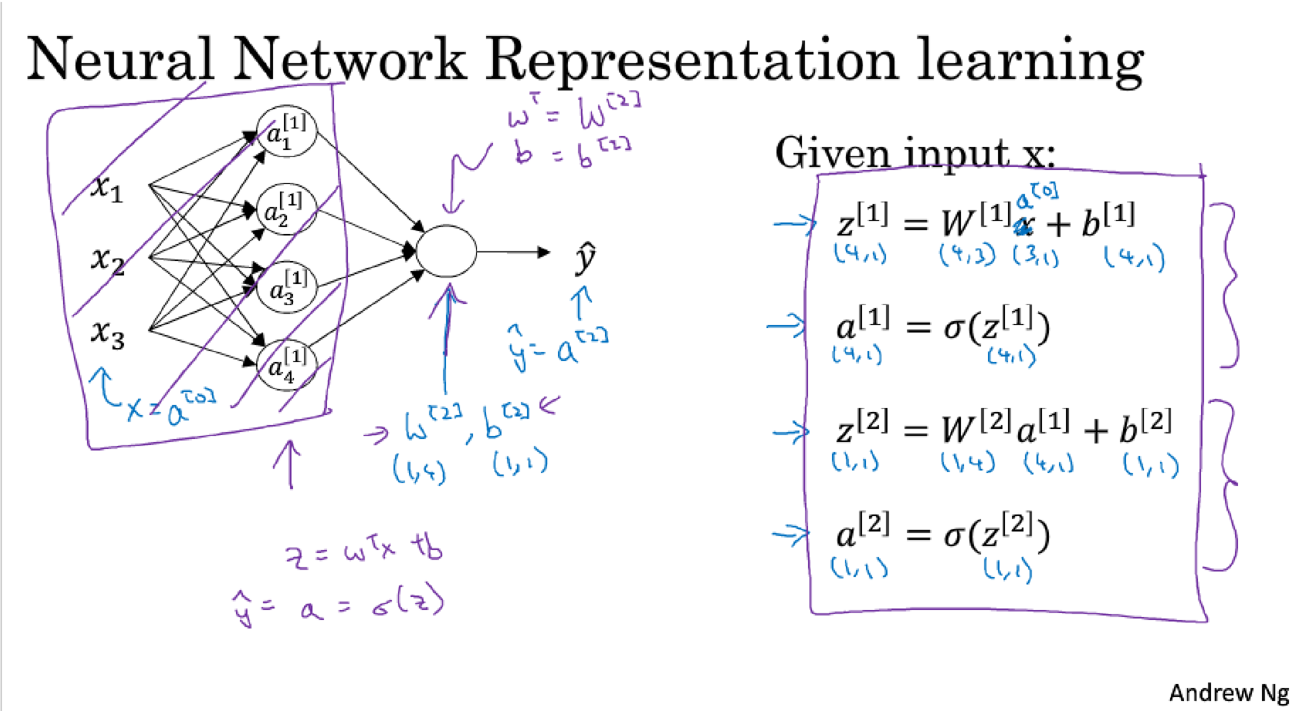


$z = w^T x + b$

$a = \sigma(z)$



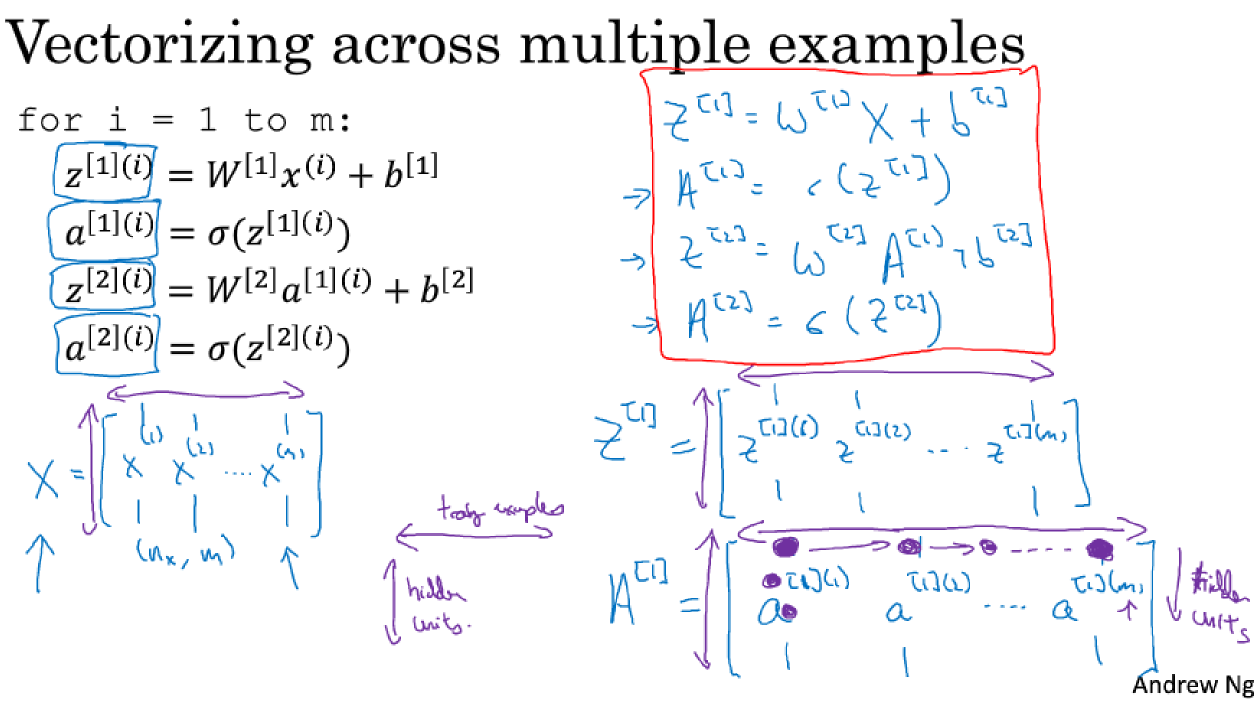
其中，每个结点都对应这两个部分的运算，z运算和a运算。



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。

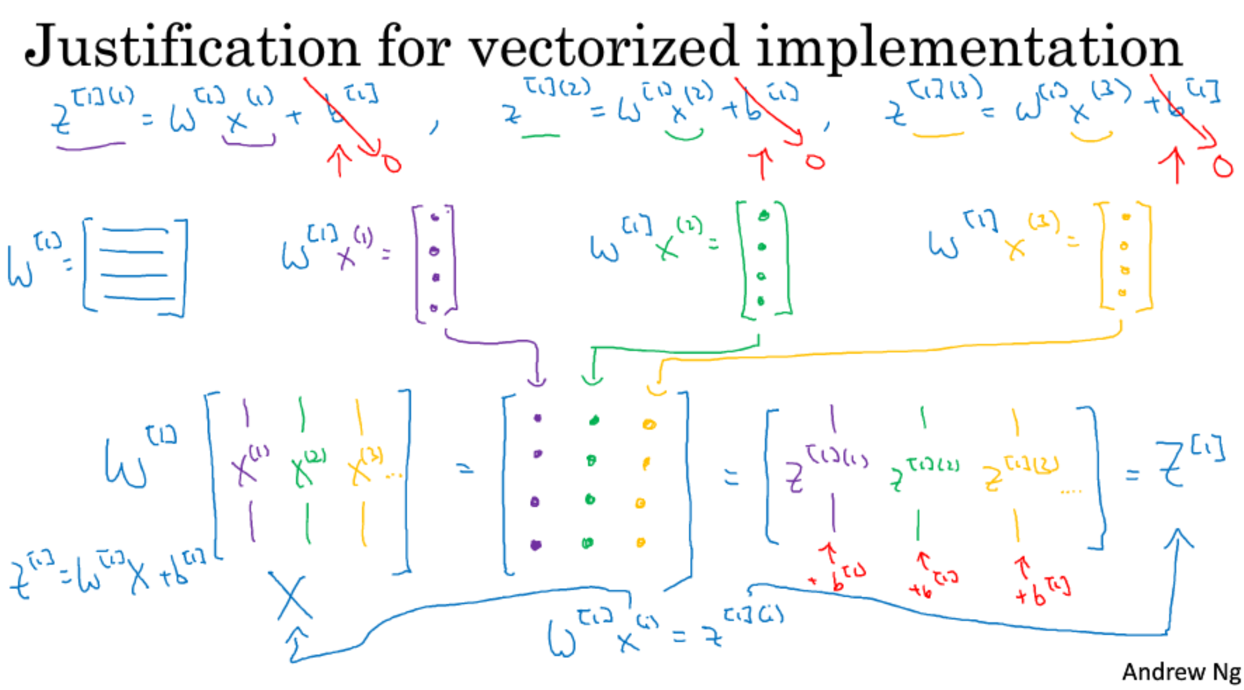
3.4 多个例子中的向量化

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。



向量化多个样本

3.5 向量化的实现的解释

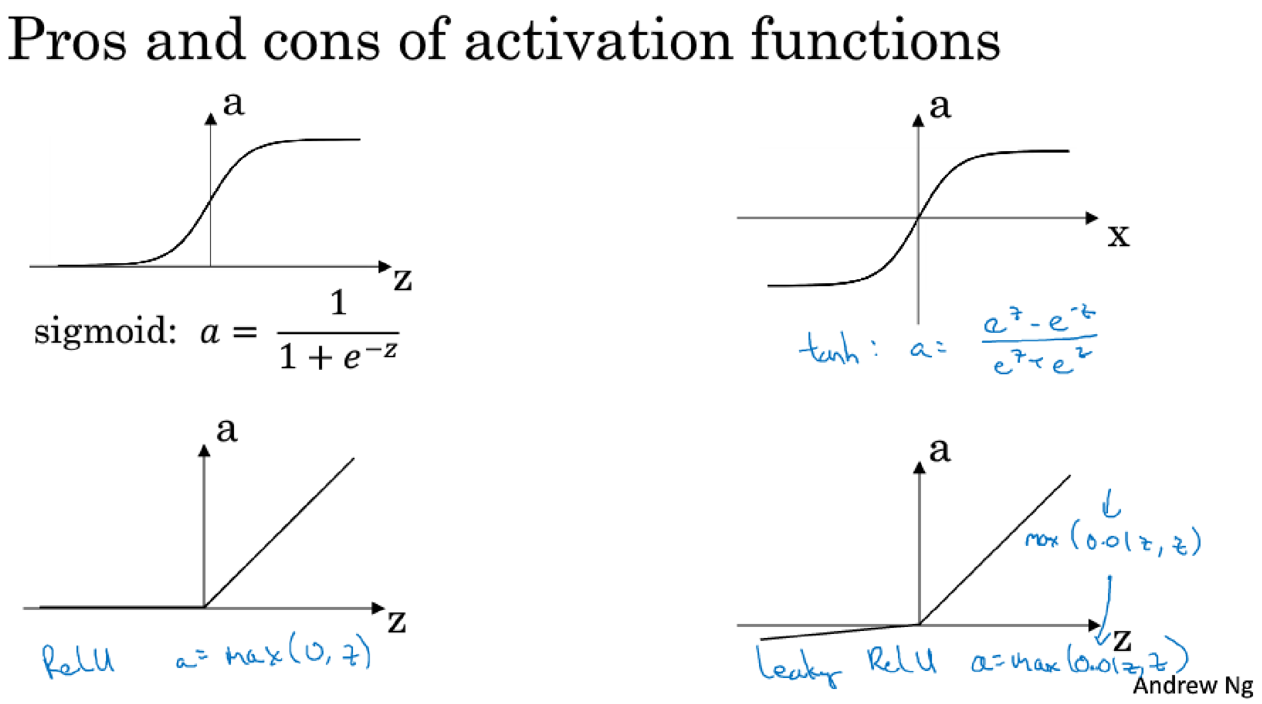


由图可以看出，在m个训练样本中，每次计算都是在重复相同的过程，均得到同样大小和结构的输出，所以利用向量化的思想将单个样本合并到一个矩阵中，其大小为(xn,m)(xn,m)，其中xnxn表示每个样本输入网络的神经元个数，也可以认为是单个样本的特征数，m表示训练样本的个数。

通过向量化，可以更加便捷快速地实现神经网络的计算。

3.6 激活函数

几种不同的激活函数g(x)g(x)：



- 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：$a' = 1 - a^2$
- 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][−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}{0,1}，故一般会选择sigmoid函数。

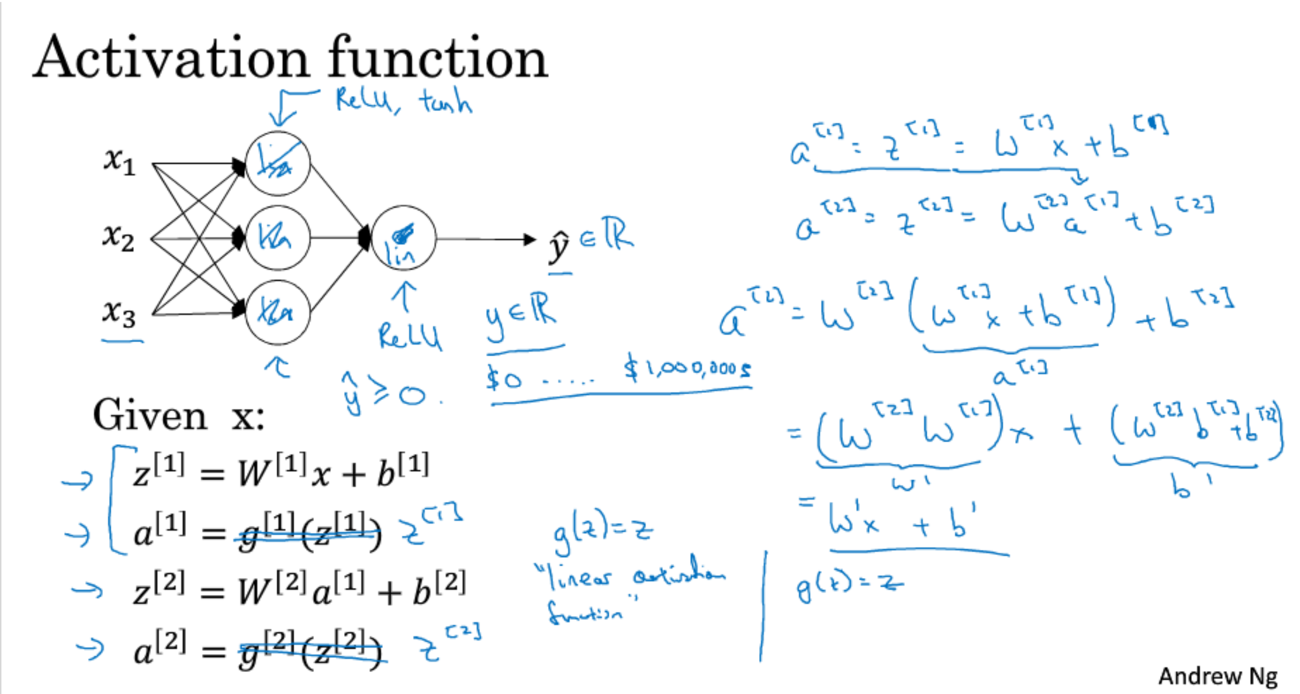
然而sigmoid和tanh函数在当|z||z|很大的时候，梯度会很小，在依据梯度的算法中，更新在后期会变得很慢。在实际应用中，要使|z||z|尽可能的落在0值附近。

ReLU弥补了前两者的缺陷，当z>0z>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<0z<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<0z<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。

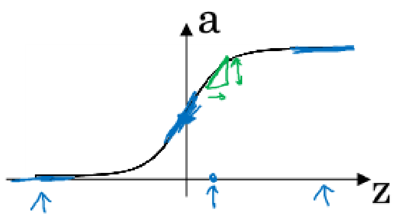
3.7 为什么需要非线性激活函数



3.8 激活函数的导数

sigmoId激活函数

Sigmoid activation function



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

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

$\boxed{g'(z)} = \boxed{\frac{d}{dz}g(z)}$ = slope of $g(x)$ at z

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

$= g(z)\left(1-g(z)\right)$

$= \boxed{a(1-a)}$

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

$z=10.$ $g(z) \approx 1$

$\frac{d}{dz}g(z) \approx 1(1-1) \approx 0$

$z=-10$ $g(z) \approx 0$

$\frac{d}{dz}g(z) \approx 0\cdot(1-0) \approx 0$

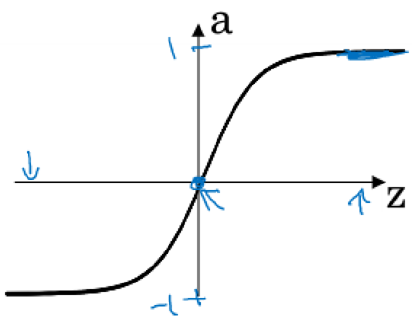
$z=0$ $g(z)=\frac{1}{2}$

$\frac{d}{dz}g(z) = \frac{1}{2}\left(1-\frac{1}{2}\right) = \frac{1}{4}$

Andrew Ng

Tanh激活函数

Tanh activation function



$g(z) = \tanh(z)$

$= \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

$g'(z) = \frac{d}{dz}g(z)$ = slope of $g(z)$ at z

$= 1-(\tanh(z))^2$

$a = g(z),$ $g'(z) = 1-a^2$

$z=10$ $\tanh(z) \approx 1$

$g'(z) \approx 0$

$z=-10$ $\tanh(z) \approx -1$

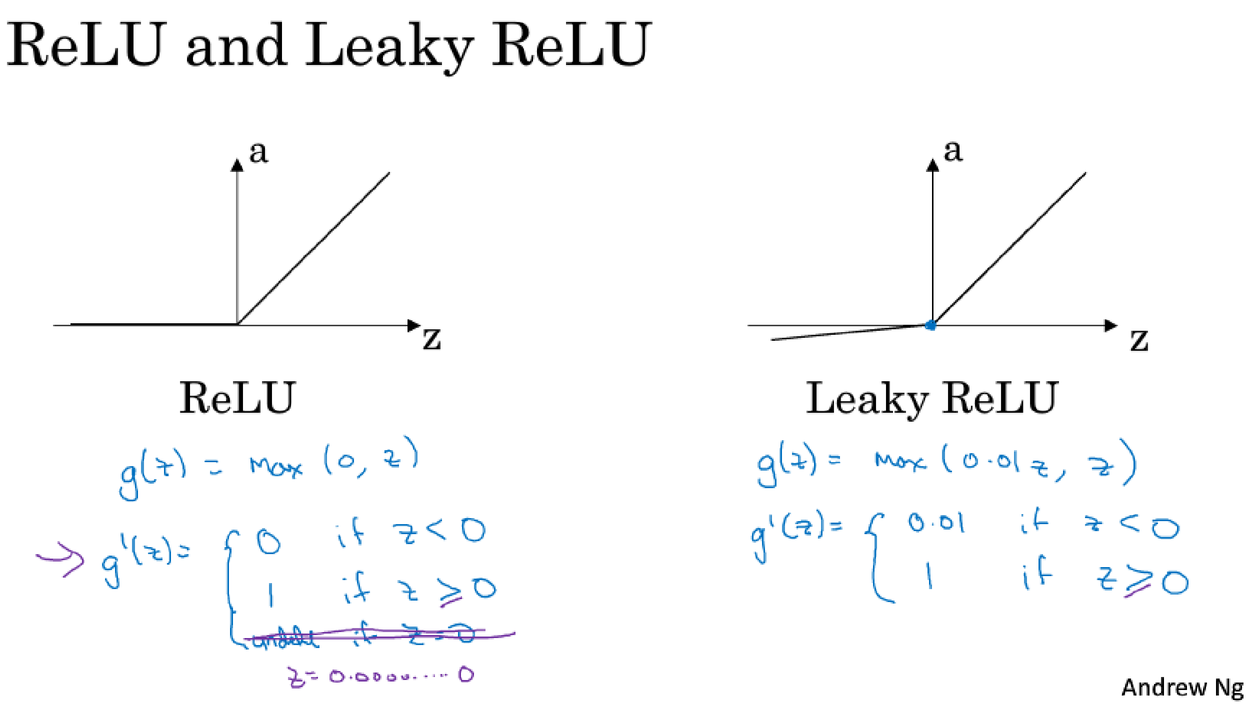
$g'(z) \approx 0$

$z=0$ $\tanh(z)=0$

$g'(z)=1$

Andrew Ng

Relu激活函数



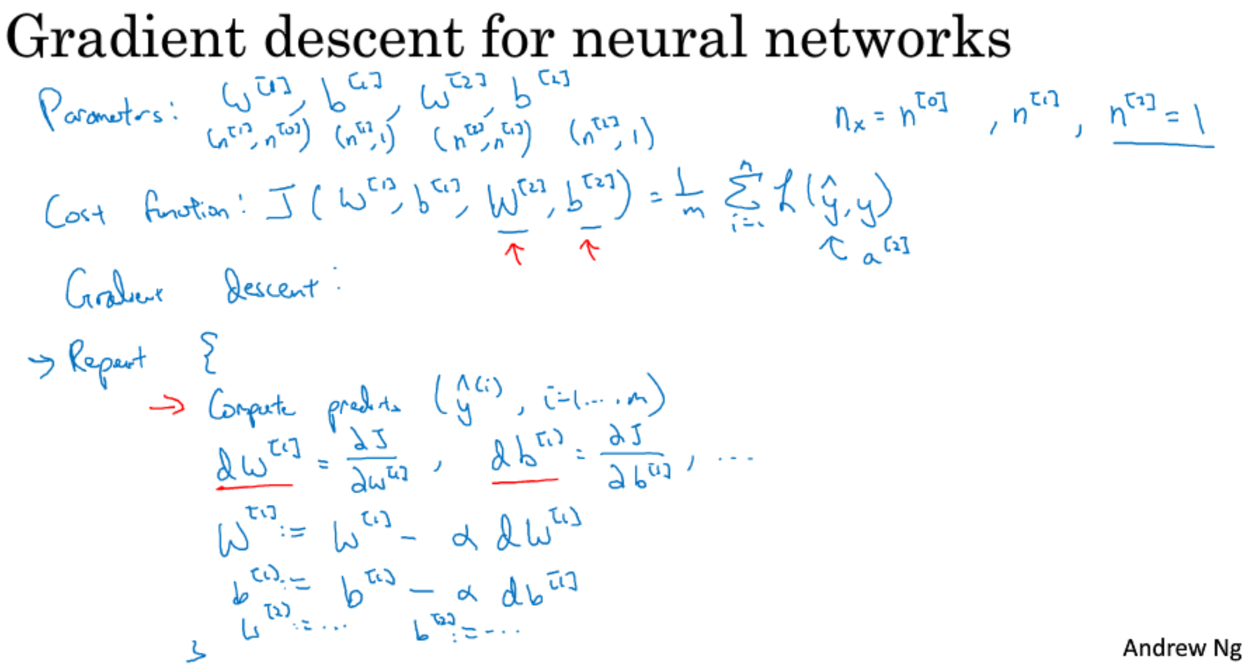
3.9 神经网络的激活函数

激活函数的梯度下降法

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]W[1],b[1],W[2],b[2]；
- 输入层特征向量个数：nx=n[0]nx=n[0]；
- 隐藏层神经元个数：n[1]n[1]；
- 输出层神经元个数：n[2]=1n[2]=1；
- W[1]W[1]的维度为(n[1],n[0])(n[1],n[0])，b[1]b[1]的维度为(n[1],1)(n[1],1)；
- W[2]W[2]的维度为(n[2],n[1])(n[2],n[1])，b[2]b[2]的维度为(n[2],1)(n[2],1)；

- 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$(n^{[2]}, 1)$；

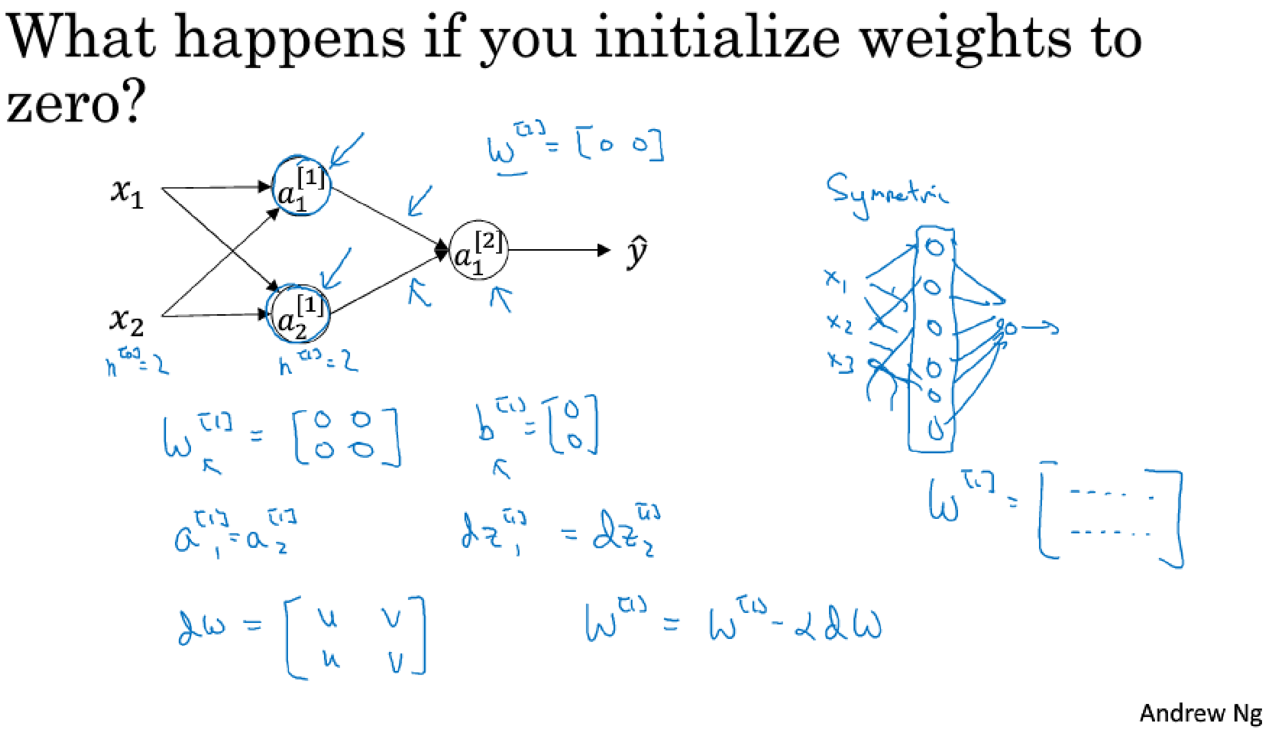


下面为该例子的神经网络反向梯度下降公式（左）和其代码向量化（右）：

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

3.10 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，WW参数要进行随机初始化，bb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.zero((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+bZ=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。