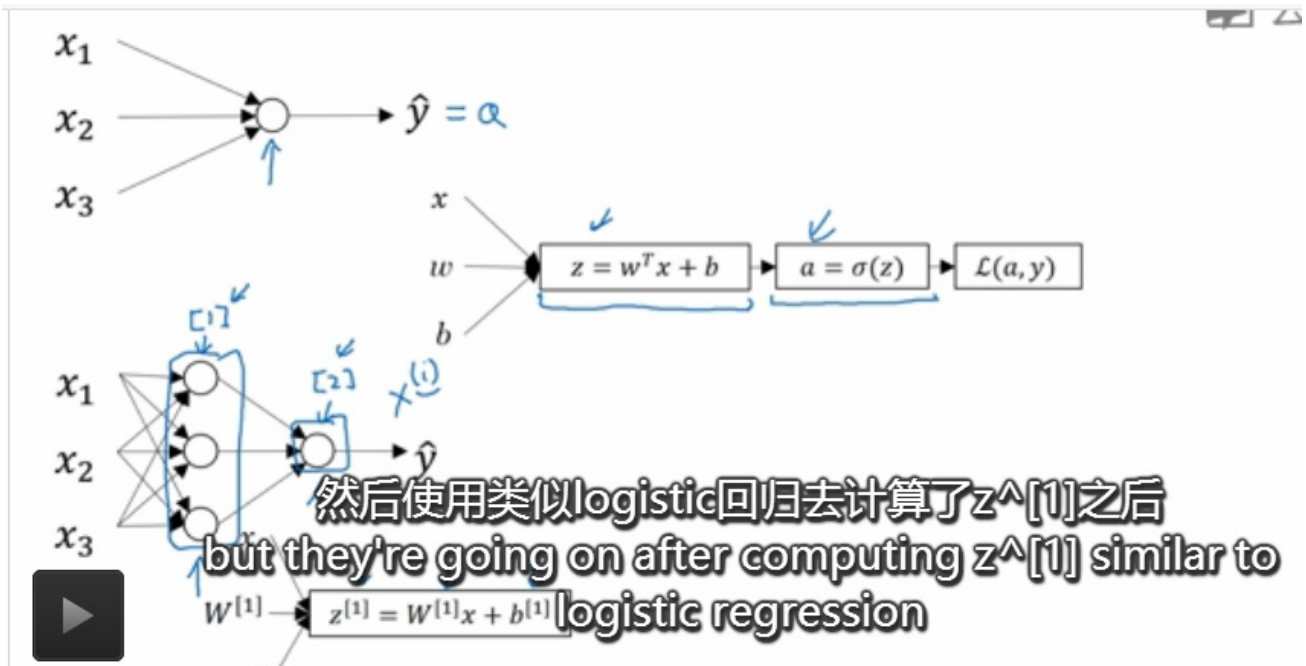


第一篇第三周

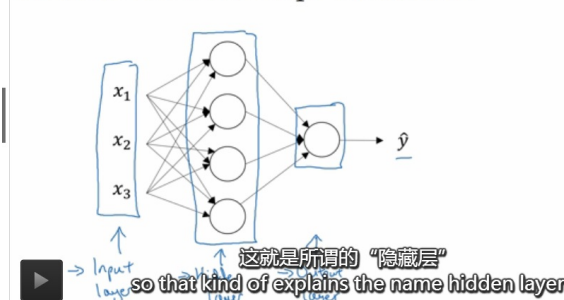
3.1、如何实现神经网络：



从上图可以看出左下角的是一个神经网络的结构，矩形框的是它的层数，有两层， x 是输入， w 和 b 是参数，用线性回归计算 $z^{[1]}$ 之后，需要使用 $\text{sigmoid}(z^{[1]})$ 计算 $a^{[1]}$ ，接着你需要用另外一个线性方程计算 $z^{[2]}$ ，接着计算 $a^{[2]}$ 。

3.2、神经网络的表示：

Neural Network Representation



神经网络分为输入层，隐藏层和输出层。

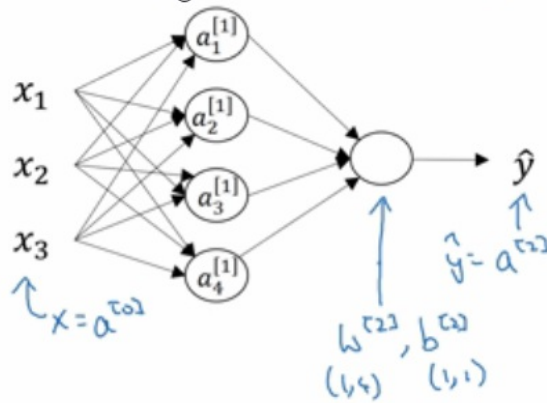
隐藏层：在训练集中，这些中间节点的真实数值，我们是不知道的，在训练集你看不到他们的值，你只能看到输入层和输出层。

计算神经网络的层数，一般不包括输入层，所以上面的是两层神经网络。隐藏层是第一层，输出层是第二层。

隐藏层的参数维度是 3×4 的矩阵，输出层的参数矩阵是 4×1 ，与节点数有关。

3.3、神经网络的输出：

Neural Network Representation learning



Given input x:

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

$\begin{matrix} (4,1) & (4,2) & (3,1) & (4,1) \end{matrix}$

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

$\begin{matrix} (4,1) & (4,1) \end{matrix}$

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

$\begin{matrix} (1,1) & (1,4) & (4,1) & (1,1) \end{matrix}$

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

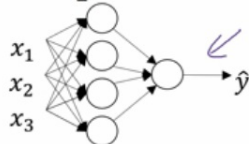
$\begin{matrix} (1,1) \end{matrix}$

如果你把这最后的输出单元
and if you think of this last output unit

注意上图的矩阵化的维度，一定要能够进行矩阵相乘。通过上面右边的函数转化成python的4行代码，你就可以算出上面神经网络的输出。

3.4、神经网络向量化：

Recap of vectorizing across multiple examples



$$X = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & | & | \end{bmatrix}$$

```
for i = 1 to m
  → z[1](i) = W[1]x(i) + b[1]
  → a[1](i) = σ(z[1](i))
  → z[2](i) = W[2]a[1](i) + b[2]
  → a[2](i) = σ(z[2](i))
```

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

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

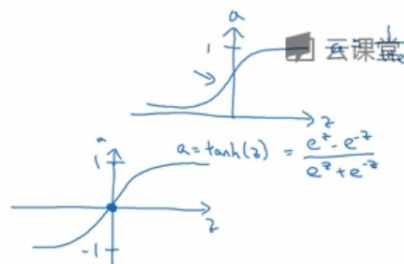
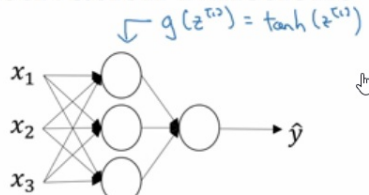
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

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

对应各列堆叠起来是这样的
The stack of the corresponding columns as follows

3.6、激活函数

Activation functions

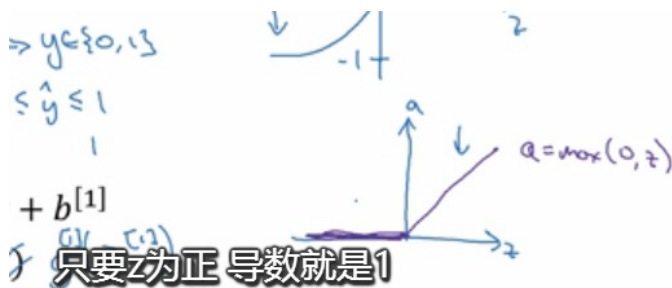


Given x:

使用tanh而不是σ函数也有类似数据中心化的效果
using a tanh instead of a sigmoid function kind of has
the effect of centering your data

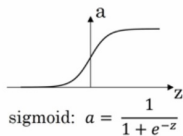
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

tanh的效果更好。在所有场合都更优越，这两个函数的缺点是当z很大或者很小时，使得导数接近0，这样会拖慢梯度下降算法。为了不拖慢梯度下降算法，可以使用relu函数：



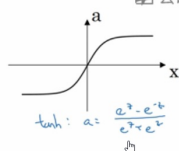
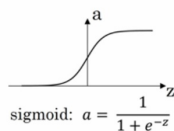
经验：如果输出是0或者1，做二元分类，用第一个函数，作为输出层的激活函数，然后其他所有单元都用relu函数。relu的优点是学习速度非常快。

Pros and cons of activation functions



除非用在二元分类的输出层 不然绝对不要用
 I would say never use this except for the output layer if you are doing binary classification

Pros and cons of activation functions



还有最常用的默认激活函数是ReLU
 and then the default the most commonly used activation function is the ReLU

3.7、为什么要使用非线性的激活函数：

如果使用线性激活函数，那么神经网络只是把输入线性组合再输出，这样的话无论你的深度网络多少层，一直在做的只是线性激活函数，所以不如直接去掉隐藏层，线性隐藏层一点都没有用，因为不管多少个线性函数它们的组合仍然是线性函数。

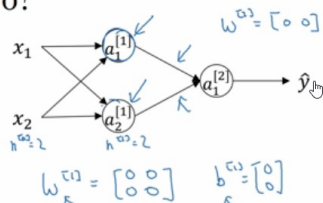
3.8、激活函数的导数

当你对你的神经网络使用反向传播的时候，你需要计算激活函数的导数或者斜率。

3.9、要训练参数，就要使用梯度下降，在训练神经网络时，随机初始化参数很重要，而不是初始化成全0，全部初始化为0的话，梯度下降算法就会失效。

3.11、

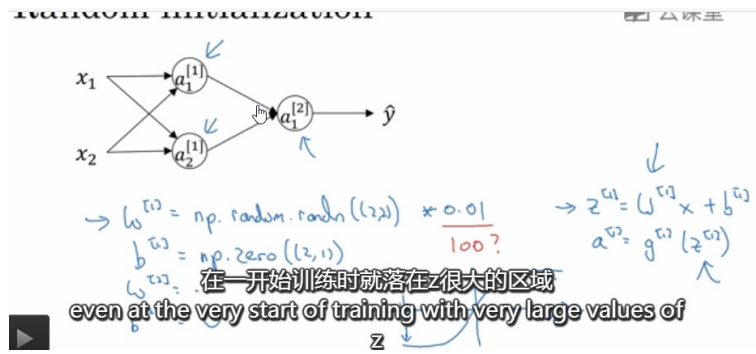
What happens if you initialize weights to zero?



那么这个隐藏单元和这个隐藏单元就完全一样了
 then this hidden unit and this hidden unit are completely identical

这就是所谓的完全对称。意味着节点计算完全一样的函数，如果你将w值所有初始化为0，那么因为两个隐藏层节点一开始就做同样的计算，两个隐藏层单元对输出单元的影响也一样大，那么在一次迭代之后，同样的对称性依旧存在，两个隐藏层单元仍然是对称的，所以无论你训练多长时间，两个隐藏层单元仍在计算完全一样的函数，所以，多个隐藏层单元真的没有意义，这个问题的解决是随机初始化所有的参数，你可以用 $w1 = np.random.randn([2,2])$, b没有这种问题，所以就可以都初始化为0，w初始化为0.0几的数据，是为了防止tanh或者sigmoid函数落在平缓部分，梯度的斜度非常小，也就是说w如果太大，在一开始训练的时候就落在Z很大的区域，导致你的tanh或者sigmoid激活函数接近饱和，从而减慢学习速

度，如果没有tanh或者sigmoid激活函数，就没那么大的问题，因为二分类的时候，有sigmoid函数，所以w不要初始化的太大。0.01是比较合理的，所以初始化参数一般都很小，但是层次很深的就要有大于0.01的初始值，下周再讲，上面结论仅针对单层神经网络，如下图：



吴恩达深度学习代码: https://github.com/hejue/Deep_learning_coursera_Andrew-Ng