2.11 向量化(Vectorization)

向量化是非常基础的去除代码中 **for** 循环的艺术，在深度学习安全领域、深度学习实践中，你会经常发现自己训练大数据集，因为深度学习算法处理大数据集效果很棒，所以你的代码运行速度非常重要，否则如果在大数据集上，你的代码可能花费很长时间去运行，你将要等待非常长的时间去得到结果。所以在深度学习领域，运行向量化是一个关键的技巧，让我们举个栗子说明什么是向量化。

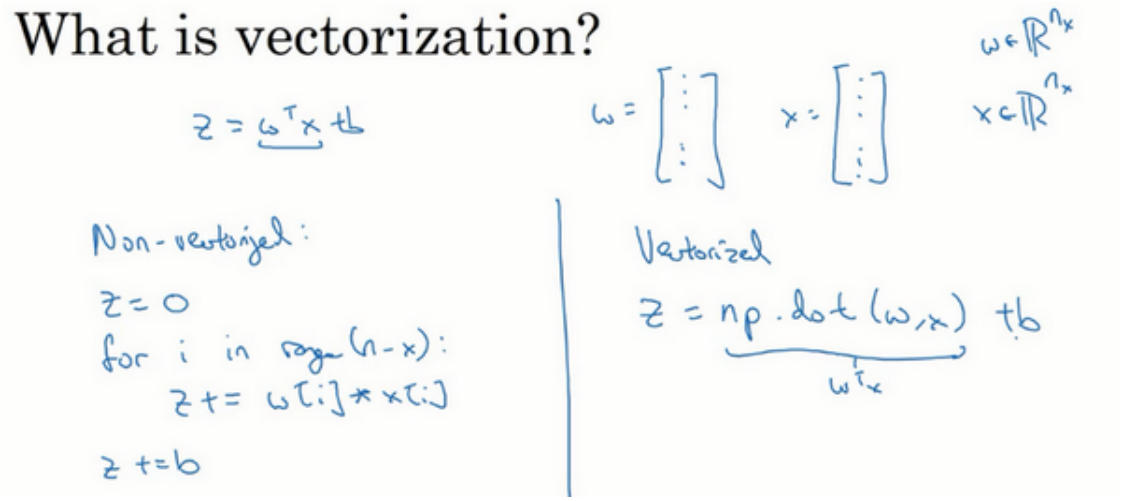
在逻辑回归中你需要去计算$z = w^T x + b$，w、x都是列向量。如果你有很多的特征那么就会有一个非常大的向量，所以$w \in \mathbb{R}^{n_x}$ ，$x \in \mathbb{R}^{n_x}$，所以如果你想使用非向量化方法去计算$w^T x$，你需要用如下方式（**python**）

```
z=0
for i in range(n_x)
   z+=w[i]*x[i]
z+=b
```

这是一个非向量化的实现，你会发现这真的很慢，作为一个对比，向量化实现将会非常直接计算$w^T x$，代码如下：

```
z=np.dot(w,x)+b
```

这是向量化计算$w^T x$的方法，你将会发现这个非常快



和 for 循环相比，向量化可以快速得到结果。

你可能听过很多类似如下的话，“大规模的深度学习使用了 **GPU** 或者图像处理单元实现”，但是我做的所有的案例都是在 **jupyter notebook** 上面实现，这里只有 **CPU**，CPU 和 GPU

都有并行化的指令，他们有时候会叫做 **SIMD** 指令，这个代表了一个单独指令多维数据，这个的基础意义是，如果你使用了 **built-in** 函数,像 `np.function` 或者并不要求你实现循环的函数，它可以让 python 的充分利用并行化计算，这是事实在 **GPU** 和 **CPU** 上面计算，**GPU** 更加擅长 **SIMD** 计算，但是 **CPU** 事实上也不是太差，可能没有 **GPU** 那么擅长吧。接下来的视频中，你将看到向量化怎么能够加速你的代码，经验法则是，无论什么时候，避免使用明确的 for 循环。

以下代码及运行结果截图：

In [1]:
```
import numpy as np

a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

In [13]:
```
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print("Vectorized version:" + str(1000*(toc-tic)) +"ms")
```

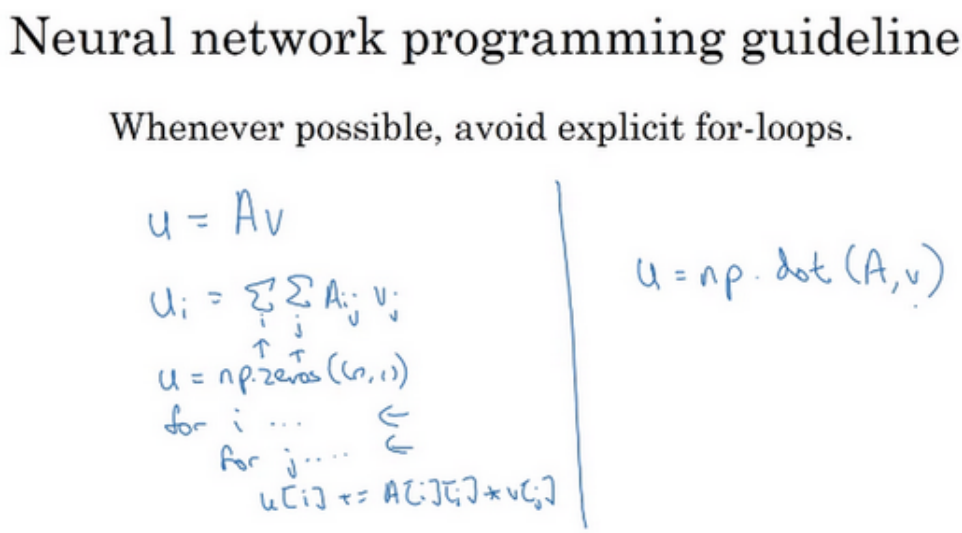
```
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) +"ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")
```

```
250286.989866
Vectorized version:1.5027523040771484ms
250286.989866
For loop:474.29513931274414ms
```

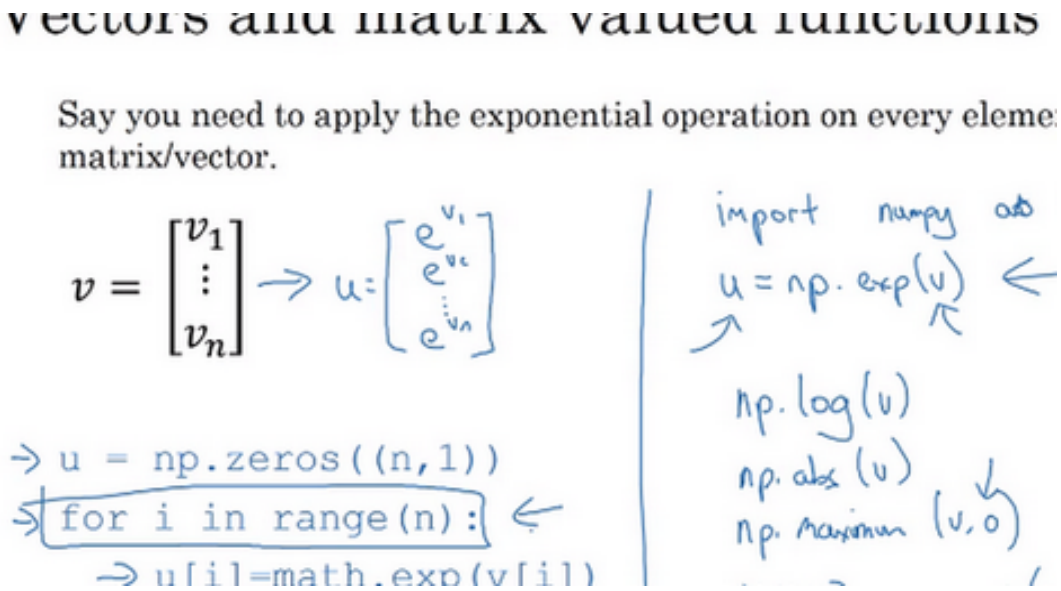
2.12 向量化的更多例子（More Examples of Vectorization）

经验提醒我，当我们在写神经网络程序时，或者在写逻辑(**logistic**)回归，或者其他神经网络模型时，应该**避免写循环(loop)语句**。虽然有时写循环(**loop**)是不可避免的，但是我们可以使用比如 **numpy** 的内置函数或者其他办法去计算。当你这样使用后，程序效率总是快于循环(**loop**)。

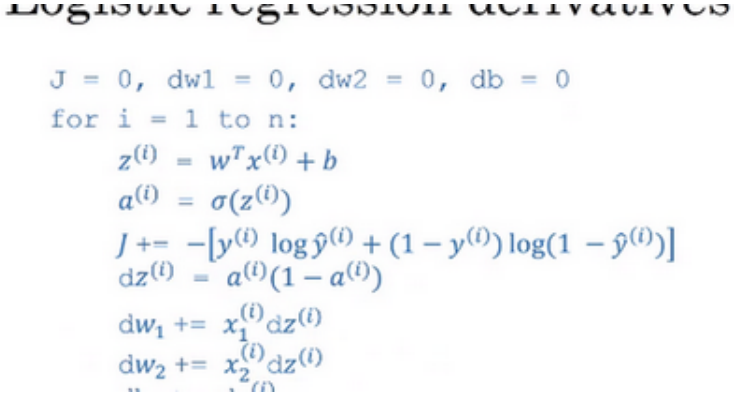
让我们看另外一个例子。如果你想计算向量$u = Av$，这时矩阵乘法定义为，矩阵乘法的定义就是：$u_i = \sum_j A_{ij} v_i$，这取决于你怎么定义u_i值。同样使用非向量化实现，$u = np.zeros(n,1)$，并且通过两层循环$for(i): for(j):$，得到$u[i] = u[i] + A[i][j] * v[j]$ 。现在就有了i 和 j 的两层循环，这就是非向量化。向量化方式就可以用$u = np.dot(A,v)$，右边这种向量化实现方式，消除了两层循环使得代码运行速度更快。



事实上，**numpy** 库有很多向量函数。比如 `u=np.log` 是计算对数函数(log)、`np.abs()` 是计算数据的绝对值、`np.maximum()` 计算元素 y 中的最大值，你也可以 `np.maximum(v,0)` 、v `**` `2` 代表获得元素 y 每个值得平方、$\frac{1}{v}$ 获取元素 y 的倒数等等。所以当你想写循环时候，检查 **numpy** 是否存在类似的内置函数，从而避免使用循环(**loop**)方式。



那么，将刚才所学到的内容，运用在逻辑回归的梯度下降上，看看我们是否能简化两个计算过程中的某一步。这是我们逻辑回归的求导代码，有两层循环。在这例子我们有n个特征值。如果你有超过两个特征时，需要循环 dw_1 、dw_2 、dw_3 等等。所以 j 的实际值是1、2 和 n_x，就是你想要更新的值。所以我们想要消除第二循环，在这一行，这样我们就不用初始化 dw_1 ， dw_2 都等于 0。去掉这些，而是定义 dw 为一个向量，设置 $u = np.zeros(n(x),1)$。定义了一个x行的一维向量，从而替代循环。我们仅仅使用了一个向量操作 $dw = dw + x^{(i)}dz^{(i)}$ 。最后，我们得到 $dw = dw/m$ 。现在我们通过将两层循环转成一层循环，我们仍然还有这个循环训练样本。



Logistic regression derivatives

$J = 0$, ~~$dw_1 = 0$, $dw_2 = 0$~~, $db = 0$ $\quad dw = np.zeros((n_x, 1))$

for $i = 1$ to n:

$z^{(i)} = w^T x^{(i)} + b$

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

$J \mathrel{+}= -\left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right]$

$dz^{(i)} = a^{(i)}(1 - a^{(i)})$

for $j = 1 \ldots n_x$:

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$ $\quad n_x = 2$ $\quad dw \mathrel{+}= x^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

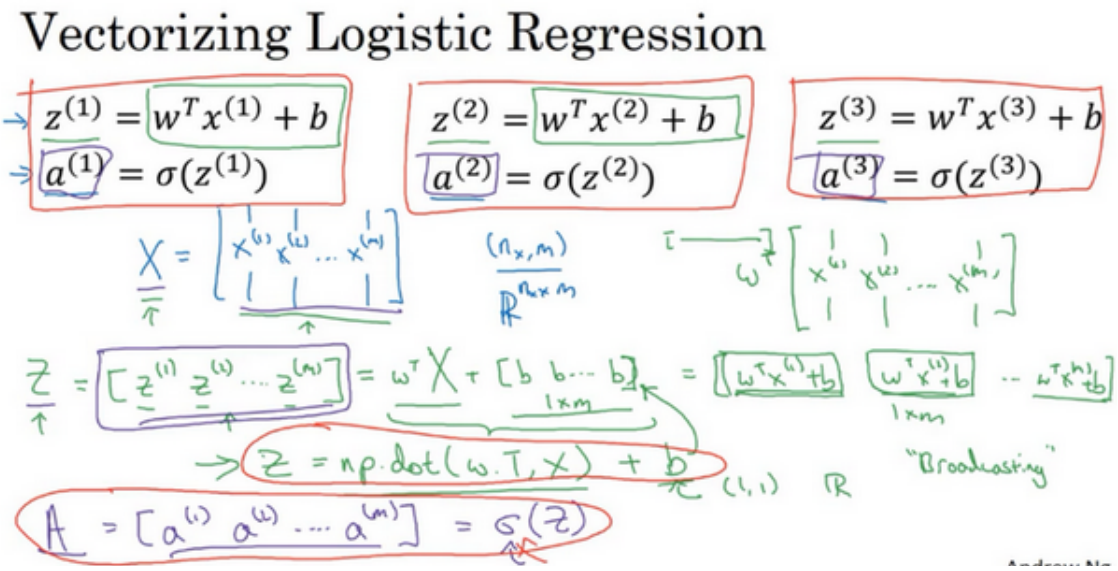
$J = J/m$, ~~$dw_1 = dw_1/m$, $dw_2 = dw_2/m$~~, $db = db/m$

2.13 向量化逻辑回归(Vectorizing Logistic Regression)

让我们开始吧，首先我们回顾一下逻辑回归的前向传播步骤。所以，如果你有 m 个训练样本，然后对第一个样本进行预测，你需要这样计算。计算 z，我正在使用这个熟悉的公式 $z^{(1)} = w^T x^{(1)} + b$ 。然后计算激活函数 $a^{(1)} = \sigma(z^{(1)})$ ，计算第一个样本的预测值 y 。

对第二个样本进行预测，你需要计算 $z^{(2)} = w^T x^{(2)} + b$ ， $a^{(2)} = \sigma(z^{(2)})$ 。

对第三个样本进行预测，你需要计算 $z^{(3)} = w^T x^{(3)} + b$ ， $a^{(3)} = \sigma(z^{(3)})$ ，依次类推。如果你有 m 个训练样本，你可能需要这样做 m 次，可以看出，为了完成前向传播步骤，即对我们的 m 个样本都计算出预测值。首先，回忆一下我们曾经定义了一个矩阵 X 作为你的训练输入，(如下图中蓝色 X)像这样在不同的列中堆积在一起。这是一个 n_x 行 m 列的矩阵。我现在将它写为 **Python numpy** 的形式 (n_x, m) ，这只是表示 X 是一个 n_x 乘以 m 的矩阵 $R^{n_x \times m}$。



现在我首先想做的是告诉你该如何在一个步骤中计算 z_1、 z_2 、z_3 等等。实际上，只用了一行代码。所以，我打算先构建一个 $1 \times m$ 的矩阵，实际上它是一个行向量，同时我准备计算 $z^{(1)}$， $z^{(2)}$ ……一直到 $z^{(m)}$ ，所有值都是在同一时间内完成。结果发现它可以表达为 w 的转置乘以大写矩阵 x 然后加上向量 $[bb\ldots b]$ ， $([z^{(1)} z^{(2)} \ldots z^{(m)}] = w^T + [bb\ldots b])$ 。$[bb\ldots b]$ 是一个 $1 \times m$ 的向量或者 $1 \times m$ 的矩阵或者是一个 m 维的行向量。所以希望你熟悉矩阵乘法，你会发现的 w 转置乘以 $x^{(1)}$ ， $x^{(2)}$ 一直到 $x^{(m)}$ 。所以 w 转置可以是一个行向量。所以第一项 $w^T X$ 将计算 w 的转置乘以 $x^{(1)}$， w 转置乘以$x^{(2)}$ 等等。然后我们加上第二项 $[bb\ldots b]$ ，你最终将 b 加到了每个元素上。所以你最终得到了

另一个 $1\times m$ 的向量，$[z^{(1)}z^{(2)}\dots z^{(m)}]=w^TX+[bb\dots b]=[w^Tx^{(1)}+b,w^Tx^{(2)}+b\dots w^Tx^{(m)}+b]$ 。

$w^Tx^{(1)}+b$ 这是第一个元素，$w^Tx^{(2)}+b$ 这是第二个元素，$w^Tx^{(m)}+b$ 这是第 m 个元素。

如果你参照上面的定义，第一个元素恰好是 $z^{(1)}$ 的定义，第二个元素恰好是 $z^{(2)}$ 的定义，等等。所以，因为X是一次获得的，当你得到你的训练样本，一个一个横向堆积起来，这里我将 $[z^{(1)}z^{(2)}\dots z^{(m)}]$ 定义为大写的 Z ，你用小写 z 表示并将它们横向排在一起。所以当你将不同训练样本对应的小写 x 横向堆积在一起时得到大写变量 X 并且将小写变量也用相同方法处理，将它们横向堆积起来，你就得到大写变量 Z 。结果发现，为了计算 $W^TX+[bb\dots b]$ ，**numpy** 命令是$\boldsymbol{Z=np.dot(w.T,X)+b}$。这里在 **Python** 中有一个巧妙的地方，这里 b 是一个实数，或者你可以说是一个 1×1 矩阵，只是一个普通的实数。但是当你将这个向量加上这个实数时，**Python** 自动把这个实数 b 扩展成一个 $1\times m$ 的行向量。所以这种情况下的操作似乎有点不可思议，它在 **Python** 中被称作**广播(brosdcasting)**，目前你不用对此感到顾虑，我们将在下一个视频中进行进一步的讲解。话说回来它只用一行代码，用这一行代码，你可以计算大写的 Z，而大写 Z 是一个包含所有小写$z^{(1)}$ 到 $z^{(m)}$ 的 $1\times m$ 的矩阵。这就是 Z 的内容，关于变量 a 又是如何呢？

我们接下来要做的就是找到一个同时计算 $[a^{(1)}a^{(2)}\dots a^{(m)}]$ 的方法。就像把小写 x 堆积起来得到大写 X 和横向堆积小写 z 得到大写 Z 一样，堆积小写变量 a 将形成一个新的变量，我们将它定义为大写 A。在编程作业中，你将看到怎样用一个向量在 **sigmoid** 函数中进行计算。所以 **sigmoid** 函数中输入大写 Z 作为变量并且非常高效地输出大写 A。你将在编程作业中看到它的细节。

总结一下，在这张幻灯片中我们已经看到，不需要 **for** 循环，利用 m 个训练样本一次性计算出小写 z 和小写 a，用一行代码即可完成。

```
Z = np.dot(w.T,X) + b
```

这一行代码：$A=[a^{(1)}a^{(2)}\dots a^{(m)}]=\sigma(Z)$ ，通过恰当地运用σ一次性计算所有 a。这就是在同一时间内你如何完成一个所有 m 个训练样本的前向传播向量化计算。

2.14 向量化 logistic 回归的梯度输出（Vectorizing Logistic Regression's Gradient）

如何向量化计算的同时，对整个训练集预测结果a，这是我们之前已经讨论过的内容。在本次视频中我们将学习如何向量化地计算m个训练数据的梯度，本次视频的重点是如何**同时**计算 m 个数据的梯度，并且实现一个非常高效的逻辑回归算法**(Logistic Regression)**。

之前我们在讲梯度计算的时候，列举过几个例子， $dz^{(1)} = a^{(1)} - y^{(1)}$，$dz^{(2)} = a^{(2)} - y^{(2)}$ ……等等一系列类似公式。现在，对 m个训练数据做同样的运算，我们可以定义一个新的变量 $dZ = [dz^{(1)}, dz^{(2)} \dots dz^{(m)}]$ ，所有的 dz 变量横向排列，因此，**dZ 是一个 $1 \times m$ 的矩阵**，或者说，一个 m 维行向量。在之前的幻灯片中，我们已经知道如何计算A，即$[a^{(1)}, a^{(2)} \dots a^{(m)}]$,我们需要找到这样的一个行向量 $Y = [y^{(1)} y^{(2)} \dots y^{(m)}]$ ，由此，我们可以这样计算 $dZ = A - Y = [a^{(1)} - y^{(1)} a^{(2)} - y^{(2)} \dots a^{(m)} - y^{(m)}]$，不难发现第一个元素就是 $dz^{(1)}$，第二个元素就是 $dz^{(2)}$ ……所以我们现在仅需一行代码，就可以同时完成这所有的计算。

在之前的实现中，我们已经去掉了一个 **for** 循环，但我们仍有一个遍历训练集的循环，如下所示：

$$dw = 0$$

$$dw += x^{(1)} * dz^{(1)}$$

$$dw += x^{(2)} * dz^{(2)}$$

……

$$dw += x^{(m)} * dz^{(m)}$$

$$dw = \frac{dw}{m}$$

$$db = 0$$

$$db += dz^{(1)}$$

$$db += dz^{(2)}$$

……

$$db += dz^{(m)}$$

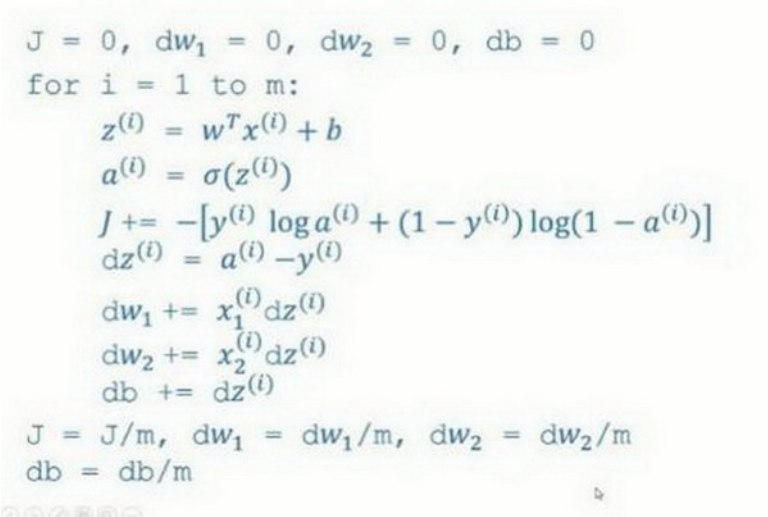
$$db = \frac{db}{m}$$

上述（伪）代码就是我们在之前实现中做的，我们已经去掉了一个 **for** 循环，但用上述

方法计算 dw 仍然需要一个循环遍历训练集，我们现在要做的就是将其向量化！

首先我们来看 db，不难发现 $db = \frac{1}{m}\sum_{i=1}^{m} d\,z^{(i)}$ ， 之前的讲解中，我们知道所有的$dz^{i)}$已经组成一个行向量 dZ了，所以在 **Python** 中，我们很容易地想到$db = \frac{1}{m} * np.sum(dZ)$；接下来看$dw$，我们先写出它的公式 $dw = \frac{1}{m} * X * dz^T$ 其中，X 是一个行向量。因此展开后 $dw = \frac{1}{m} * (x^{(1)}dz^{(1)} + x^{(2)}dz^{(2)} + \ldots + x^{m}dz^{m})$ 。因此我们可以仅用两行代码进行计算：$db = \frac{1}{m} * np.sum(dZ)$， $dw = \frac{1}{m} * X * dz^T$。这样，我们就避免了在训练集上使用 for 循环。

现在，让我们回顾一下，看看我们之前怎么实现的逻辑回归，可以发现，没有向量化是非常低效的，如下图所示代码：



我们的目标是不使用 **for** 循环，而是向量，我们可以这么做：

$$Z = w^T X + b = np.dot(w.T, X) + b$$

$$A = \sigma(Z)$$

$$dZ = A - Y$$

$$dw = \frac{1}{m} * X * dz^T$$

$$db = \frac{1}{m} * np.sum(dZ)$$

$$w := w - a * dw$$

$$b := b - a * db$$

2.15 Python 中的广播（Broadcasting in Python）

这是一个不同食物(每 100g)中不同营养成分的卡路里含量表格，表格为 3 行 4 列，列表示不同的食物种类，从左至右依次为苹果，牛肉，鸡蛋，土豆。行表示不同的营养成分，从上到下依次为碳水化合物，蛋白质，脂肪。

Calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

那么，我们现在想要计算不同食物中不同营养成分中的卡路里百分比。

现在计算苹果中的碳水化合物卡路里百分比含量，首先计算苹果（100g）中三种营养成分卡路里总和 56+1.2+1.8 = 59，然后用 56/59 = 94.9%算出结果。

可以看出苹果中的卡路里大部分来自于碳水化合物，而牛肉则不同。

对于其他食物，计算方法类似。首先，按列求和，计算每种食物中（100g）三种营养成分总和，然后分别用不用营养成分的卡路里数量除以总和，计算百分比。

那么，能否不使用 for 循环完成这样的一个计算过程呢？

假设上图的表格是一个 3 行 4 列的矩阵A，记为 $A_{3\times4}$，接下来我们要使用 **Python** 的 **numpy** 库完成这样的计算。我们打算使用两行代码完成，第一行代码对每一列进行求和，第二行代码分别计算每种食物每种营养成分的百分比。

在 **jupyter notebook** 中输入如下代码，按 shift+Enter 运行，输出如下。

In [6]:
```
1 import numpy as np
2
3 A = np.array([[56.0, 0.0, 4.4, 68.0],
4               [1.2, 104.0, 52.0, 8.0],
5               [1.8, 135.0, 99.0, 0.9]])
6
7 print(A)
```

```
[[  56.     0.     4.4   68. ]
 [   1.2  104.    52.     8. ]
 [   1.8  135.    99.     0.9]]
```

下面使用如下代码计算每列的和，可以看到输出是每种食物(100g)的卡路里总和。

```
In [7]:
1 cal = A.sum(axis=0)
2 print(cal)
```

```
[ 59.    239.    155.4    76.9]
```

其中 sum 的参数 axis=0 表示求和运算按列执行，之后会详细解释。

接下来计算百分比，这条指令将 3×4的矩阵A除以一个1×4的矩阵，得到了一个 3×4 的结果矩阵，这个结果矩阵就是我们要求的百分比含量。

```
In [8]:
1 percentage = 100*A/cal.reshape(1,4)
2 print(percentage)
```

```
[[ 94.91525424   0.           2.83140283  88.42652796]
 [  2.03389831  43.51464435  33.46203346  10.40312094]
 [  3.05084746  56.48535565  63.70656371   1.17035111]]
```

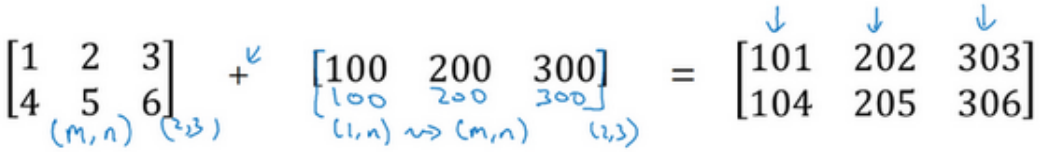
下面再来解释一下 A.sum(axis = 0)中的参数 axis。**axis 用来指明将要进行的运算是沿着哪个轴执行，在 numpy 中，0 轴是垂直的，也就是列，而 1 轴是水平的，也就是行。**

而第二个 A/cal.reshape(1,4)指令则调用了 **numpy** 中的广播机制。这里使用 3×4 的矩阵A除以 1×4的矩阵cal。技术上来讲，其实并不需要再将矩阵cal reshape(重塑)成 1×4，因为矩阵cal本身已经是 1×4了。但是当我们写代码时不确定矩阵维度的时候，通常会对矩阵进行重塑来确保得到我们想要的列向量或行向量。重塑操作 reshape 是一个常量时间的操作，时间复杂度是$O(1)$，它的调用代价极低。

那么一个 3×4 的矩阵是怎么和 1×4的矩阵做除法的呢？让我们来看一些更多的广播的例子。

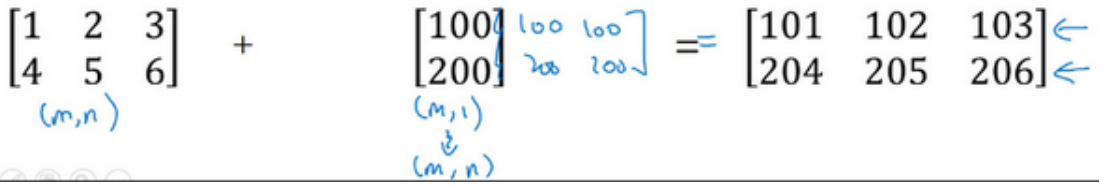
在 **numpy** 中，**当一个 4×1的列向量与一个常数做加法时，实际上会将常数扩展为一个 4×1的列向量，然后两者做逐元素加法。**结果就是右边的这个向量。这种广播机制对于行向量和列向量均可以使用。

再看下一个例子。



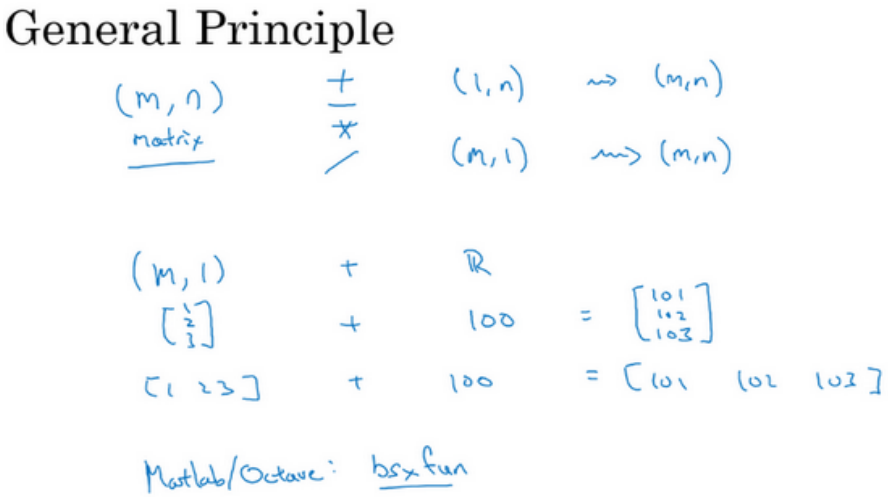
用一个 2×3的矩阵和一个 1×3 的矩阵相加，其泛化形式是 $m\times n$ 的矩阵和 $1\times n$ 的矩阵相加。在执行加法操作时，其实是将 $1\times n$ 的矩阵复制成为 $m\times n$ 的矩阵，然后两者做逐元素加法得到结果。针对这个具体例子，相当于在矩阵的第一列加 100，第二列加 200，第三列加 300。

下面是最后一个例子



这里相当于是一个 $m\times n$ 的矩阵加上一个 $m\times1$ 的矩阵。在进行运算时，会先将 $m\times1$ 矩阵水平复制 n 次，变成一个 $m\times n$ 的矩阵，然后再执行逐元素加法。

广播机制的一般原则如下：



这里我先说一下我本人对 **numpy** 广播机制的理解，再解释上面这张幻灯片。

首先是 **numpy** 广播机制

如果两个数组的后缘维度的轴长度相符或其中一方的轴长度为 1，则认为它们是广播兼容的。广播会在缺失维度和轴长度为 1 的维度上进行。

后缘维度的轴长度：`A.shape[-1]` 即矩阵维度元组中的最后一个位置的值

对于视频中卡路里计算的例子，矩阵 $A_{3,4}$ 后缘维度的轴长度是 4，而矩阵 $cal_{1,4}$ 的后缘维度也是 4，则他们满足后缘维度轴长度相符，可以进行广播。广播会在轴长度为 1 的维

度进行，轴长度为 1 的维度对应 `axis=0`，即垂直方向，矩阵 $\text{cal}_{1,4}$ 沿 `axis=0`(垂直方向)复制成为 $\text{cal\_temp}_{3,4}$ ，之后两者进行逐元素除法运算。

现在解释上图中的例子

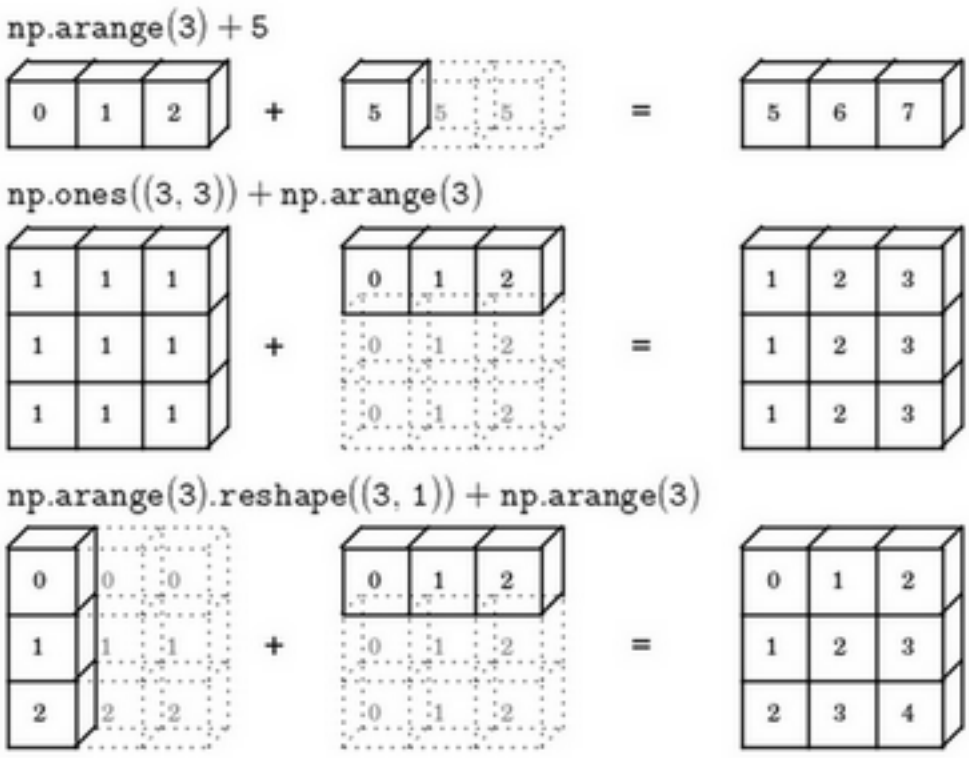
矩阵 $A_{m,n}$ 和矩阵 $B_{1,n}$ 进行四则运算，后缘维度轴长度相符，可以广播，广播沿着轴长度为 1 的轴进行，即 $B_{1,n}$ 广播成为 $B_{m,n}{}'$ ，之后做逐元素四则运算。

矩阵 $A_{m,n}$ 和矩阵 $B_{m,1}$ 进行四则运算，后缘维度轴长度不相符，但其中一方轴长度为 1，可以广播，广播沿着轴长度为 1 的轴进行，即 $B_{m,1}$ 广播成为 $B_{m,n}{}'$ ，之后做逐元素四则运算。

矩阵 $A_{m,1}$ 和常数R 进行四则运算，后缘维度轴长度不相符，但其中一方轴长度为 1，可以广播，广播沿着缺失维度和轴长度为 1 的轴进行，缺失维度就是 `axis=0`,轴长度为 1 的轴是 `axis=1`，即R广播成为 $B_{m,1}{}'$ ，之后做逐元素四则运算。

最后，对于 **Matlab/Octave** 有类似功能的函数 `bsxfun`。

总结一下 `broadcasting`，可以看看下面的图：



2.16 关于 python _ numpy 向量的说明（A note on python or numpy vectors）参考视频：

Python 的特性允许你使用广播（**broadcasting**）功能，这是 **Python** 的 **numpy** 程序语言库中最灵活的地方。而我认为这是程序语言的优点，也是缺点。优点的原因在于它们创造出语言的表达性，**Python** 语言巨大的灵活性使得你仅仅通过一行代码就能做很多事情。但是这也是缺点，由于广播巨大的灵活性，有时候你对于广播的特点以及广播的工作原理这些细节不熟悉的话，你可能会产生很细微或者看起来很奇怪的 bug。例如，如果你将一个列向量添加到一个行向量中，你会以为它报出维度不匹配或类型错误之类的错误，但是实际上你会得到一个行向量和列向量的求和。

在 **Python** 的这些奇怪的影响之中，其实是有一个内在的逻辑关系的。但是如果对 Python 不熟悉的话，我就曾经见过的一些学生非常生硬、非常艰难地去寻找 bug。所以我在这里想做的就是分享给你们一些技巧，这些技巧对我非常有用，它们能消除或者简化我的代码中所有看起来很奇怪的 bug。同时我也希望通过这些技巧，你也能更容易地写没有 bug 的 **Python** 和 **numpy** 代码。

为了演示 Python-numpy 的一个容易被忽略的效果，特别是怎样在 Python-numpy 中构造向量，让我来做一个快速示范。首先设置$a = np.random.randn(5)$，这样会生成存储在数组 a 中的 5 个高斯随机数变量。之后输出 a，从屏幕上可以得知，此时 a 的 **shape**（形状）是一个$(5,)$的结构。**这在 Python 中被称作一个一维数组。它既不是一个行向量也不是一个列向量，这也导致它有一些不是很直观的效果。**举个例子，如果我输出一个转置阵，最终结果它会和a看起来一样，所以a和a的转置阵最终结果看起来一样。而如果我输出a和a的转置阵的内积，你可能会想：a乘以a的转置返回给你的可能会是一个矩阵。但是如果我这样做，你只会得到一个数。

```
In [1]: import numpy as np

        a = np.random.randn(5)

In [2]: print(a)
        [ 0.50290632 -0.29691149  0.95429604 -0.82126861 -1.46269164]

In [3]: print(a.shape)
        (5,)

In [4]: print(a.T)
        [ 0.50290632 -0.29691149  0.95429604 -0.82126861 -1.46269164]

In [5]: print(np.dot(a,a.T))
        4.06570109321
```

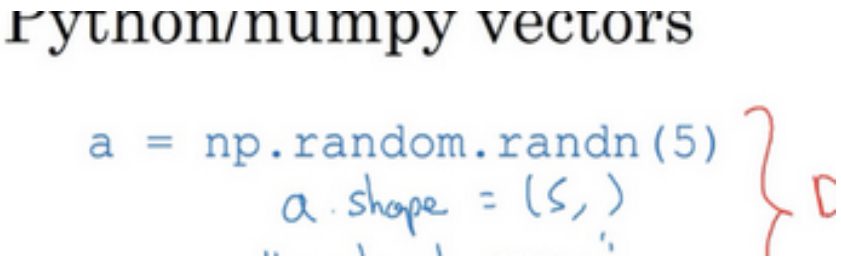
所以我建议当你编写神经网络时，不要在它的 **shape** 是$(5,)$还是$(n,)$或者一维数组时使用数据结构。相反，如果你设置 a 为(5,1)，那么这就将置于 5 行 1 列向量中。在先前的操作里 a 和 a 的转置看起来一样，而现在这样的 a 变成一个新的 a 的转置，并且它是一个行向量。请注意一个细微的差别，在这种数据结构中，当我们输出 a 的转置时有两对方括号，而之前只有一对方括号，所以这就是 1 行 5 列的矩阵和一维数组的差别。

```
In [4]: print(a.T)
        [ 0.50290632 -0.29691149  0.95429604 -0.82

In [5]: print(np.dot(a,a.T))
        4.06570109321

In [6]: a = np.random.randn(5,1)
        print(a)
        [[-0.0967311 ]
         [-2.38617377]
         [-0.3243588 ]
         [-0.96216349]
         [ 0.54410384]]

In [7]: print(a.T)
        [[-0.0967311  -2.38617377 -0.3243588  -0.9
```

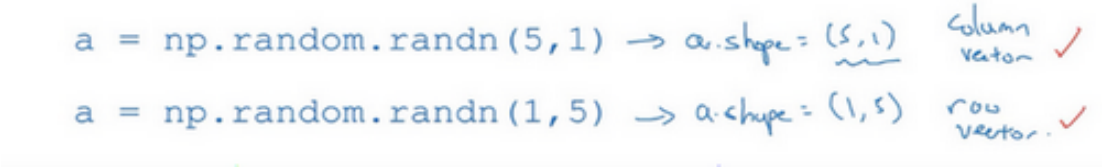
如果你输出 a 和 a 的转置的乘积，然后会返回给你一个向量的外积，是吧？所以这两个向量的外积返回给你的是一个矩阵。

```
3]: print(np.dot(a,a.T))

[[ 0.00935691  0.23081721  0.03137558  0
 [ 0.23081721  5.69382526  0.77397645  2
 [ 0.03137558  0.77397645  0.10520863  0
 [ 0.09307113  2.29588928  0.31208619  0
```

就我们刚才看到的，再进一步说明。首先我们刚刚运行的命令是这个 $(a = np.random.randn(5))$，而且它生成了一个数据结构 $(a.shape)$，**$a.shape$是$(5,)$，一个有趣的东西。这被称作 a 的一维数组，同时这也是一个非常有趣的数据结构。**它不像行向量和列向量那样表现的很一致，这也让它的一些影响不那么明显。所以我建议，当你在编程练习或者在执行逻辑回归和神经网络时，你不需要使用这些一维数组。



相反，如果你每次创建一个数组，你都得让它成为一个列向量，产生一个(5,1)向量或者你让它成为一个行向量，那么你的向量的行为可能会更容易被理解。所以在这种情况下，$a.shape$等同于(5,1)。这种表现很像 a，但是实际上却是一个列向量。同时这也是为什么当它是一个列向量的时候，你能认为这是矩阵(5,1)；同时这里 $a.shape$ 将要变成(1,5)，这就像行向量一样。所以当你需要一个向量时，我会说用这个或那个**(column vector or row vector)**，但绝不会是一维数组。

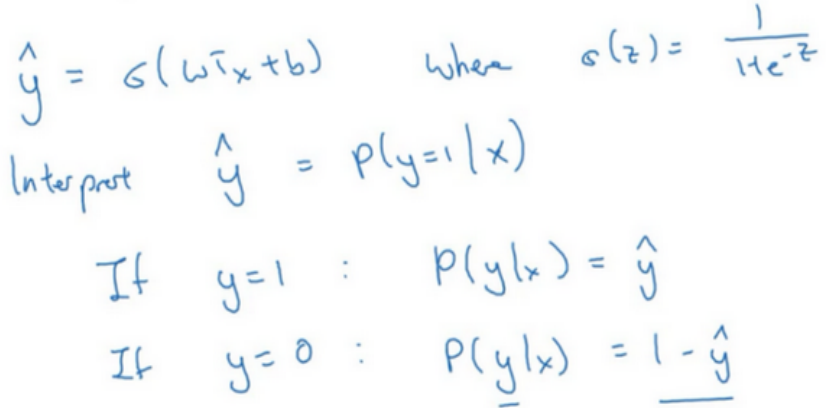


我写代码时还有一件经常做的事，那就是**如果我不完全确定一个向量的维度(dimension)，我经常会扔进一个断言语句(assertion statement)**。像这样，去确保在这种情况下是一个(5,1)向量，或者说是一个列向量。这些断言语句实际上是要去执行的，并且它们也会有助于为你的代码提供信息。所以不论你要做什么，不要犹豫直接插入断言语句。如果你不小心以一维数组来执行，你也能够重新改变数组维数 $a = reshape$，表明一个(5,1)数组或者一个(1,5)数组，以致于它表现更像列向量或行向量。

因此，要去简化你的代码，而且不要使用一维数组。总是使用 $n \times 1$ 维矩阵（基本上是列向量），或者 $1 \times n$ 维矩阵（基本上是行向量），这样你可以减少很多 **assert** 语句来节省核矩阵和数组的维数的时间。另外，为了确保你的矩阵或向量所需要的维数时，不要羞于 **reshape** 操作。

2.18 （选修）logistic 损失函数的解释（Explanation of logistic regression cost function）

在前面的视频中，我们已经分析了逻辑回归的损失函数表达式，在这节选修视频中，我将给出一个简洁的证明来说明逻辑回归的损失函数为什么是这种形式。



回想一下，在逻辑回归中，需要预测的结果$\hat{y}$,可以表示为$\hat{y} = \sigma(w^T x + b)$，$\sigma$是我们熟悉的$S$型函数 $\sigma(z) = \sigma(w^T x + b) = \frac{1}{1+e^{-z}}$ 。我们约定 $\hat{y} = p(y = 1|x)$ ，即算法的输出$\hat{y}$ 是给定训练样本 x 条件下 y 等于 1 的概率。

换句话说，如果$y = 1$，在给定训练样本 x 条件下$\mathrm{y} = \hat{y}$；

反过来说，如果$y = 0$，在给定训练样本x条件下 $(y = 1 - \hat{y})$，

因此，如果 $\hat{y}$ 代表 $y = 1$ 的概率，那么$1 - \hat{y}$就是 $y = 0$的概率。

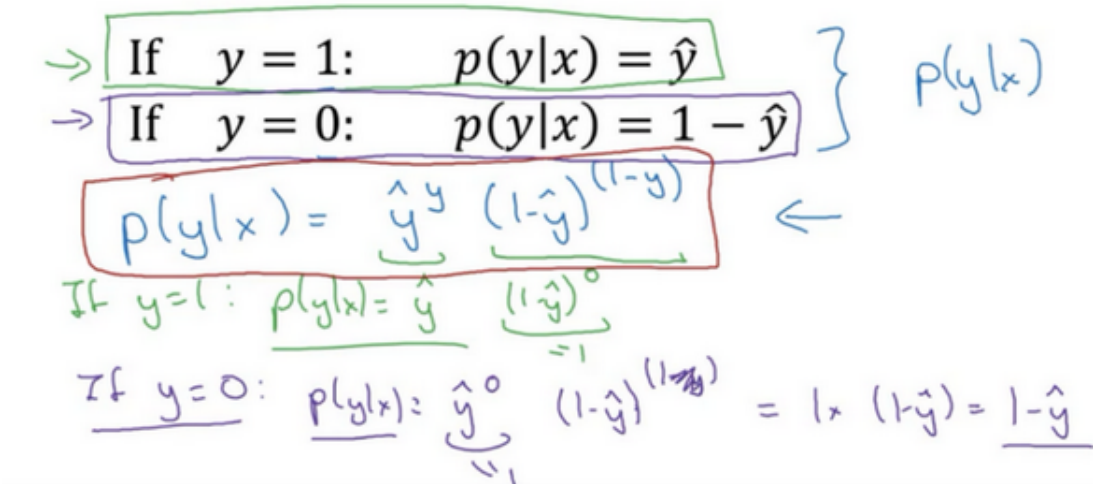
接下来，我们就来分析这两个条件概率公式。

If $y = 1$: $p(y|x) = \hat{y}$
If $y = 0$: $p(y|x) = 1 - \hat{y}$

这两个条件概率公式定义形式为 $p(y|x)$并且代表了 $y = 0$ 或者 $y = 1$ 这两种情况，我们可以将这两个公式合并成一个公式。需要指出的是我们讨论的是二分类问题的损失函数，因此，y的取值只能是 0 或者 1。上述的两个条件概率公式可以合并成如下公式：

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{(1-y)}$$

接下来我会解释为什么可以合并成这种形式的表达式：$(1 - \hat{y})$的$(1 - y)$次方这行表达式包含了上面的两个条件概率公式，我来解释一下为什么。



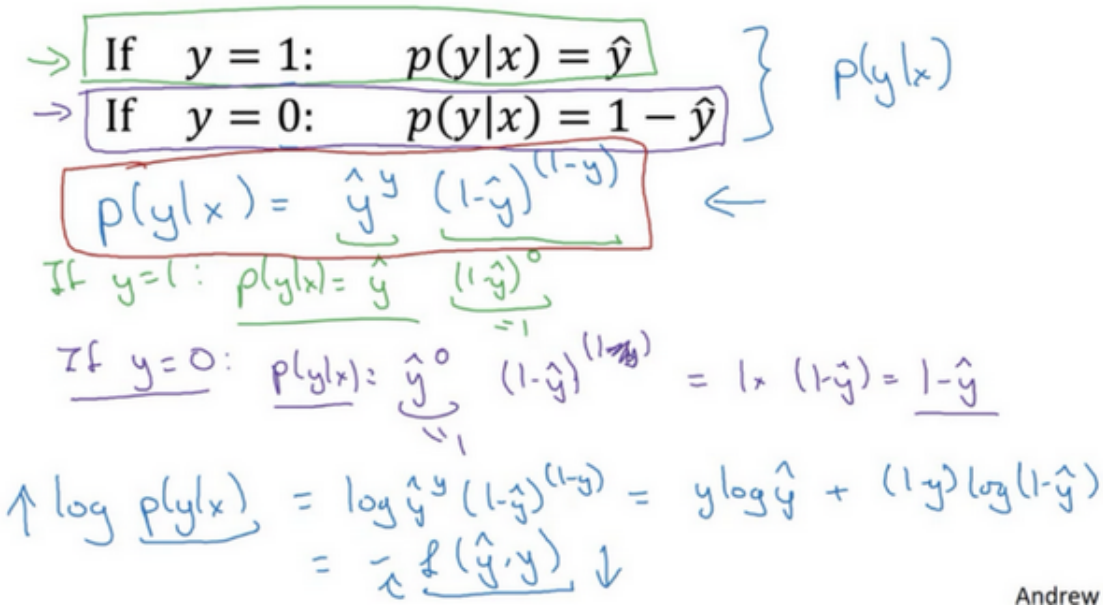
第一种情况，假设 $y = 1$，由于$y = 1$，那么$(\hat{y})^y = \hat{y}$，因为 $\hat{y}$的 1 次方等于$\hat{y}$，$1 - (1 - \hat{y})^{(1-y)}$的指数项$(1 - y)$等于 0，由于任何数的 0 次方都是 1，$\hat{y}$乘以 1 等于$\hat{y}$。因此当$y = 1$时 $p(y|x) = \hat{y}$（图中绿色部分）。

第二种情况，当 $y = 0$ 时 $p(y|x)$ 等于多少呢? 假设$y = 0$，$\hat{y}$的y次方就是 $\hat{y}$ 的 0 次方，任何数的 0 次方都等于 1，因此 $p(y|x) = 1 \times (1 - \hat{y})^{1-y}$ ，前面假设 $y = 0$ 因此$(1 - y)$就等于 1，因此 $p(y|x) = 1 \times (1 - \hat{y})$。因此在这里当$y = 0$时，$p(y|x) = 1 - \hat{y}$。这就是这个公式(第二个公式，图中紫色字体部分)的结果。

因此，刚才的推导表明 $p(y|x) = \hat{y}^{(y)} (1 - \hat{y})^{(1-y)}$，就是 $p(y|x)$ 的完整定义。由于 log 函数是严格单调递增的函数，最大化 $log(p(y|x))$ 等价于最大化 $p(y|x)$ 并且地计算 $p(y|x)$ 的 log 对数，就是计算 $log(\hat{y}^{(y)} (1 - \hat{y})^{(1-y)})$ (其实就是将 $p(y|x)$ 代入)，通过对数函数化简为：

$$y log\, \hat{y} + (1 - y) log(1 - \hat{y})$$

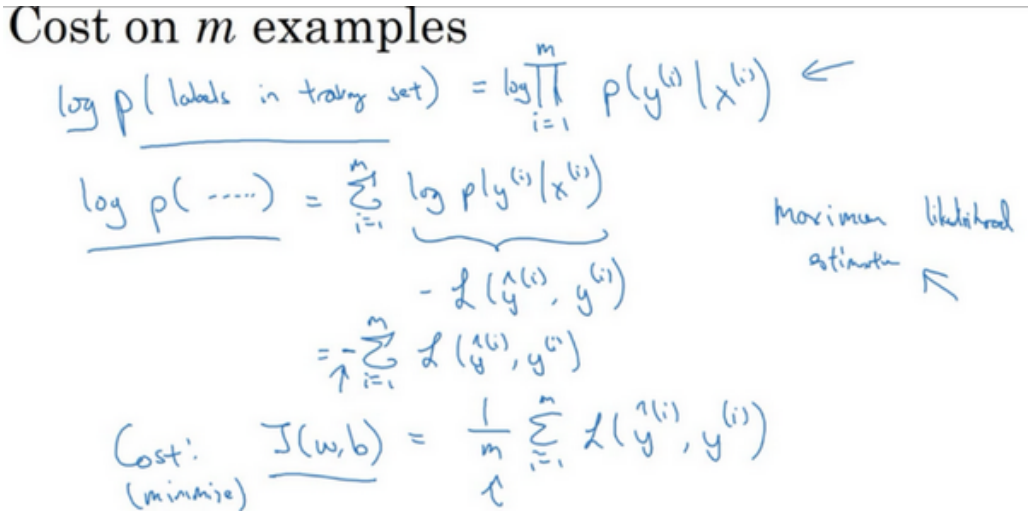
而这就是我们前面提到的损失函数的负数 $(-L(\hat{y}, y))$ ，前面有一个负号的原因是当你训练学习算法时需要算法输出值的概率是最大的（以最大的概率预测这个值），然而在逻辑回归中我们需要最小化损失函数，因此最小化损失函数与最大化条件概率的对数 $log(p(y|x))$ 关联起来了，因此这就是单个训练样本的损失函数表达式。



在 m个训练样本的整个训练集中又该如何表示呢，让我们一起来探讨一下。

让我们一起来探讨一下，整个训练集中标签的概率，更正式地来写一下。假设所有的训练样本服从同一分布且相互独立，也即独立同分布的，所有这些样本的联合概率就是每个样本概率的乘积:

$$P(\text{labels in training set}) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})。$$

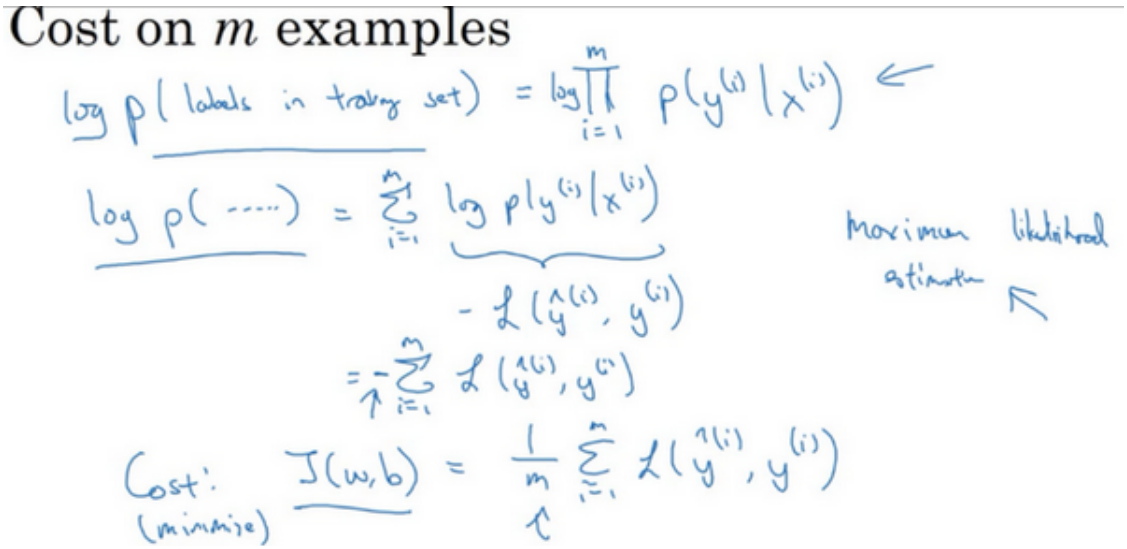


如果你想做最大似然估计，需要寻找一组参数，使得给定样本的观测值概率最大，但令这个概率最大化等价于令其对数最大化，在等式两边取对数：

$$logp(\text{labels in training set}) = log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} logP(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} -L(\hat{y}^{(i)}, y^{(i)})$$

在统计学里面，有一个方法叫做最大似然估计，即求出一组参数，使这个式子取最大值，也就是说，使得这个式子取最大值，$\sum_{i=1}^{m} -L(\hat{y}^{(i)}, y^{(i)})$，可以将负号移到求和符号的外面，

$-\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$，这样我们就推导出了前面给出的 logistic 回归的成本函数$J(w,b) = \sum_{i=1}^{m} L(\hat{y}^{(i)}, \hat{y^{(i)}})$。



由于训练模型时，目标是让成本函数最小化，所以我们不是直接用最大似然概率，要去掉这里的负号，最后为了方便，可以对成本函数进行适当的缩放，我们就在前面加一个额外的常数因子$\frac{1}{m}$，即:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})。$$

总结一下，为了最小化成本函数$J(w,b)$，我们从 **logistic** 回归模型的最大似然估计的角度出发，假设训练集中的样本都是独立同分布的条件下。尽管这节课是选修性质的，但还是感谢观看本节视频。我希望通过本节课您能更好地明白逻辑回归的损失函数，为什么是那种形式，明白了损失函数的原理，希望您能继续完成课后的练习，前面课程的练习以及本周的测验，在课后的小测验和编程练习中,祝您好运。