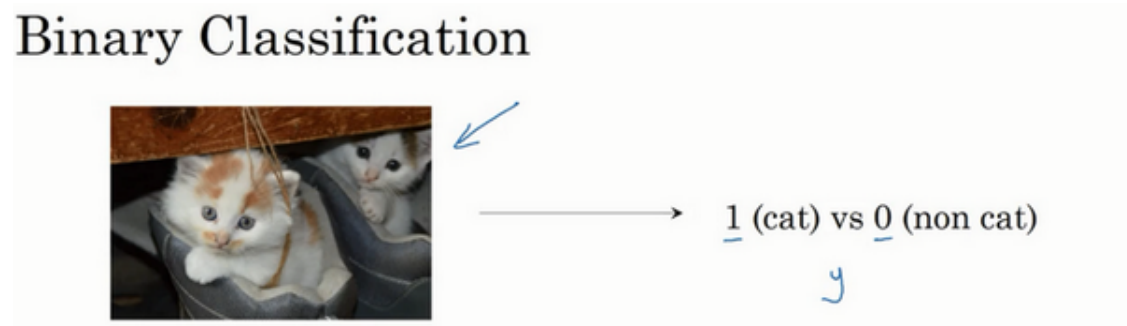


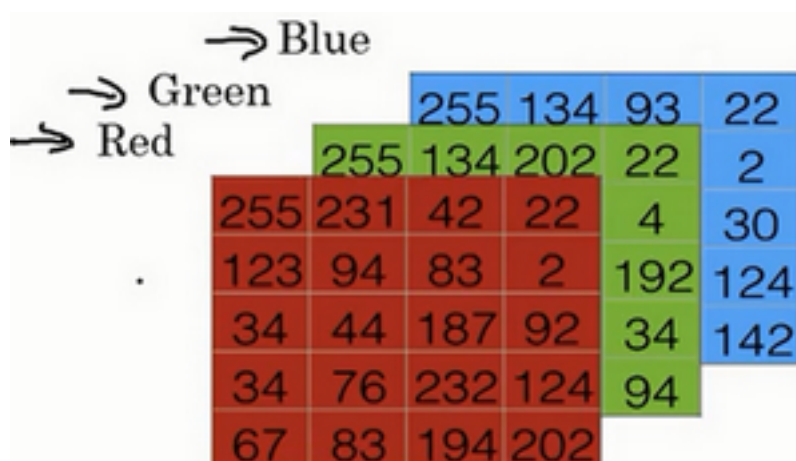
2.1 二分类(Binary Classification)

在神经网络的计算中，通常先有一个叫做前向暂停(forward pause)或叫做前向传播(foward propagation)的步骤，接着有一个叫做反向暂停(backward pause) 或叫做反向传播(backward propagation)的步骤。

逻辑回归是一个用于二分类(binary classification)的算法。首先我们从一个问题开始说起，这里有一个二分类问题的例子，假如你有一张图片作为输入，比如这只猫，如果识别这张图片为猫，则输出标签 1 作为结果；如果识别出不是猫，那么输出标签 0 作为结果。现在我们可以用字母 y 来表示输出的结果标签，如下图所示：

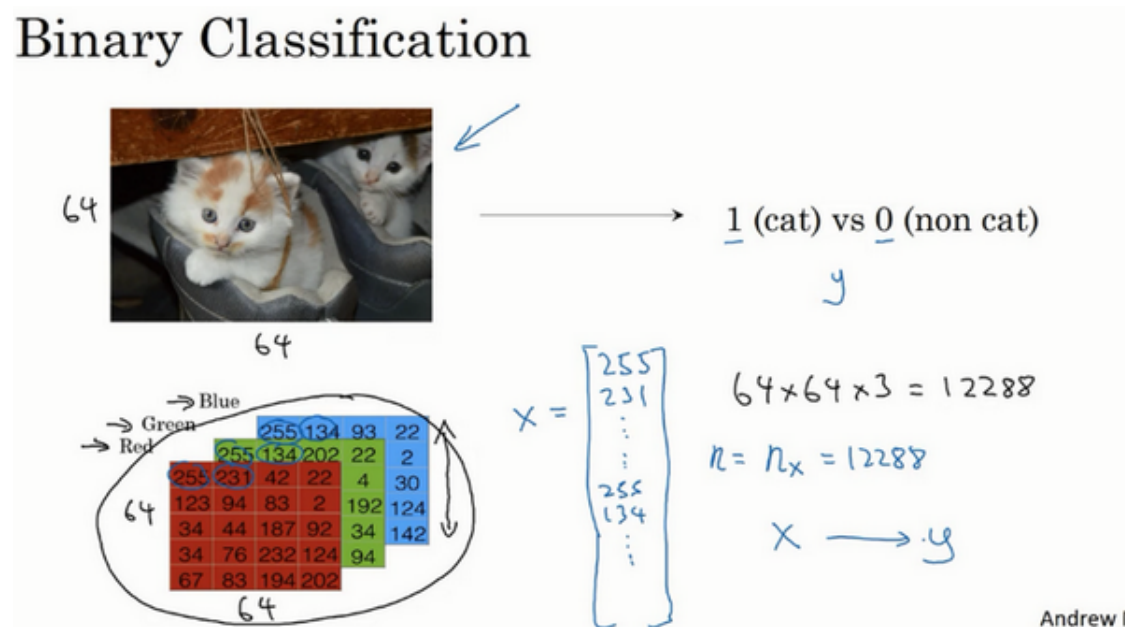


我们来看看一张图片在计算机中是如何表示的，为了保存一张图片，需要保存三个矩阵，它们分别对应图片中的红、绿、蓝三种颜色通道，如果你的图片大小为 64×64 像素，那么你就有三个规模为 64×64 的矩阵，分别对应图片中红、绿、蓝三种像素的强度值。为了便于表示，这里我画了三个很小的矩阵，注意它们的规模为 5×4 而不是 64×64 ，如下图所示：



为了把这些像素值放到一个特征向量中，我们需要把这些像素值提取出来，然后放入一个特征向量 x 。为了把这些像素值转换为特征向量 x ，我们需要像下面这样定义一个特征向量 x 来表示这张图片，我们把所有的像素都取出来，例如 255、231 等等，直到取完所有的

红色像素，接着最后是 255、134、...、255、134 等等，直到得到一个特征向量，把图片中所有的红、绿、蓝像素值都列出来。如果图片的大小为 64×64 像素，那么向量 x 的总维度，将是 64 乘以 64 乘以 3，这是三个像素矩阵中像素的总量。在这个例子中结果为 12,288。现在我们用 $n_x = 12,288$ ，来表示输入特征向量的维度，有时候为了简洁，我会直接用小写的 n 来表示输入特征向量 x 的维度。所以在二分类问题中，我们的目标就是习得一个分类器，它以图片的特征向量作为输入，然后预测输出结果 y 为 1 还是 0，也就是预测图片中是否有猫：



接下来我们说明一些在余下课程中，需要用到的一些符号。

符号定义：

x ：表示一个 n_x 维数据，为输入数据，维度为 $(n_x, 1)$ ；

y ：表示输出结果，取值为 $(0,1)$ ；

$(x^{(i)}, y^{(i)})$ ：表示第 i 组数据，可能是训练数据，也可能是测试数据，此处默认为训练数据；

$X = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$ ：表示所有的训练数据集的输入值，放在一个 $n_x \times m$ 的矩阵中，其中 m 表示样本数目；

$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$ ：对应表示所有训练数据集的输出值，维度为 $1 \times m$ 。

用一对 (x, y) 来表示一个单独的样本， x 代表 n_x 维的特征向量， y 表示标签(输出结果)只能为 0 或 1。而训练集将由 m 个训练样本组成，其中 $(x^{(1)}, y^{(1)})$ 表示第一个样本的输入和输出， $(x^{(2)}, y^{(2)})$ 表示第二个样本的输入和输出，直到最后一个样本 $(x^{(m)}, y^{(m)})$ ，然后所有的这些一起表示整个训练集。有时候为了强调这是训练样本的个数，会写作 M_{train} ，当涉及到测试集的时候，我们会使用 M_{test} 来表示测试集的样本数，所以这是测试集的样本数：

$$(x, y) \quad x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$$

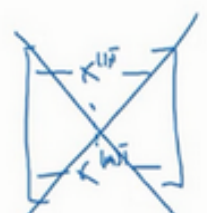
$$m \text{ training examples: } \{(\underline{x}^{(1)}, y^{(1)}), (\underline{x}^{(2)}, y^{(2)}), \dots, (\underline{x}^{(m)}, y^{(m)})\}$$

$$M = M_{\text{train}} \quad M_{\text{test}} = \# \text{ test examples.}$$

最后为了能把训练集表示得更紧凑一点，我们会定义一个矩阵用大写 X 的表示，它由输入向量 $x^{(1)}$ 、 $x^{(2)}$ 等组成，如下图放在矩阵的列中，所以现在我们把 $x^{(1)}$ 作为第一列放在矩阵中， $x^{(2)}$ 作为第二列， $x^{(m)}$ 放到第 m 列，然后我们就得到了训练集矩阵 X 。所以这个矩阵有 m 列， m 是训练集的样本数量，然后这个矩阵的高度记为 n_x ，注意有时候可能因为其他某些原因，矩阵 X 会由训练样本按照行堆叠起来而不是列，如下图所示： $x^{(1)}$ 的转置直到 $x^{(m)}$ 的转置，但是在实现神经网络的时候，使用左边的这种形式，会让整个实现的过程变得更加简单：

$$m \text{ training examples: } \{(\underline{x}^{(1)}, y^{(1)}), (\underline{x}^{(2)}, y^{(2)}), \dots, (\underline{x}^{(m)}, y^{(m)})\}$$

$$M = M_{\text{train}} \quad M_{\text{test}} = \# \text{ test examples.}$$

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


现在来简单温习一下： X 是一个规模为 n_x 乘以 m 的矩阵，当你用 **Python** 实现的时候，你会看到 **`X.shape`**，这是一条 **Python** 命令，用于显示矩阵的规模，即 **`X.shape`** 等于 (n_x, m) ， X 是一个规模为 n_x 乘以 m 的矩阵。所以综上所述，这就是如何将训练样本（输入向量 X 的集合）表示为一个矩阵。

那么输出标签 y 呢？同样的道理，为了能更加容易地实现一个神经网络，将标签 y 放在列中将会使得后续计算非常方便，所以我们定义大写的 Y 等于 $y^{(1)}, y^{(2)}, \dots, y^{(m)}$ ，所以在这里是一个规模为 1 乘以 m 的矩阵，同样地使用 **Python** 将表示为 **`Y.shape`** 等于 $(1, m)$ ，表示这是一个规模为 1 乘以 m 的矩阵。

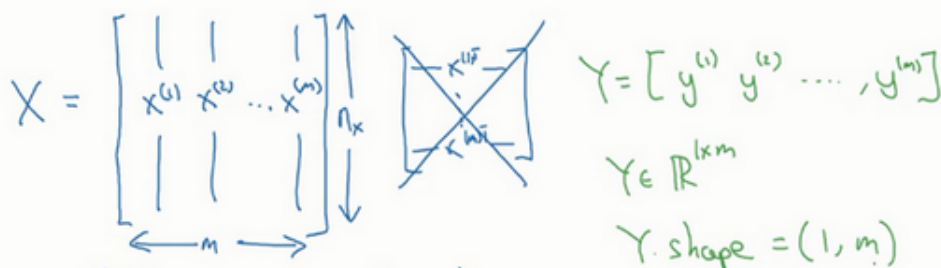
当你在后面的课程中实现神经网络的时候，你会发现，一个好的符号约定能够将不同训

NOTATION

$$(x, y) \quad x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$$

$$m \text{ training examples: } \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$$

$$M = M_{\text{train}} \quad M_{\text{test}} = \# \text{ test examples.}$$



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

$$Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]$$

$$Y \in \mathbb{R}^{1 \times m}$$

$$Y.\text{shape} = (1, m)$$

练样本的数据很好地组织起来。而我所说的数据不仅包括 x 或者 y 还包括之后你会看到的其他的量。将不同的训练样本的数据提取出来，然后就像刚刚我们对 x 或者 y 所做的那样，将他们堆叠在矩阵的列中，形成我们之后会在逻辑回归和神经网络上要用到的符号表示。如果有时候你忘了这些符号的意思，比如什么是 m ，或者什么是 n ，或者忘了其他一些东西，我们也会在课程的网站上放上符号说明，然后你可以快速地查阅每个具体的符号代表什么意思，好了，我们接着到下一个视频，在下一个视频中，我们将以逻辑回归作为开始。

备注：附录里也写了符号说明。

2.2 逻辑回归(Logistic Regression)

在这个视频中，我们会重温逻辑回归学习算法，该算法适用于二分类问题，本节将主要介绍逻辑回归的 **Hypothesis Function**（假设函数）。

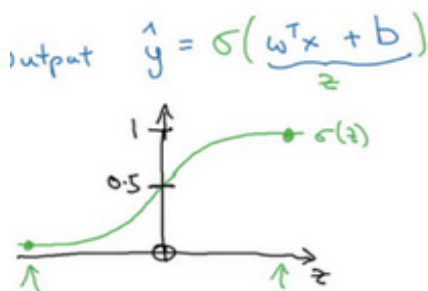
对于二元分类问题来讲，给定一个输入特征向量 X ，它可能对应一张图片，你想识别这张图片识别看它是否是一只猫或者不是一只猫的图片，你想要一个算法能够输出预测，你只能称之为 \hat{y} ，也就是你对实际值 y 的估计。更正式地来说，你想让 \hat{y} 表示 y 等于1的一种可能性或者是机会，前提条件是给定了输入特征 X 。换句话说，如果 X 是我们在上个视频看到的图片，你想让 \hat{y} 来告诉你这是一只猫的图片的机率有多大。在之前的视频中所说的， X 是一个 n_x 维的向量（相当于有 n_x 个特征的特征向量）。我们用 w 来表示逻辑回归的参数，这也是一个 n_x 维向量（因为 w 实际上是特征权重，维度与特征向量相同），参数里面还有 b ，这是一个实数（表示偏差）。所以给出输入 x 以及参数 w 和 b 之后，我们怎样产生输出预测值 \hat{y} ，一件你可以尝试却不可行的事是让 $\hat{y} = w^T x + b$ 。

Logistic Regression

$$\begin{aligned} \text{Given } x, \text{ want } \hat{y} &= \frac{P(y=1|x)}{0 \leq \hat{y} \leq 1} \\ x &\in \mathbb{R}^{n_x} \\ \text{Parameters: } w &\in \mathbb{R}^{n_x}, b \in \mathbb{R}. \\ \text{Output } \hat{y} &= \sigma(w^T x + b) \end{aligned}$$

这时候我们得到的是一个关于输入 x 的线性函数，实际上这是你在做线性回归时所用到的，但是这对于二元分类问题来讲不是一个非常好的算法，因为你想让 \hat{y} 表示实际值 y 等于1的机率的话， \hat{y} 应该在0到1之间。这是一个需要解决的问题，因为 $w^T x + b$ 可能比1要大得多，或者甚至为一个负值。对于你想要的在0和1之间的概率来说它是没有意义的，因此在逻辑回归中，我们的输出应该是 \hat{y} 等于由上面得到的线性函数式子作为自变量的 **sigmoid** 函数中，公式如上图最下面所示，将线性函数转换为非线性函数。

下图是 **sigmoid** 函数的图像，如果我把水平轴作为 z 轴，那么关于 z 的 **sigmoid** 函数是这样的，它是平滑地从0走向1，让我在这里标记纵轴，这是0，曲线与纵轴相交的截距是0.5，这就是关于 z 的 **sigmoid** 函数的图像。我们通常都使用 z 来表示 $w^T x + b$ 的值。



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

If z large $\sigma(z) \approx \frac{1}{1+0}$

If z large negative number

$$\sigma(z) = \frac{1}{1 + e^{-z}} \approx \frac{1}{1 + \text{Big num}} \approx 0$$

关于 **sigmoid** 函数的公式是这样的, $\sigma(z) = \frac{1}{1+e^{-z}}$, 在这里 z 是一个实数, 这里要说明一些要注意的事情, 如果 z 非常大那么 e^{-z} 将会接近于 0, 关于 z 的 **sigmoid** 函数将会近似等于 1 除以 1 加上某个非常接近于 0 的项, 因为 e 的指数如果是个绝对值很大的负数的话, 这项将会接近于 0, 所以如果 z 很大的话那么关于 z 的 **sigmoid** 函数会非常接近 1。相反地, 如果 z 非常小或者说是一个绝对值很大的负数, 那么关于 e^{-z} 这项会变成一个很大的数, 你可以认为这是 1 除以 1 加上一个非常非常大的数, 所以这个就接近于 0。实际上你看到当 z 变成一个绝对值很大的负数, 关于 z 的 **sigmoid** 函数就会非常接近于 0, 因此当你实现逻辑回归时, 你的工作就是去让机器学习参数 w 以及 b 这样才使得 \hat{y} 成为对 $y = 1$ 这一情况的概率的一个很好的估计。

$$x_0 = 1, \quad x \in \mathbb{R}^{n_x+1}$$

$$\hat{y} = \sigma(\Theta^T x)$$

$$\Theta = \begin{bmatrix} \Theta_0 \\ \Theta_1 \\ \Theta_2 \\ \vdots \\ \Theta_{n_x} \end{bmatrix} \left\{ \begin{array}{l} b \leftarrow \\ w \leftarrow \end{array} \right.$$

2.3 逻辑回归的代价函数 (Logistic Regression Cost Function)

为什么需要代价函数:

为了训练逻辑回归模型的参数参数 w 和参数 b 我们, 需要一个代价函数, 通过训练代价函数来得到参数 w 和参数 b 。先看一下逻辑回归的输出函数:

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\text{Given } \{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}, \text{ want } \hat{y}^{(i)} \approx y^{(i)}.$$

为了让模型通过学习调整参数, 你需要给予一个 m 样本的训练集, 这会让你在训练集上找到参数 w 和参数 b , 来得到你的输出。

对训练集的预测值, 我们将它写成 \hat{y} , 我们更希望它会接近于训练集中的 y 值, 为了对上面的公式更详细的介绍, 我们需要说明上面的定义是对一个训练样本来说的, 这种形式也使用于每个训练样本, 我们使用这些带有圆括号的上标来区分索引和样本, 训练样本 i 所对应的预测值是 $y^{(i)}$, 是用训练样本的 $w^T x^{(i)} + b$ 然后通过 **sigmoid** 函数来得到, 也可以把 z 定义为 $z^{(i)} = w^T x^{(i)} + b$, 我们将使用这个符号 (i) 注解, 上标 (i) 来指明数据表示 x 或者 y 或者 z 或者其他数据的第 i 个训练样本, 这就是上标 (i) 的含义。

损失函数:

损失函数又叫做误差函数, 用来衡量算法的运行情况, **Loss function:** $L(\hat{y}, y)$.

我们通过这个 L 称为的损失函数, 来衡量预测输出值和实际值有多接近。一般我们用预测值和实际值的平方差或者它们平方差的一半, 但是通常在逻辑回归中我们不这么做, 因为当我们在学习逻辑回归参数的时候, 会发现我们的优化目标不是凸优化, 只能找到多个局部最优值, 梯度下降法很可能找不到全局最优值, 虽然平方差是一个不错的损失函数, 但是我们在逻辑回归模型中会定义另外一个损失函数。

我们在逻辑回归中用到的损失函数是: $L(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$

为什么要用这个函数作为逻辑损失函数? 当我们使用平方误差作为损失函数的时候, 你会想要让这个误差尽可能地小, 对于这个逻辑回归损失函数, 我们也想让它尽可能地小, 为了更好地理解这个损失函数怎么起作用, 我们举两个例子:

当 $y = 1$ 时损失函数 $L = -\log(\hat{y})$, 如果想要损失函数 L 尽可能得小, 那么 \hat{y} 就要尽可能大, 因为 **sigmoid** 函数取值 $[0,1]$, 所以 \hat{y} 会无限接近于 1。

当 $y = 0$ 时损失函数 $L = -\log(1 - \hat{y})$ ，如果想要损失函数 L 尽可能得小，那么 \hat{y} 就要尽可能小，因为 **sigmoid** 函数取值 $[0,1]$ ，所以 \hat{y} 会无限接近于 0。

在这门课中有很多的函数效果和现在这个类似，就是如果 y 等于 1，我们就尽可能让 \hat{y} 变大，如果 y 等于 0，我们就尽可能让 \hat{y} 变小。损失函数是在单个训练样本中定义的，它衡量的是算法在单个训练样本中表现如何，为了衡量算法在全部训练样本上的表现如何，我们需要定义一个算法的代价函数，算法的代价函数是对 m 个样本的损失函数求和然后除以 m ：

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

损失函数只适用于像这样的单个训练样本，而代价函数是参数的总代价，所以在训练逻辑回归模型时候，我们需要找到合适的 w 和 b ，来让代价函数 J 的总代价降到最低。根据我们对逻辑回归算法的推导及对单个样本的损失函数的推导和针对算法所选用参数的总代价函数的推导，结果表明逻辑回归可以看做是一个非常小的神经网络，在下一个视频中，我们会看到神经网络会做什么。

2.4 梯度下降法（Gradient Descent）

梯度下降法可以做什么？

在你测试集上，通过最小化代价函数（成本函数） $J(w, b)$ 来训练的参数 w 和 b ,

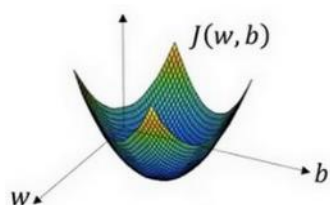
Gradient Descent

$$\text{Recap: } \hat{y} = \sigma(w^T x + b), \sigma(z) = \frac{1}{1+e^{-z}}$$

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

如图，在第二行给出和之前一样的逻辑回归算法的代价函数（成本函数）

梯度下降法的形象化说明



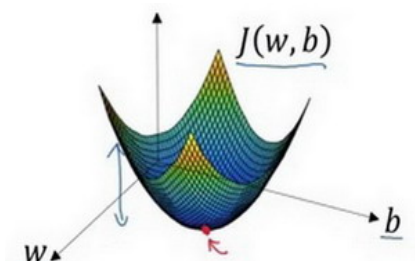
在这个图中，横轴表示你的空间参数 w 和 b ，在实践中， w 可以是更高的维度，但是为了更好地绘图，我们定义 w 和 b ，都是单一实数，代价函数（成本函数） $J(w, b)$ 是在水平轴 w 和 b 上的曲面，因此曲面的高度就是 $J(w, b)$ 在某一点的函数值。我们所做的就是找到使得代价函数（成本函数） $J(w, b)$ 函数值是最小值，对应的参数 w 和 b 。



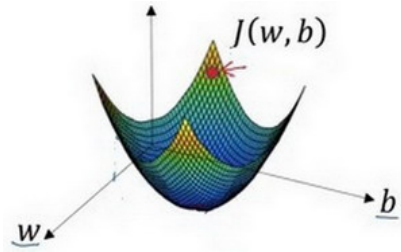
如图，代价函数（成本函数） $J(w, b)$ 是一个凸函数(convex function)，像一个大碗一样。



如图，这就与刚才的图有些相反，因为它非凸的并且有很多不同的局部最小值。由于逻辑回归的代价函数（成本函数） $J(w, b)$ 特性，我们必须定义代价函数（成本函数） $J(w, b)$ 为凸函数。 初始化 w 和 b ,

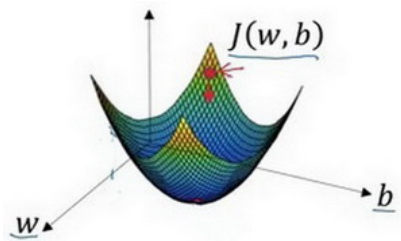


可以用如图那个小红点来初始化参数 w 和 b ，也可以采用随机初始化的方法，对于逻辑回归几乎所有的初始化方法都有效，因为函数是凸函数，无论在哪里初始化，应该达到同一点或大致相同的点。

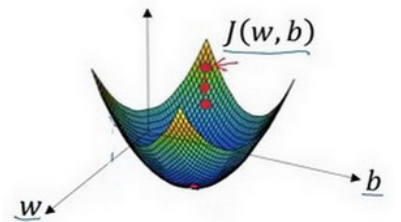


我们以如图的小红点的坐标来初始化参数 w 和 b 。

2. 朝最陡的下坡方向走一步，不断地迭代



我们朝最陡的下坡方向走一步，如图，走到了如图中第二个小红点处。

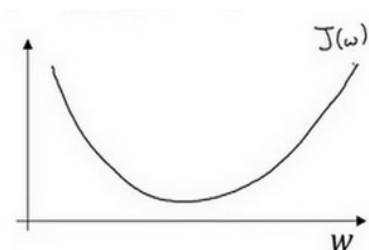


我们可能停在这里也有可能继续朝最陡的下坡方向再走一步，如图，经过两次迭代走到第三个小红点处。

3. 直到走到全局最优解或者接近全局最优解的地方

通过以上的三个步骤我们可以找到全局最优解，也就是代价函数（成本函数） $J(w, b)$ 这个凸函数的最小值点。

梯度下降法的细节化说明（仅有一个参数）



假定代价函数（成本函数） $J(w)$ 只有一个参数 w ，即用一维曲线代替多维曲线，这样可以更好画出图像。

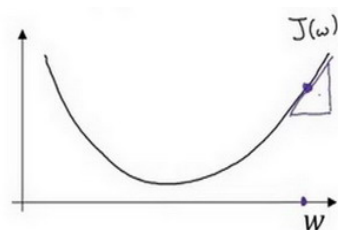
$$\text{Repeat } \left\{ \begin{array}{l} w := w - \alpha \frac{dJ(w)}{dw} \end{array} \right.$$

$$w := w - \alpha \frac{dJ(w)}{dw}$$

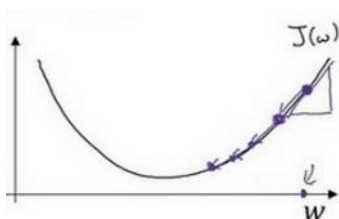
迭代就是不断重复做如图的公式：

$:=$ 表示更新参数，

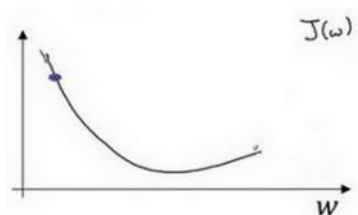
α 表示学习率（**learning rate**），用来控制步长（**step**），即向下走一步的长度 $\frac{dJ(w)}{dw}$ 就是函数 $J(w)$ 对 w 求导（**derivative**），在代码中我们会使用 dw 表示这个结果



对于导数更加形象化的理解就是斜率（**slope**），如图该点的导数就是这个点相切于 $J(w)$ 的小三角形的高除宽。假设我们以如图点为初始化点，该点处的斜率的符号是正的，即 $\frac{dJ(w)}{dw} > 0$ ，所以接下来会向左走一步。

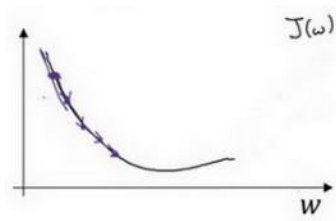


整个梯度下降法的迭代过程就是不断地向左走，直至逼近最小值点。



假设我们以如图点为初始化点，该点处的斜率的符号是负的，即 $\frac{dJ(w)}{dw} < 0$ ，所以接下

来会向右走一步。



整个梯度下降法的迭代过程就是不断地向右走，即朝着最小值点方向走。

梯度下降法的细节化说明（两个参数）

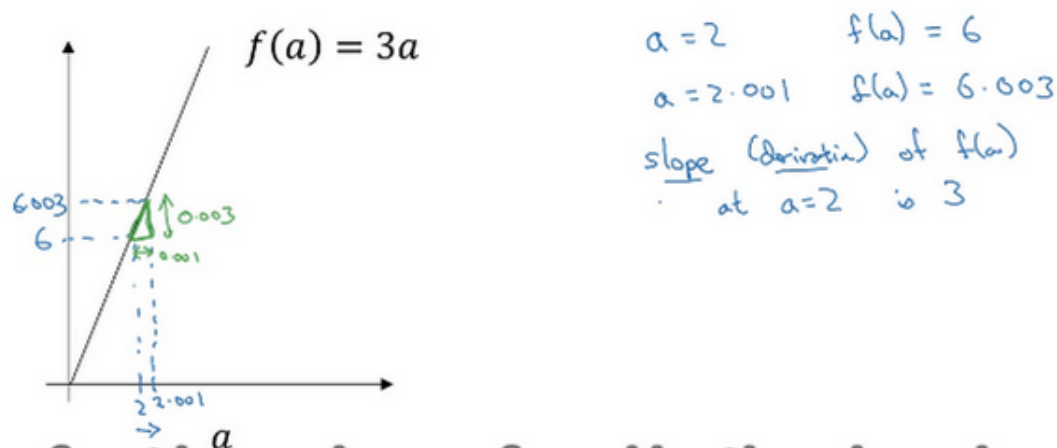
逻辑回归的代价函数（成本函数） $J(w, b)$ 是含有两个参数的。

$$w := w - a \frac{\partial J(w, b)}{\partial w} \quad b := b - a \frac{\partial J(w, b)}{\partial b}$$

∂ 表示求偏导符号，可以读作 **round**， $\frac{\partial J(w, b)}{\partial w}$ 就是函数 $J(w, b)$ 对 w 求偏导，在代码中我们会使用 dw 表示这个结果， $\frac{\partial J(w, b)}{\partial b}$ 就是函数 $J(w, b)$ 对 b 求偏导，在代码中我们会使用 db 表示这个结果，小写字母 d 用在求导数（**derivative**），即函数只有一个参数，偏导数符号 ∂ 用在求偏导（**partial derivative**），即函数含有两个以上的参数。

2.5 导数 (Derivatives)

Intuition about derivatives



一个函数 $f(a) = 3a$ ，它是一条直线。下面我们来简单理解下导数。让我们看看函数中几个点，假定 $a = 2$ ，那么 $f(a)$ 是 a 的 3 倍等于 6，也就是说如果 $a = 2$ ，那么函数 $f(a) = 6$ 。假定稍微改变一点点 a 的值，只增加一点，变为 2.001，这时 a 将向右做微小的移动。0.001 的差别实在是太小了，不能在图中显示出来，我们把它右移一点，现在 $f(a)$ 等于 a 的 3 倍是 6.003，画在图里，比例不太符合。请看绿色高亮部分的这个小三角形，如果向右移动 0.001，那么 $f(a)$ 增加 0.003， $f(a)$ 的值增加 3 倍于右移的 a ，因此我们说函数 $f(a)$ 在 $a = 2$ ，是这个导数的斜率，或者说，当 $a = 2$ 时，斜率是 3。导数这个概念意味着斜率，导数听起来是一个很可怕、很令人惊恐的词，但是斜率以一种很友好的方式来描述导数这个概念。所以提到导数，我们把它当作函数的斜率就好了。更正式的斜率定义为在上图这个绿色的小三角形中，高除以宽。即斜率等于 0.003 除以 0.001，等于 3。或者说导数等于 3，这表示当你将 a 右移 0.001， $f(a)$ 的值增加 3 倍水平方向的量。

现在让我们从不同的角度理解这个函数。

假设 $a = 5$ ，此时 $f(a) = 3a = 15$ 。

把 a 右移一个很小的幅度，增加到 5.001， $f(a) = 15.003$ 。即在 $a = 5$ 时，斜率是 3，这就是表示，当微小改变变量 a 的值， $\frac{df(a)}{da} = 3$ 。一个等价的导数表达式可以这样写 $\frac{d}{da}f(a)$ ，不管你是否将 $f(a)$ 放在上面或者放在右边都没有关系。

在这个视频中，我讲解导数讨论的情况是我们将 a 偏移 0.001，如果你想知道导数的数学定义，导数是你右移很小的 a 值（不是 0.001，而是一个非常非常小的值）。通常导数的定义是你右移 a （可度量的值）一个无限小的值， $f(a)$ 增加 3 倍（增加了一个非常非常小的值）。也

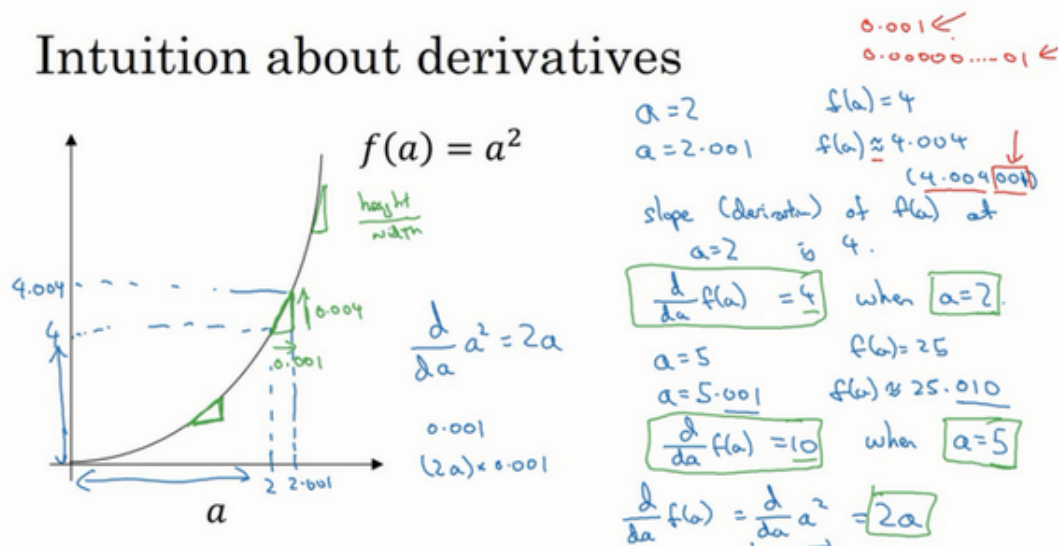
就是这个三角形右边的高度。

那就是导数的正式定义。但是为了直观的认识，我们将探讨右移 $a = 0.001$ 这个值，即使 0.001 并不是无穷小的可测数据。导数的一个特性是：这个函数任何地方的斜率总是等于 3 ，不管 $a = 2$ 或 $a = 5$ ，这个函数的斜率总等于 3 ，也就是说不管 a 的值如何变化，如果你增加 0.001 ， $f(a)$ 的值就增加 3 倍。这个函数在所有地方的斜率都相等。一种证明方式是无论你将小三角形画在哪里，它的高除以宽总是 3 。

我希望带给你一种感觉：什么是斜率？什么是导函数？对于一条直线，在例子中函数的斜率，在任何地方都是 3 。在下一个视频让我们看一个更复杂的例子，这个例子中函数在不同点的斜率是可变的。

2.6 更多的导数例子 (More Derivative Examples)

在这个视频中我将给出一个更加复杂的例子，在这个例子中，函数在不同点处的斜率是不一样的，先来举个例子：



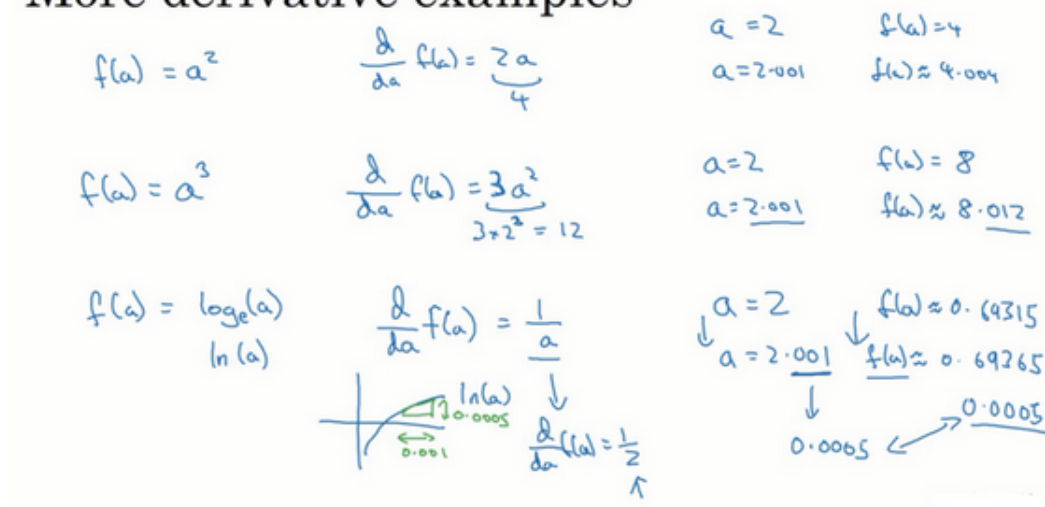
我在这里画一个函数， $f(a) = a^2$ ，如果 $a = 2$ 的话，那么 $f(a) = 4$ 。让我们稍稍往右推进一点点，现在 $a = 2.001$ ，则 $f(a) \approx 4.004$ (如果你用计算器算的话，这个准确值应该为4.004。0.001我只是为了简便起见，省略了后面的部分)，如果你在这儿画，一个小三角形，你就会发现，如果把 a 往右移动0.001，那么 $f(a)$ 将增大四倍，即增大0.004。在微积分中，我们把这个三角形斜边的斜率，称为 $f(a)$ 在点 $a = 2$ 处的导数(即为4)，或者写成微积分的形式，当 $a = 2$ 的时候， $\frac{d}{da} f(a) = 4$ 。由此可知，函数 $f(a) = a^2$ ，在 a 取不同值的时候，它的斜率是不同的，这和上个视频中的例子是不同的。

这里有种直观的方法可以解释，为什么一个点的斜率，在不同位置会不同。如果你在曲线上，的不同位置画一些小小的三角形，你就会发现，三角形高和宽的比值，在曲线上不同的地方，它们是不同的。所以当 $a = 2$ 时，斜率为4；而当 $a = 5$ 时，斜率为10。如果你翻看微积分的课本，课本会告诉你，函数 $f(a) = a^2$ 的斜率(即导数)为 $2a$ 。这意味着任意给定点 a ，如果你稍微将 a 增大0.001，那么你会看到 $f(a)$ 将增大 $2a$ ，即增大的值为点在 a 处斜率或导数，乘以你向右移动的距离。

现在有个小细节需要注意，导数增大的值，不是刚好等于导数公式算出来的值，而只是根据导数算出来的一个估计值。

为了总结这节课所学的知识，我们再来看看几个例子：

More derivative examples

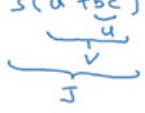


假设 $f(a) = a^3$ 如果你翻看导数公式表，你会发现这个函数的导数，等于 $3a^2$ 。所以这是什么意思呢，同样地举一个例子：我们再次令 $a = 2$ ，所以 $a^3 = 8$ ，如果我们又将 a 增大一点点，你会发现 $f(a) \approx 8.012$ ，你可以自己检查一遍，如果我们取 8.012 ，你会发现 2.001^3 ，和 8.012 很接近，事实上当 $a = 2$ 时，导数值为 3×2^2 ，即 $3 \times 4 = 12$ 。所以导数公式，表明如果你将 a 向右移动 0.001 时， $f(a)$ 将会向右移动 12 倍，即 0.012 。

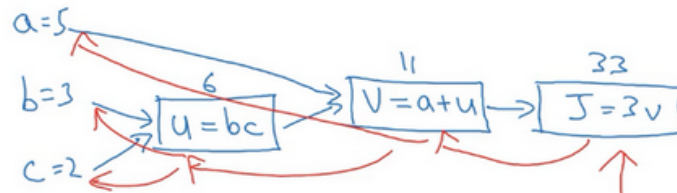
来看最后一个例子，假设 $f(a) = \log_e a$ ，有些可能会写作 $\ln a$ ，函数 $\log a$ 的斜率应该为 $\frac{1}{a}$ ，所以我们可以解释如下：如果 a 取任何值，比如又取 $a = 2$ ，然后又把 a 向右边移动 0.001 那么 $f(a)$ 将增大 $\frac{1}{a} \times 0.001$ ，如果你借助计算器的话，你会发现当 $a = 2$ 时 $f(a) \approx 0.69315$ ；而 $a = 2.001$ 时， $f(a) \approx 0.69365$ 。所以 $f(a)$ 增大了 0.0005 ，如果你查看导数公式，当 $a = 2$ 的时候，导数值 $\frac{d}{da} f(a) = \frac{1}{2}$ 。这表明如果你把 增大 0.001 ， $f(a)$ 将只会增大 0.001 的二分之一，即 0.0005 。如果你画个小三角形你就会发现，如果 x 轴增加了 0.001 ，那么 y 轴上的函数 $\log a$ ，将增大 0.001 的一半 即 0.0005 。所以 $\frac{1}{a}$ ，当 $a = 2$ 时这里是 $\frac{1}{2}$ ，就是当 $a = 2$ 时这条线的斜率。这些就是有关，导数的一些知识。

2.7 计算图 (Computation Graph)

Computation Graph

$$J(a,b,c) = 3(a+bc) = 3(5+3 \times 2) = 33$$


$$\begin{aligned} u &= bc \\ v &= a+u \\ J &= 3v \end{aligned}$$



我们尝试计算函数 J ， J 是由三个变量 a, b, c 组成的函数，这个函数是 $3(a + bc)$ 。计算这个函数实际上有三个不同的步骤，首先是计算 b 乘以 c ，我们把它储存在变量 u 中，因此 $u = bc$ ；然后计算 $v = a + u$ ；最后输出 $J = 3v$ ，这就是要计算的函数 J 。我们可以把这三步画成如下的计算图，我先在这画三个变量 a, b, c ，第一步就是计算 $u = bc$ ，我在这周围放个矩形框，它的输入是 b, c ，接着第二步 $v = a + u$ ，最后一步 $J = 3v$ 。

举个例子： $a = 5, b = 3, c = 2$ ， $u = bc$ 就是6，就是 $5+6=11$ 。 J 是3倍的，因此。即 $3 \times (5 + 3 \times 2)$ 。如果你把它算出来，实际上得到33就是 J 的值。当有不同的或者一些特殊的输出变量时，例如本例中的 J 和逻辑回归中你想优化的代价函数 J ，因此计算图用来处理这些计算会很方便。从这个小例子中我们可以看出，通过一个从左向右的过程，你可以计算出 J 的值。为了计算导数，从右到左（红色箭头，和蓝色箭头的过程相反）的过程是用于计算导数最自然的方式。

概括一下：计算图组织计算的形式是用蓝色箭头从左到右的计算，让我们看看下一个视频中如何进行反向红色箭头(也就是从右到左)的导数计算，让我们继续下一个视频的学习。

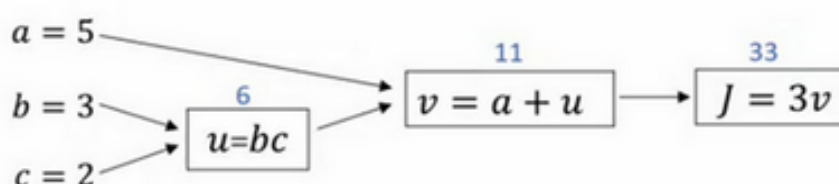
2.8 使用计算图求导数 (Derivatives with a Computation Graph)

在上一个视频中，我们看了一个例子使用流程计算图来计算函数 J 。现在我们看看流程图的描述，看看你如何利用它计算出函数 J 的导数。

下面用到的公式：

$$\frac{dJ}{du} = \frac{dJ}{dv} \frac{dv}{du}, \quad \frac{dJ}{db} = \frac{dJ}{du} \frac{du}{db}, \quad \frac{dJ}{da} = \frac{dJ}{du} \frac{du}{da}$$

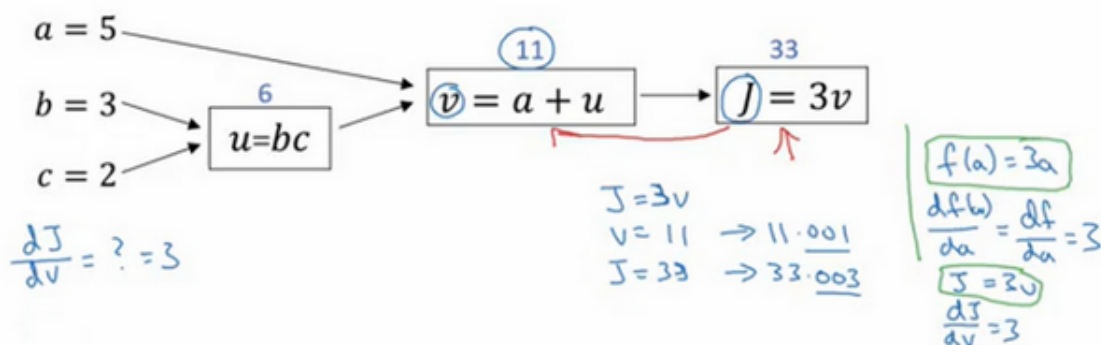
这是一个流程图：



假设你要计算 $\frac{dJ}{dv}$ ，那要怎么算呢？好，比如说，我们要把这个 v 值拿过来，改变一下，那么 J 的值会怎么变呢？

所以定义上 $J = 3v$ ，现在 $v = 11$ ，所以如果你让 v 增加一点点，比如到 11.001，那么 $J = 3v = 33.003$ ，所以我这里 v 增加了 0.001，然后最终结果是 J 上升到原来的 3 倍，所以 $\frac{dJ}{dv} = 3$ ，因为对于任何 v 的增量 J 都会有 3 倍增量，而且这类似于我们在上一个视频中的例子，我们有 $f(a) = 3a$ ，然后我们推导出 $\frac{df(a)}{da} = 3$ ，所以这里我们有 $J = 3v$ ，所以 $\frac{dJ}{dv} = 3$ ，这里 J 扮演了 f 的角色，在之前的视频里的例子。

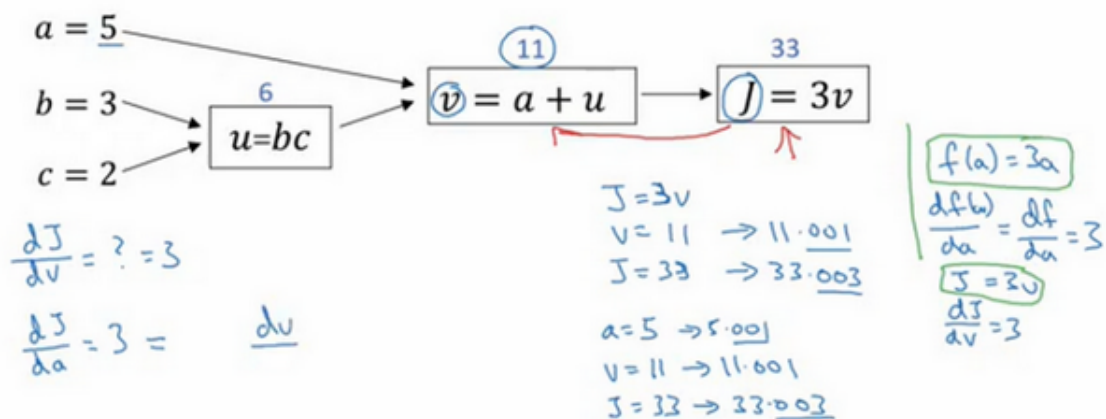
在反向传播算法中的术语，我们看到，如果你想计算最后输出变量的导数，使用你最关心的变量对 v 的导数，那么我们就做完了一步反向传播，在这个流程图中是一个反向步。



我们来看另一个例子， $\frac{dJ}{da}$ 是多少呢？换句话说，如果我们提高 a 的数值，对 J 的数值有什么影响？

么影响？

好，我们看看这个例子。变量 $a = 5$ ，我们让它增加到 5.001，那么对 v 的影响就是 $a + u$ ，之前 $v = 11$ ，现在变成 11.001，我们从上面看到现在 J 就变成 33.003 了，所以我们看到的是，如果你让 a 增加 0.001， J 增加 0.003。那么增加 a ，我是说如果你把这个 5 换成某个新值，那么 a 的改变量就会传播到流程图的最右，所以 J 最后是 33.003。所以 J 的增量是 3 乘以 a 的增量，意味着这个导数是 3。



要解释这个计算过程，其中一种方式是：如果你改变了 a ，那么也会改变 v ，通过改变 v ，也会改变 J ，所以 J 值的净变化量，当你提升这个值 (0.001)，当你把 a 值提高一点点，这就是 J 的变化量 (0.003)。

$$\begin{aligned} a &= 5 \rightarrow 5.001 \\ v &= 11 \rightarrow 11.001 \\ J &= 33 \rightarrow 33.003 \end{aligned}$$

首先 a 增加了， v 也会增加， v 增加多少呢？这取决于 $\frac{dv}{da}$ ，然后 v 的变化导致 J 也在增加，所以这在微积分里实际上叫链式法则，如果 a 影响到 v ， v 影响到 J ，那么当你让 a 变大时， J 的变化量就是当你改变 a 时， v 的变化量乘以改变 v 时 J 的变化量，在微积分里这叫链式法则。

Handwritten chain rule calculation:

$\frac{dJ}{dv} = ? = 3$

$\frac{dJ}{da} = 3 = \frac{dJ}{dv} \frac{dv}{da}$

$\frac{dv}{da} = 1$

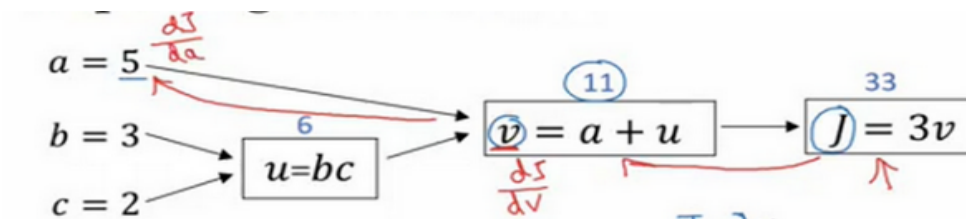
$\frac{dJ}{da} = 3$

$a \rightarrow v \rightarrow J$

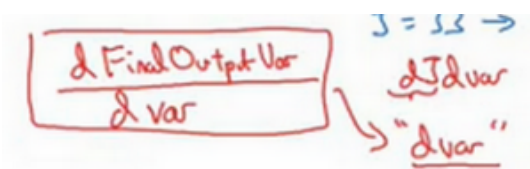
我们从这个计算中看到，如果你让 a 增加 0.001， v 也会变化相同的大小，所以 $\frac{dv}{da} = 1$ 。

事实上，如果你代入进去，我们之前算过 $\frac{dJ}{dv} = 3$ ， $\frac{dv}{da} = 1$ ，所以这个乘积 3×1 ，实际上就给出了正确答案， $\frac{dJ}{da} = 3$ 。

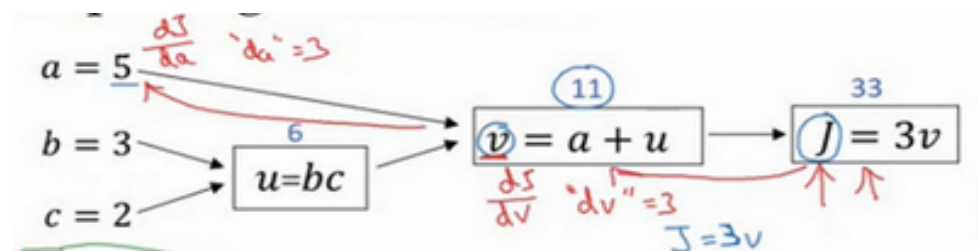
这张小图表示了如何计算， $\frac{dJ}{dv}$ 就是 J 对变量 v 的导数，它可以帮助你计算 $\frac{dJ}{da}$ ，所以这是另一步反向传播计算。



现在我想介绍一个新的符号约定，当你编程实现反向传播时，通常会有一个最终输出值是你关心的，最终的输出变量，你真正想要关心或者说优化的。在这种情况下最终的输出变量是 J ，就是流程图里最后一个符号，所以有很多计算尝试计算输出变量的导数，所以输出变量对某个变量的导数，我们就用 $dvar$ 命名，所以在很多计算中你需要计算最终输出结果的导数，在这个例子里是 J ，还有各种中间变量，比如 a 、 b 、 c 、 u 、 v ，当你在软件里实现的时候，变量名叫什么？你可以做的一件事是，在 **python** 中，你可以写一个很长的变量名，比如`dFinalOutputVar_dvar`，但这个变量名有点长，我们就用`dJ_dvar`，但因为你一直对 dJ 求导，对这个最终输出变量求导。我这里要介绍一个新符号，在程序里，当你编程的时候，在代码里，我们就使用变量名 $dvar$ ，来表示那个量。

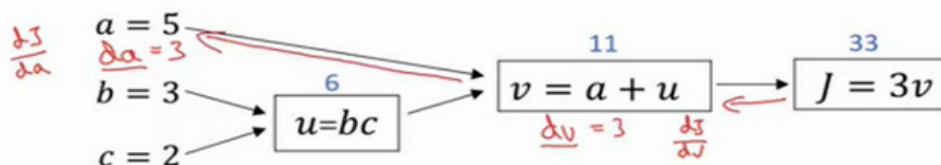


好，所以在程序里是 $dvar$ 表示导数，你关心的最终变量 J 的导数，有时最后是 L ，对代码中各种中间量的导数，所以代码里这个东西，你用 dv 表示这个值，所以 $dv = 3$ ，你的代码表示就是 $da = 3$ 。



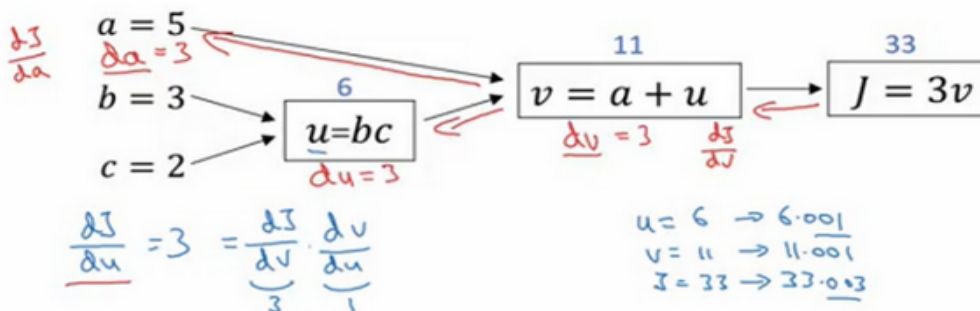
好，所以我们通过这个流程图完成部分的后向传播算法。我们在下一张幻灯片看看这个例子剩下的部分。

我们清理出一张新的流程图，我们回顾一下，到目前为止，我们一直在来回传播，并计算 $dv = 3$ ，再次， dv 是代码里的变量名，其真正的定义是 $\frac{dJ}{dv}$ 。我发现 $da = 3$ ，再次， da 是代码里的变量名，其实代表 $\frac{dJ}{da}$ 的值。



大概手算了一下，两条直线怎么计算反向传播。

好，我们继续计算导数，我们看看这个值 u ，那么 $\frac{dJ}{du}$ 是多少呢？通过和之前类似的计算，现在我们从 $u = 6$ 出发，如果你令 u 增加到 6.001，那么 v 之前是 11，现在变成 11.001 了， J 就从 33 变成 33.003，所以 J 增量是 3 倍，所以 $\frac{dJ}{du} = 3$ 。对 u 的分析很类似对 a 的分析，实际上这计算起来就是 $\frac{dJ}{dv} \cdot \frac{dv}{du}$ ，有了这个，我们可以算出 $\frac{dJ}{dv} = 3$ ， $\frac{dv}{du} = 1$ ，最终算出结果是 $3 \times 1 = 3$ 。



所以我们还有一步反向传播，我们最终计算出 $du = 3$ ，这里的 du 当然了，就是 $\frac{dJ}{du}$ 。

现在，我们仔细看看最后一个例子，那么 $\frac{dJ}{db}$ 呢？想象一下，如果你改变了 b 的值，你想要然后变化一点，让 J 值到达最大或最小，那么导数是什么呢？这个 J 函数的斜率，当你稍微改变 b 值之后。事实上，使用微积分链式法则，这可以写成两者的乘积，就是 $\frac{dJ}{du} \cdot \frac{du}{db}$ ，理由是，如果你改变 b 一点点，所以 b 变化比如说 3.001，它影响 J 的方式是，首先会影响 u ，它对 u 的影响有多大？好， u 的定义是 $b \cdot c$ ，所以 $b = 3$ 时这是 6，现在就变成 6.002 了，对吧，因为我们的例子中 $c = 2$ ，所以这告诉我们 $\frac{du}{db} = 2$ 当你让 b 增加 0.001 时， u 就增加两倍。所以

$\frac{du}{db} = 2$ ，现在我想 u 的增加量已经是 b 的两倍，那么 $\frac{dJ}{du}$ 是多少呢？我们已经弄清楚了，这等于

3，所以让这两部分相乘，我们发现 $\frac{dJ}{db} = 6$ 。

好，这就是第二部分的推导，其中我们想知道 u 增加 0.002，会对 J 有什么影响。实际上 $\frac{dJ}{du} = 3$ ，这告诉我们 u 增加 0.002 之后， J 上升了 3 倍，那么 J 应该上升 0.006，对吧。这可以从 $\frac{dJ}{du} = 3$ 推导出来。

如果你仔细看看这些数学内容，你会发现，如果 b 变成 3.001，那么 u 就变成 6.002， v 变成 11.002，然后 $J = 3v = 33.006$ ，对吧？这就是如何得到 $\frac{dJ}{db} = 6$ 。

Handwritten derivations and numerical examples:

$$\frac{dJ}{du} = 3 = \frac{dJ}{dv} \cdot \frac{dv}{du}$$

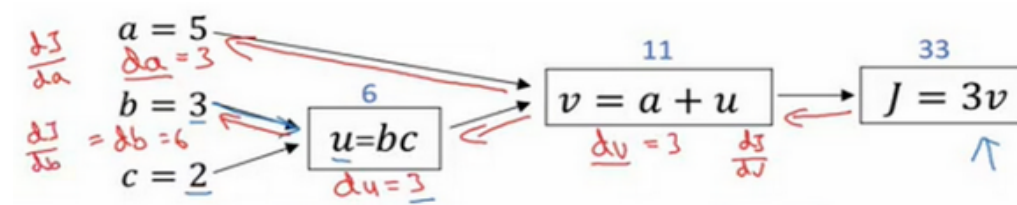
$$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 6$$

Numerical examples:

- $u = 6 \rightarrow 6.001$
- $v = 11 \rightarrow 11.001$
- $J = 33 \rightarrow 33.003$
- $b = 3 \rightarrow 3.001$
- $u = b \cdot c = 6 \rightarrow 6.002$
- $J = 33.006$
- $c = 2$
- $v = 11.002$
- $J = 33.006$

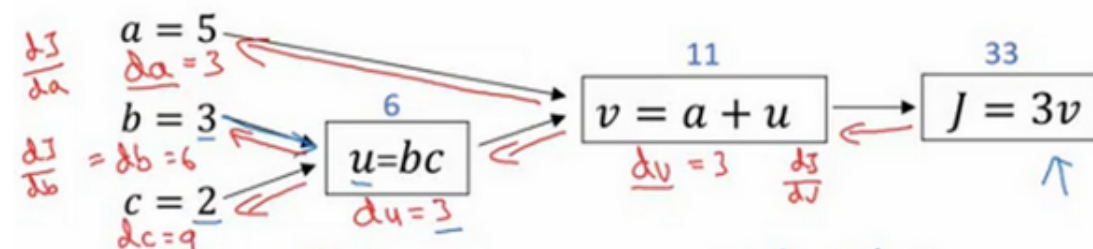
Andrew Ng

为了填进去，如果我们反向走的话， $db = 6$ ，而 db 其实是 Python 代码中的变量名，表示 $\frac{dJ}{db}$ 。



我不会很详细地介绍最后一个例子，但事实上，如果你计算 $\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc} = 3 \times 3$ ，这个结果是 9。

我不会详细说明这个例子，在最后一步，我们可以推出 $dc = 9$ 。



所以这个视频的要点是，对于那个例子，当计算所有这些导数时，最有效率的办法是从

右到左计算，跟着这个红色箭头走。特别是当我们第一次计算对 v 的导数时，之后在计算对 a 导数就可以用到。然后对 u 的导数，比如说这个项和这里这个项：

The image shows two handwritten equations illustrating the chain rule for backpropagation. The first equation is $\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 6$. In this equation, $\frac{dJ}{du}$ is boxed in green, and $\frac{du}{db}$ has a red arrow pointing to it from below with the value 2. The second equation is $\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc} = 9$. Here, $\frac{dJ}{du}$ is also boxed in green, and a red arrow points to it from the first equation's $\frac{du}{db}$ term, indicating the reuse of the derivative value.

可以帮助计算对 b 的导数，然后对 c 的导数。

所以这是一个计算流程图，就是正向或者说从左到右的计算来计算成本函数 J ，你可能需要优化的函数，然后反向从右到左计算导数。如果你不熟悉微积分或链式法则，我知道这里有些细节讲的很快，但如果你没有跟上所有细节，也不用怕。在下一个视频中，我会再过一遍。在逻辑回归的背景下过一遍，并给你介绍需要做什么才能编写代码，实现逻辑回归模型中的导数计算。

2.9 逻辑回归中的梯度下降 (Logistic Regression Gradient Descent)

假设样本只有两个特征 x_1 和 x_2 ，为了计算 z ，我们需要输入参数 w_1 、 w_2 和 b ，除此之外还有特征值 x_1 和 x_2 。因此 z 的计算公式为： $z = w_1x_1 + w_2x_2 + b$

回想一下逻辑回归的公式定义如下： $\hat{y} = a = \sigma(z)$ 其中 $z = w^T x + b$ ， $\sigma(z) = \frac{1}{1+e^{-z}}$

损失函数： $L(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)}\log\hat{y}^{(i)} - (1 - y^{(i)})\log(1 - \hat{y}^{(i)})$

代价函数： $J(w, b) = \frac{1}{m} \sum_i^m L(\hat{y}^{(i)}, y^{(i)})$

假设现在只考虑单个样本的情况，单个样本的代价函数定义如下：

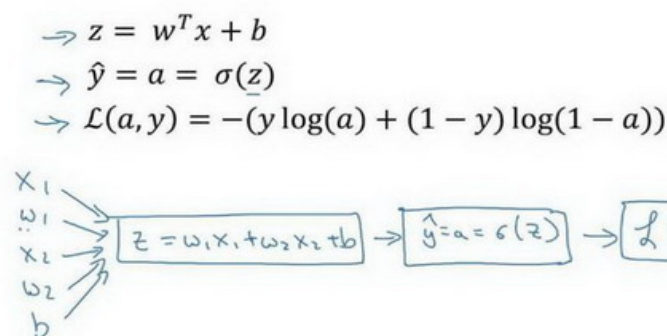
$$L(a, y) = -(y\log(a) + (1 - y)\log(1 - a))$$

其中 a 是逻辑回归的输出， y 是样本的标签值。现在让我们画出表示这个计算的计算图。

这里先复习下梯度下降法， w 和 b 的修正量可以表达如下：

$$w := w - a \frac{\partial J(w, b)}{\partial w}, \quad b := b - a \frac{\partial J(w, b)}{\partial b}$$

Logistic regression recap



如图：在这个公式的外侧画上长方形。然后计算： $\hat{y} = a = \sigma(z)$ 也就是计算图的下一步。最后计算损失函数 $L(a, y)$ 。有了计算图，我就不需要再写出公式了。因此，为了使得逻辑回归中最小化代价函数 $L(a, y)$ ，我们需要做的仅仅是修改参数 w 和 b 的值。前面我们已经讲解了如何在单个训练样本上计算代价函数的前向步骤。现在让我们来讨论通过反向计算出导数。因为我们想要计算出的代价函数 $L(a, y)$ 的导数，首先我们需要反向计算出代价函数 $L(a, y)$ 关于 a 的导数，在编写代码时，你只需要用 da 来表示 $\frac{dL(a, y)}{da}$ 。

通过微积分得到： $\frac{dL(a, y)}{da} = -y/a + (1 - y)/(1 - a)$

现在可以再反向一步，在编写 **Python** 代码时，你只需要用 dz 来表示代价函数 L 关于 z 的导数 $\frac{dL}{dz}$ ，也可以写成 $\frac{dL(a,y)}{dz}$ ，这两种写法都是正确的。 $\frac{dL}{dz} = a - y$ 。

$$\text{因为 } \frac{dL(a,y)}{dz} = \frac{dL}{dz} = \left(\frac{dL}{da}\right) \cdot \left(\frac{da}{dz}\right), \text{ 并且 } \frac{da}{dz} = a \cdot (1 - a), \text{ 而 } \frac{dL}{da} = \left(-\frac{y}{a} + \frac{(1-y)}{(1-a)}\right),$$

因此将这两项相乘，得到：

$$dz = \frac{dL(a,y)}{dz} = \frac{dL}{dz} = \left(\frac{dL}{da}\right) \cdot \left(\frac{da}{dz}\right) = \left(-\frac{y}{a} + \frac{(1-y)}{(1-a)}\right) \cdot a(1 - a) = a - y$$

视频中为了简化推导过程，假设 n_x 这个推导的过程就是我之前提到过的链式法则。如果你对微积分熟悉，放心地去推导整个求导过程，如果不熟悉微积分，你只需要知道 $dz = (a - y)$ 已经计算好了。

现在进行最后一步反向推导，也就是计算 w 和 b 变化对代价函数 L 的影响，特别地，可以用：

$$dw_1 = \frac{1}{m} \sum_i^m x_1^{(i)} (a^{(i)} - y^{(i)})$$

$$dw_2 = \frac{1}{m} \sum_i^m x_2^{(i)} (a^{(i)} - y^{(i)})$$

$$db = \frac{1}{m} \sum_i^m (a^{(i)} - y^{(i)})$$

视频中， dw_1 表示 $\frac{\partial L}{\partial w_1} = x_1 \cdot dz$ ， dw_2 表示 $\frac{\partial L}{\partial w_2} = x_2 \cdot dz$ ， $db = dz$ 。

因此，关于单个样本的梯度下降算法，你所需要做的就是如下的事情：

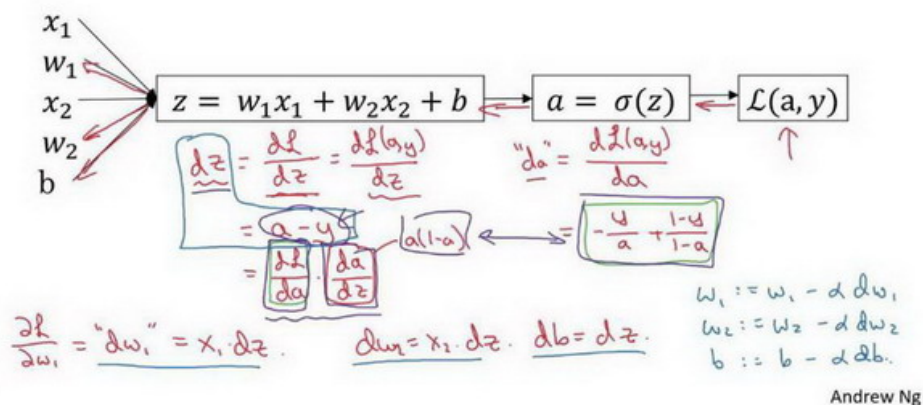
使用公式 $dz = (a - y)$ 计算 dz ，

使用 $dw_1 = x_1 \cdot dz$ 计算 dw_1 ， $dw_2 = x_2 \cdot dz$ 计算 dw_2 ， $db = dz$ 来计算 db ，

然后：更新 $w_1 = w_1 - \alpha dw_1$ ， 更新 $w_2 = w_2 - \alpha dw_2$ ， 更新 $b = b - \alpha db$ 。

这就是关于单个样本实例的梯度下降算法中参数更新一次的步骤。

Logistic regression derivatives



现在你已经知道了怎样计算导数，并且实现针对单个训练样本的逻辑回归的梯度下降算法。但是，训练逻辑回归模型不仅仅只有一个训练样本，而是有 m 个训练样本的整个训练集。因此在下一节视频中，我们将这些思想应用到整个训练样本集中，而不仅仅是单个样本上。

2.10 m 个样本的梯度下降(Gradient Descent on m Examples)

在之前的视频中,你已经看到如何计算导数,以及应用梯度下降在逻辑回归的一个训练样本上。现在我们想要把它应用在 m 个训练样本上。

Logistic regression on m examples

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \ell(a^{(i)}, y^{(i)})$$

$(x^{(i)}, y^{(i)})$

$$\rightarrow a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

$\underline{dw_1^{(i)}}, \underline{dw_2^{(i)}}, \underline{db^{(i)}}$

$$\underline{\frac{\partial}{\partial}} J(w, b) = \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial w_i} \ell(a^{(i)}, y^{(i)})$$

首先,让我们时刻记住有关于损失函数 $J(w, b)$ 的定义。

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

当你的算法输出关于样本 y 的 $a^{(i)}$, $a^{(i)}$ 是训练样本的预测值,即: $\sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$ 。 所以我们在前面的幻灯中展示的是对于任意单个训练样本,如何计算微分当你只有一个训练样本。因此 dw_1 , dw_2 和 db 添上上标 i 表示你求得的相应的值。如果你面对的是我们在之前的幻灯中演示的那种情况,但只使用了一个训练样本 $(x^{(i)}, y^{(i)})$ 。

现在你知道带有求和的全局代价函数,实际上是 1 到 m 项各个损失的平均。 所以它表明全局代价函数对 w_1 的微分,对 w_1 的微分也同样是各项损失对 w_1 微分的平均。

Logistic regression on m examples

$$J=0; \underline{dw_1}=0; \underline{dw_2}=0; \underline{db}=0$$

For $i=1$ to m

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

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

$$J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$$

$$\underline{dz^{(i)}} = a^{(i)} - y^{(i)}$$

$$\begin{aligned} \uparrow \quad dw_1 &+= x_1^{(i)} dz^{(i)} \\ dw_2 &+= x_2^{(i)} dz^{(i)} \end{aligned} \quad \downarrow n=2$$

$$\begin{aligned} dw_1 &+= x_1^{(i)} dz^{(i)} \\ db &+= dz^{(i)} \end{aligned}$$

$J/=m \leftarrow$

$$\underline{dw_1}/=m; \quad \underline{dw_2}/=m; \quad \underline{db}/=m. \leftarrow$$

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$$w_1 := w_1 - \alpha \underline{dw_1}$$

$$w_2 := w_2 - \alpha \underline{dw_2}$$

$$b := b - \alpha \underline{db}$$

Vec.

但之前我们已经演示了如何计算这项，即之前幻灯中演示的如何对单个训练样本进行计算。所以你真正需要做的是计算这些微分，如我们在之前的训练样本上做的。并且求平均，这会给你全局梯度值，你能够把它直接应用到梯度下降算法中。

所以这里有很多细节，但让我们把这些装进一个具体的算法。同时你需要一起应用的就是逻辑回归和梯度下降。

我们初始化 $J = 0, dw_1 = 0, dw_2 = 0, db = 0$

代码流程：

$J=0; dw1=0; dw2=0; db=0;$

for $i = 1$ to m

$z(i) = wx(i)+b;$

$a(i) = \text{sigmoid}(z(i));$

$J += -[y(i)\log(a(i))+(1-y(i)) \log(1-a(i))];$

$dz(i) = a(i)-y(i);$

$dw1 += x1(i)dz(i);$

$dw2 += x2(i)dz(i);$

$db += dz(i);$

$J/=m;$

$dw1/=m;$

$dw2/=m;$

$db/=m;$

$w = w - \alpha * dw$

$b = b - \alpha * db$

但这种计算中有两个缺点,也就是说应用此方法在逻辑回归上你需要编写两个 **for** 循环。第一个 **for** 循环是一个小循环遍历 m 个训练样本,第二个 **for** 循环是一个遍历所有特征的 **for** 循环。这个例子中我们只有 2 个特征,所以 n 等于 2 并且 n_x 等于 2。但如果你有更多特征,你开始编写你的因此 dw_1 , dw_2 , 你有相似的计算从 dw_3 一直下去到 dw_n 。所以看来你需要一个 **for** 循环遍历所有 n 个特征。

当你应用深度学习算法,你会发现在代码中显式地使用 **for** 循环使你的算法很低效,同时在深度学习领域会有越来越大的数据集。所以能够应用你的算法且没有显式的 **for** 循环会是重要的,并且会帮助你适用于更大的数据集。所以这里有一些叫做向量化技术,它可以允许你的代码摆脱这些显式的 **for** 循环。

我想在先于深度学习的时代,也就是深度学习兴起之前,向量化是很棒的。可以使你有时候加速你的运算,但有时候也未必能够。但是在深度学习时代向量化,摆脱 **for** 循环已经变得相当重要。因为我们越来越多地训练非常大的数据集,因此你真的需要你的代码变得非常高效。所以在接下来的几个视频中,我们会谈到向量化,以及如何应用向量化而连一个 **for** 循环都不使用。所以学习了这些,我希望你有关于如何应用逻辑回归,或是用于逻辑回归的梯度下降,事情会变得更加清晰。当你进行编程练习,但在真正做编程练习之前让我们先谈谈向量化。然后你可以应用全部这些东西,应用一个梯度下降的迭代而不使用任何 **for** 循环。