### 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，、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 , ，所以如果你想使用非向量化方法去计算，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，代码如下：

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

这是向量化计算的方法，你将会发现这个非常快

图片包含 人员, 文字, 白板

描述已自动生成

**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。**

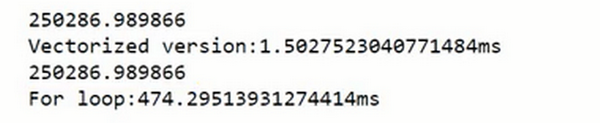
以下代码及运行结果截图：

图片包含 屏幕截图

描述已自动生成

图片包含 屏幕截图

描述已自动生成



### 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(l**ogistic**)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，这时矩阵乘法定义为，矩阵乘法的定义就是：，这取决于你怎么定义值。同样使用非向量化实现，， 并且通过两层循环，得到 。现在就有了 和 的两层循环，这就是非向量化。向量化方式就可以用，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。

图片包含 人员

描述已自动生成

事实上，**numpy**库有很多向量函数。比如 u=np.log是计算对数函数()、 np.abs() 是计算数据的绝对值、np.maximum() 计算元素中的最大值，你也可以 np.maximum(v,0) 、 代表获得元素 每个值得平方、 获取元素 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。

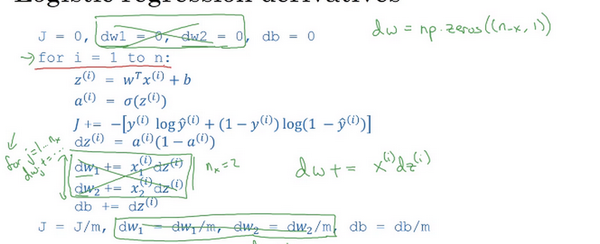
图片包含 文字

描述已自动生成

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。在这例子我们有个特征值。如果你有超过两个特征时，需要循环 、 、 等等。所以 的实际值是1、2 和 ，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 ， 都等于0。去掉这些，而是定义 为一个向量，设置 。定义了一个行的一维向量，从而替代循环。我们仅仅使用了一个向量操作 。最后，我们得到 。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。

图片包含 文字

描述已自动生成



### 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 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 ，我正在使用这个熟悉的公式 。然后计算激活函数，计算第一个样本的预测值 。

对第二个样本进行预测，你需要计算 ， 。

对第三个样本进行预测，你需要计算 ， ，依次类推。如果你有 个训练样本，你可能需要这样做 次，可以看出，为了完成前向传播步骤，即对我们的 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 作为你的训练输入，(如下图中蓝色 )像这样在不同的列中堆积在一起。这是一个 行 列的矩阵。我现在将它写为**Python numpy**的形式 ，这只是表示 是一个 乘以 的矩阵 。

图片包含 文字

描述已自动生成

现在我首先想做的是告诉你该如何在一个步骤中计算 、 、 等等。实际上，只用了一行代码。所以，我打算先构建一个 **的矩阵，实际上它是一个行向量**，同时我准备计算 ， ……一直到 ，所有值都是在同一时间内完成。结果发现它可以表达为 的转置乘以大写矩阵 然后加上向量 ， 。 是一个 的向量或者 的矩阵或者是一个 维的行向量。所以希望你熟悉矩阵乘法，你会发现的 转置乘以 ， 一直到 。所以 转置可以是一个行向量。所以第一项 将计算 的转置乘以 ， 转置乘以 等等。然后我们加上第二项 ，你最终将 加到了每个元素上。所以你最终得到了另一个 的向量， 。

这是第一个元素， 这是第二个元素， 这是第 个元素。

如果你参照上面的定义，第一个元素恰好是 的定义，第二个元素恰好是 的定义，等等。所以，因为是一次获得的，当你得到你的训练样本，一个一个横向堆积起来，这里我将 定义为大写的 ，你用小写 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 横向堆积在一起时得到大写变量 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 。结果发现，为了计算 ，**numpy**命令是。这里在**Python**中有一个巧妙的地方，这里 是一个实数，或者你可以说是一个 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 扩展成一个 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 ，而大写 是一个包含所有小写 到 的 的矩阵。这就是 的内容，关于变量 又是如何呢？

我们接下来要做的就是找到一个同时计算 的方法。就像把小写 堆积起来得到大写 和横向堆积小写 得到大写 一样，堆积小写变量 将形成一个新的变量，我们将它定义为大写 。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 作为变量并且非常高效地输出大写 。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 个训练样本一次性计算出小写 和小写 ，用一行代码即可完成。

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

这一行代码： ，通过恰当地运用一次性计算所有 。这就是在同一时间内你如何完成一个所有 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，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个训练数据的梯度，本次视频的重点是如何**同时**计算 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， ， ……等等一系列类似公式。现在，对 个训练数据做同样的运算，我们可以定义一个新的变量 ，所有的 变量横向排列，因此， **是一个 的矩阵**，或者说，一个 维行向量。在之前的幻灯片中，我们已经知道如何计算，即 ,我们需要找到这样的一个行向量 ，由此，我们可以这样计算 ，不难发现第一个元素就是 ，第二个元素就是 ……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：

………….

………….

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 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 ，不难发现 ， 之前的讲解中，我们知道所有的已经组成一个行向量 了，所以在**Python**中，我们很容易地想到；接下来看，我们先写出它的公式 其中， 是一个行向量。因此展开后 。因此我们可以仅用两行代码进行计算：， 。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：

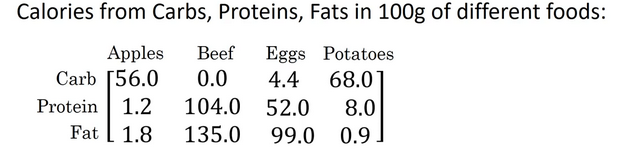
图片包含 文字

描述已自动生成

我们的目标是不使用**for**循环，而是向量，我们可以这么做：

### 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。



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，记为 ，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。

图片包含 屏幕截图

描述已自动生成

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。

图片包含 屏幕截图

描述已自动生成

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 的矩阵除以一个的矩阵，得到了一个 的结果矩阵，这个结果矩阵就是我们要求的百分比含量。

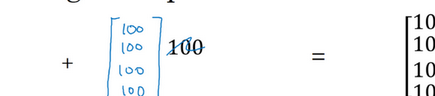
图片包含 屏幕截图

描述已自动生成

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。**

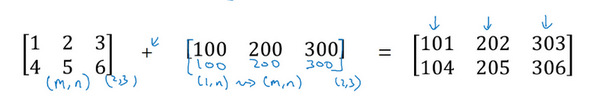
而第二个A/cal.reshape(1,4)指令则调用了**numpy**中的广播机制。这里使用 的矩阵除以 的矩阵。技术上来讲，其实并不需要再将矩阵 reshape(重塑)成 ，因为矩阵本身已经是 了。但是当我们写代码时不确定矩阵维度的时候，通常会对矩阵进行重塑来确保得到我们想要的列向量或行向量。重塑操作reshape是一个常量时间的操作，时间复杂度是，它的调用代价极低。

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



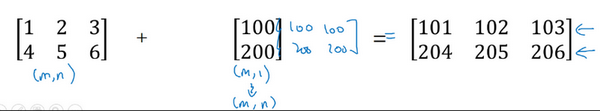
在**numpy**中，当一个 的列向量与一个常数做加法时，实际上会将常数扩展为一个 的列向量，然后两者做逐元素加法。结果就是右边的这个向量。这种广播机制对于行向量和列向量均可以使用。

再看下一个例子。



用一个 的矩阵和一个 的矩阵相加，其泛化形式是 的矩阵和 的矩阵相加。在执行加法操作时，其实是将 的矩阵复制成为 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。这就是在前一张幻灯片中计算卡路里百分比的广播机制，只不过这里是除法操作（广播机制与执行的运算种类无关）。

下面是最后一个例子



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

广播机制的一般原则如下：

图片包含 文字, 白板, 人员

描述已自动生成

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 后缘维度的轴长度是4，而矩阵 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 沿axis=0(垂直方向)复制成为 ，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 和矩阵 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 广播成为 ，之后做逐元素四则运算。

矩阵 和矩阵 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 广播成为 ，之后做逐元素四则运算。

矩阵 和常数 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，即广播成为 ，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：

图片包含 电子产品

描述已自动生成

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

本节主要讲**Python**中的**numpy**一维数组的特性，以及与行向量或列向量的区别。并介绍了老师在实际应用中的一些小技巧，去避免在coding中由于这些特性而导致的bug。

**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，这样会生成存储在数组 中的5个高斯随机数变量。之后输出 ，从屏幕上可以得知，此时 的**shape**（形状）是一个的结构。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，最终结果它会和看起来一样，所以和的转置阵最终结果看起来一样。而如果我输出和的转置阵的内积，你可能会想：乘以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。

图片包含 屏幕截图

描述已自动生成

所以我建议当你编写神经网络时，不要在它的**shape**是还是或者一维数组时使用数据结构。相反，如果你设置 为，那么这就将置于5行1列向量中。在先前的操作里 和 的转置看起来一样，而现在这样的 变成一个新的 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 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。

图片包含 屏幕截图

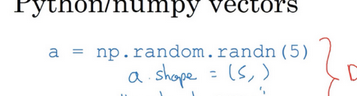
描述已自动生成

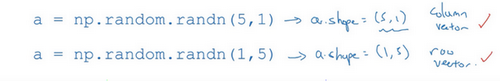
如果你输出 和 的转置的乘积，然后会返回给你一个向量的外积，是吧？所以这两个向量的外积返回给你的是一个矩阵。

图片包含 屏幕截图

描述已自动生成

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 ，而且它生成了一个数据结构 ，是，一个有趣的东西。这被称作 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，等同于。这种表现很像 ，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；同时这里 将要变成，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 ，表明一个数组或者一个数组，以致于它表现更像列向量或行向量。



我有时候看见学生因为一维数组不直观的影响，难以定位**bug**而告终。通过在原先的代码里清除一维数组，我的代码变得更加简洁。而且实际上就我在代码中表现的事情而言，我从来不使用一维数组。因此，要去简化你的代码，而且不要使用一维数组。总是使用 维矩阵（基本上是列向量），或者 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。

总之，我希望这些建议能帮助你解决一个**Python**中的**bug**，从而使你更容易地完成练习。