3.1 神经网络概览

在这一章，将学习如何实现一个神经网络，在我们深入学习技术细节之前，想带你快速的了解一下在这章课程中，学习到哪些内容：

上一章课程中我们学习了逻辑回归，其实就是一个简单的神经网络（每层只有一个神经元的神经网络），你可以把很多sigmoid单元堆叠起来构成一个神经网络，接下来我们会使用这些符号：

![神经网络图示](<http://upload-images.jianshu.io/upload_images/1779926-7281e6ca62d9fdb7.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

首先，用x表示输入特征，还有参数W和b，这样就可以计算出，会使用到新的符号，上标方括号1，表示与这些节点相关的量，也就是所谓的“层”。用上标方括号2表示和第二层相关的量，这里使用这样的上标方括号目的是区分用来表示单个训练样本的圆括号，方括号表示不同的层，然后使用类似logistic回归去计算之后，需要使用sigmoid()去计算，接下来用另一个线性方程计算，以及，就是整个神经网络的最终输出。同时还是用表示神经网络的输出。总之，在浅层神经网络中，需要反复计算a和z，最后计算损失函数。同样的，也有反向传播的过程计算梯度值。

实际上，图中的神经网络就是把逻辑回归重复了两次。

3.1 神经网络的表示

这一节将讨论神经网络这些图形的具体含义，也就是说，我画的神经网络到底代表什么。首先我们看只有一个隐藏层的神经网络。

![双层神经网络](<http://upload-images.jianshu.io/upload_images/1779926-ca15cdbd42d0745d.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

![双层神经网络表示](<http://upload-images.jianshu.io/upload_images/1779926-93ea3f79a374b70f.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

命名一下这张图的各个部分，我们有输入特征x1，x2，x3竖向堆叠起来，这是神经网络的输入层，它包含了神经网络的输入，往右的一层，我们称之为神经网络的隐藏层，接下里会讲隐藏是什么意思。这里的最后一层只有一个节点，而这个只有一个节点的层就是输出层，它负责输出预测值。

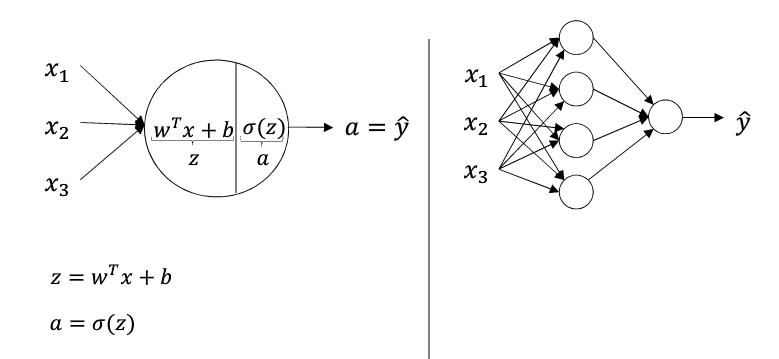
在一个神经网络中，当你使用监督学习训练它时，训练集包含了输入x，还有目标输出y，“隐藏层”的含义是在训练集中，这些中间节点的真正数值我们是不知道的，在训练集中你看不到它们的数值，只是表示你在训练集中看不到。

再引入几个符号，之前我们用向量x来表示输入特征，输入特征的数值还有另外一种表示方式，我们用来表示，而这个，就是“激活”的意思，它意味着网络中不同层的值，会传递给后面的层，输入层将x的值传递给隐藏层，所以我们将输入层的激活值称为。下一层隐藏层也会产生一些激活值，所以记作，具体地，第一层隐藏层的第一个单元，或者节点记作，这一层的第二个记作，以此类推。图中的是一个思维向量，写成python，它是一个4\*1矩阵或大小为4的列向量，因为这个隐藏层有四个单元，四个隐藏层单元，最后的输出层会产生某个数值，是个实数，的值就是。这里和logistic回归相似，在逻辑回归中，等于，这里只有一个输出层，所以没有用带方括号的上标。

这里你看到的神经网络是所谓的**双层神经网络**，当我们计算网络的层数时，不算入输入层，原因是隐藏层是第一层，输出层是第二层，将输入层称为第0层根据相关约定的符号。从表面上看这是个三层神经网络，但在论文中都称为双层神经网络，因为我们不把输入层看做一个标准的层，最后我们要知道隐藏层以及最后的输出层是带有参数的，这里的隐藏层是有两个相关的参数W和b，使用和来表示，是和第一层这个隐藏层有关的。在这个例子中，后面会看到W是一个4\*3的矩阵，b是一个4\*1的向量，第一个数4意思是有四个节点，或者说四个隐藏单元，数字3来自于有3个输入特征。之后会更加详细地讨论这些矩阵的维数，类似地，输出层也有一些和它相关的参数和，从维数看来，分别是1\*4和1\*1，这里的1\*4，是因为隐藏层有四个隐藏单元，而输出层只有一个单元。

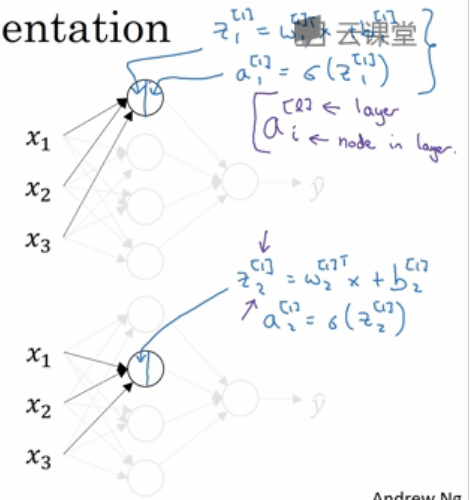
3.3 计算神经网络的输出

这一节我们探讨如何计算出神经网络的输出，你所看到的是像logistic那样的运算过程，但整个过程会重复很多遍。

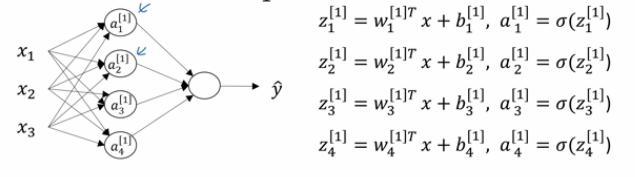


之前说过，logistic回归，这里的圆圈代表了回归计算的两个步骤，首先你按步骤计算出z，然后第二步计算激活函数，sigmoid()，所以神经网络只不过重复计算这些步骤很多次。首先，我们只看隐藏层的一个节点，暂时隐去其他节点，左边看上去和logistic回归很相似，隐藏层的这个节点进行两步计算，同样的可以这么看，节点这个圆圈的左边计算，上标[1]都是为了表示这些都是和第一隐层有关的量，这是隐层的第一个节点，圆圈的右边则是。所以这个小圆圈即神经网络的第一个节点表示执行者两步计算，载看看神经网络的第二个节点，即隐藏层的第二个节点，与上一个图的logistic回归单元类似，这个小圆圈仍然是代表了计算的两个步骤，，以及。

隐藏层前两个节点单独表示：



我们已经讨论了神经网络的前两个隐层单元，第三个第四个也表示同样的计算，现在把这些等式整理到一块，如下图：

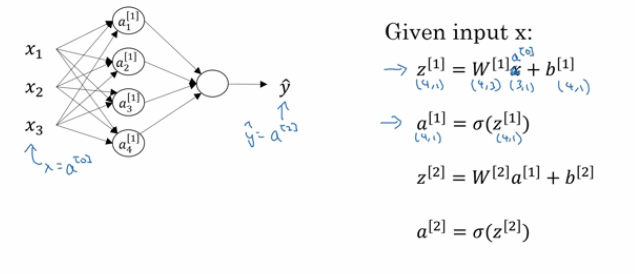


如果你确实在神经网络中执行，用for循环来做这些看起来真的很低效，接下来做的就是将这四个等式向量化，如何把z看做向量计算，可以这么做：

把这些w堆叠起来，构成一个矩阵，w本来是列向量，转置之后就是行向量，将4个w向量堆叠在一起，得到，另一种理解，我们有四个logistic回归单元，而每一个logistic回归单元都有对应的参数，向量w，把这四个向量堆叠起来，就能得到4\*3的矩阵。然后如果你把这个矩阵乘以你输入的特征x1，x2，x3，根据矩阵乘法并且加上b最后得到。而最后得到的这个向量就称作向量。是单独的z堆叠起来构成的一个列向量。

向量化是有一条经验法则，就是当我们在一层中有不同的节点，那就纵向堆叠，所以就对应隐藏层的4个不同的节点，把这四个结果纵向堆叠起来就得到。如果用令一种符号惯例来表示这个W是通过堆叠等等形成的，把这个矩阵称为，类似的关于隐藏层的所有b堆叠为，所以这是一个4\*1的向量，现在使用矩阵来计算Z，最后一件事就是用这些Z来计算a值。所以来表示这一层最后的激活值的堆叠，。

概括一下，我们发现，而。对于神经网络的第一层，给予一个输入X，我们有如下的计算：



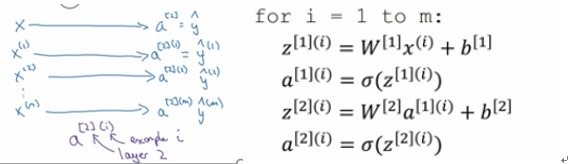
记得我们说过，X等于，就像一样，所以我们把X替换为，相当于有个别名，用同样的推导方法，可以得到下一层的表示，输出层的作用是，它带参数，，这里的就是一个1\*4的矩阵，就是一个实数，所以也就为一个实数，。 如果你把这最后的输出单元看作是logistic回归的类似物，也含有参数w和b，那么这里的w实际上就是转置，b实际上就是。

归纳一下，对于logistic回归为了计算输出或者说预测，你要计算z，和，当你有一个单隐藏层神经网络，你需要在代码中去实现的就是上一个图中的四个等式，并且你可以把这看成是一个向量化的计算过程，分别计算出这四个隐藏层中的logistic回归单元和输出层那一个单元。同样地，把整个训练样本都向量化，可以发现通过把不同训练样本堆叠起来构成矩阵，只需要稍微修改这些公式，就可以得到类似前面ligistic回归一样的结果，能够同时计算出不止一个样本的神经网络输出，而是能一次性计算你的整个训练集。

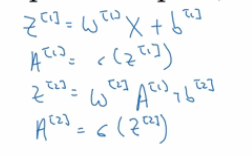
3.4 多个例子中的向量化

上一节已经知道单个训练样本是，计算神经网络的预测，这一节学习如何将不同训练样本向量化，输出结果和logistic回归很相似，如何将不同训练样本堆叠起来，放入矩阵的各列。可以把上一节的公式拿出来稍微改变一点点，把输入方式变一下，让神经网络几乎同时计算所有样本的输出。

上一节的那四个公式，说明了在给定输入特征x，对于单个训练样本，可以用它们生成一个，现在你有m个训练样本，你需要重复这个过程，用来计算，用来计算，同理的，一直到用来计算，对应的分别为, 这里都有了两个上标，圆括号i表示第i个训练样本，方括号里面表示这是神经网络的第几层，注意输入层算作第0层。所以这些表明如果你有一个没有向量化的实现，并想要计算所有训练样本的预测，你需要样本进行遍历，分别计算这4个公式，步骤如图：

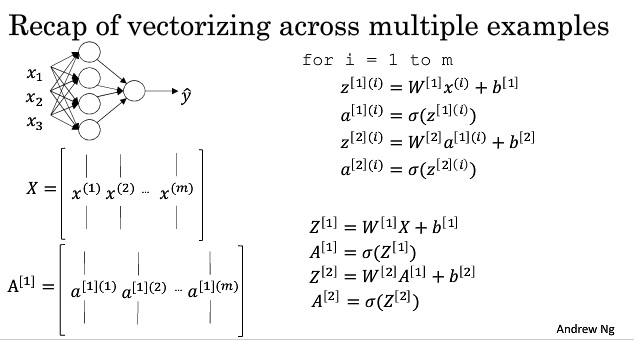


步骤都是一样的，只不过要区别是第几个样本，所有都有上标i来表示训练样本的所有样本。一般可以通过向量化来去掉for循环，记得上一章二分分类逻辑回归定义过矩阵X，就是我们的训练样本堆到个列，得到一个m\*nx的矩阵。那么在神经网络里面也一样，最后得到双层神经网络仍然是四个公式，只不过所有样本都表示在矩阵中，公式如下：



矩阵也同理，这样表示可以帮助你思考，这些矩阵，比如A和Z，横向的话，我们有对多有训练样本的用指标排序，所以横向指标就对应了不同的样本，当你从左向右扫的时候，就扫过了整个训练集，而在竖向，竖向指标就对应了神经网络里该层的不同节点。比如说，矩阵的最左上角那个，就表示了对应第一个样本，第一个隐藏单元的激活函数值，下面就对应了第二个隐藏单元，一次类推，所以从上到下扫的话就是第一个样本在隐藏单元层从上到下每个节点的激活函数值。

通过这些你就学会了如何向量化不同训练样本的神经网络算法，总结图：



3.5 向量化实现的解释

前面学习了如何将训练样本横向堆叠起来形成矩阵X，就可以导出一个在网络中正向传播算法的向量化实现，说明一下理由为什么这些方程向量化在多样本时是正确的实现。

![多样本向量化原理](<http://upload-images.jianshu.io/upload_images/1779926-626c62ff1a66b982.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

从图中可以看出来，对样本进行堆叠成矩阵后，乘以W矩阵，最后的到的Z其实也就是该层样本的列堆叠。所以可以更好的理解，这是针对单个训练样本，当你处理不同的训练样本时，就将它们堆到各列中。并且如果你想用python广播做矩阵和向量的加法，即关于b的那块，把这些b值加回来，这些值还是对，最后使用了广播，将单独加到各列上。中i代表第几层，它也是一个向量，即在不同节点上的不同b值，所以最后将这个向量加上矩阵上使用了广播技术，结果还是正确的。

所以前面这几节视频就讲的是双层神经网络，后面会讲到更深层次的神经网络基本上也还是重复这两步运算，即向量化计算以及计算，只不过重复的次数更多。

接下来，到目前为止，我们一直用的sigmoid()函数，事实证明，这不是最好的选择，下一节我们深入研究如何使用不同种类的激活函数，其中sigmoid()函数只是其中一个可能选择。

3.6 激活函数

当你要搭建一个神经网络，你可以选择的是选择隐藏层里用那一个激活函数，还有神经网络的输出单元用什么激活函数，到目前为止，我们一直用的sigmoid()函数，但有时其他函数效果要好得多。可供选择的激活函数有哪些呢，在神经网络的正向传播步骤中，有这两步，用的是sigmoid函数，也就是所谓的激活函数。

![对于给定某样本的两步计算](<http://upload-images.jianshu.io/upload_images/1779926-4e28dd754f3a1879.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

在更一般的情况下，我们可以使用不同的函数g(z)，其中g可以使非线性函数，不一定是sigmoid函数，比如说，sigmoid函数介于0和1之间，有个激活函数几乎总比sigmoid表现更好，就是tanh函数，也叫作双曲正切函数，a=tanh(z)。这个函数介于-1到1之间，函数公式为：

， 数学上这实际上就是sigmoid函数平移后的版本。

![tanh()双曲正切函数图像，横轴为z，纵轴为tanh(z)](<http://upload-images.jianshu.io/upload_images/1779926-1f62930eccf777da.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

事实证明，对于隐藏单元如果你让函数g(z)=tanh(z)，具体地看是, 这几乎总比sigmoid函数效果好，因为现在函数介于-1到1之间，激活函数的平均值就更接近于0。有时候当你训练学习模型，你可能需要平移所有数据，让数据平均值为0，使用tanh函数而不是sigmoid函数也有类似数据中心化的效果，使得数据的平均值接近0，而不是0.5，实际上这让下一层的学习更方便。将在第二门课程中，详细讨论这一点，也会介绍算法优化，但这里要记住一点，几乎不用sigmoid激活函数了，tanh函数几乎在所有场合都更优越。

一个例外就是输出层，因为如果y是0或1，那么你希望介于0和1之间更合理，而不是-1到1之间，所以会用sigmoid激活函数的一个例外场合是使用二分分类的时候。在这种情况下，你可以使用sigmoid激活函数作为输出层。所以在二分分类输出层，，在隐层里用tanh函数，输出层用sigmoid函数，所以不用层的激活函数可能不一样，有时候为了表示不用层的不用激活函数，会用上标方括号[i]来表示不同的激活函数，表示和不同，同样的方括号上标表示不同的层次。

现在sigmoid函数和tanh函数都有一个缺点，就是如果z非常大或非常小，那么导数的梯度，或者说这个函数的斜率可能就很小，所以z很大或很小的时候，函数的斜率接近于0，这样会拖慢梯度下降算法。在机器学习里面，比较受欢迎的一个工具是ReLU，修正线性单元，公式为a= max(0,z)。 只要z为正，导数就为1，当z为负时，斜率为0。如果你实际使用这个函数，当z刚好为0时，导数是没有定义的。

![ReLU()函数图像](<http://upload-images.jianshu.io/upload_images/1779926-dc3c4617bc7ce7a2.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

但如果你编程实现，那么你得到z刚好等于00000的概率很低，所以实践中不用担心这一点，你可以在z=0时，给导数赋值，你可以赋值为1或0，那样也是可以的，虽然ReLU函数在0点不可微。

在选择激活函数时有一些经验法则，如果你的输出值是0和1，如果你在做二分分类，那么sigmoid函数很适合作为输出层的激活函数，然后其他单元都用ReLU函数，现在已经变成激活函数的默认选择了。如果你不确定隐藏层应该用哪个，那么就用ReLU函数作为激活函数，这是大多数人都为用的，虽然人们有时候也会用tanh激活函数。

当然ReLU函数也有缺点，它的缺点是当z为负时，导数等于0，（理解：是不是在梯度下降时，导数就为0，参数就不会再修改了呢。但是在反向传播过程中，z的值大多不是为的。）在实践中这没什么问题，但ReLU还有另外一个版本，叫做带泄漏的ReLU，（the leaky ReLU），当z为负时，函数不再为0，它有一个很平缓的斜率。这通常比ReLU激活函数更好，不过实际上使用的频率没有那么高，这些里面就选一个就好了，我一般就用ReLU，ReLU和leaky ReLU的好处在于对于很多z空间，激活函数的导数即激活函数的斜率和0差很远，所以在实践中使用ReLU激活函数，你的神经网络的学习速度通常会快很多，比使用tanh或sigmoid函数快很多，主要原因在于ReLU没有这种函数斜率接近于0时，减慢学习的效率。

对于z的一半的学习范围，ReLU函数的斜率为0，但是在实践中有足够多的隐藏单元，令z大于0，所以对大多数样本来说还是蛮快的。

![几种激活函数的图像对比](<http://upload-images.jianshu.io/upload_images/1779926-7b2e73f1f656ed0d.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

快速回顾：不同激活函数的利弊

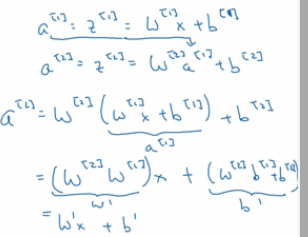
1. Sigmoid函数除非在二分分类的输出层，不然绝对不要用或者几乎从来不用，我几乎没怎么用过，因为tanh几乎在所有场合都更优越。
2. 常用的默认激活函数是ReLU函数，如果你不确定用哪个，就用这个，或者你想换一个，可以试试leaky ReLU，公式可能是(0.01z , z)。所以a是0.01z和z两者之间的最大值，这样的函数左边会稍微拐一下。你可能会问，为什么那个常数是0.01，你也可以把它设成学习函数的另一个参数，有人说这样效果很好，但很多有人这么做，你可以试试，得到一个效果比较好的常数，然后就一直用这个。

深度学习其中一个特点，在建立神经网络时经常有很多不同的选择，比如隐藏单元数，激活函数，还有如何初始化权重，接下来会讲，有很多这样的选择，所以很难选择，以及定下一个准则，来确定什么参数最适合的问题。所以后面的课程会说到在这个行业里的热门选择，冷门选择等。但是对于你的应用，你的应用的特质，事实上很难预先准确知道什么参数最有效。所以一个建议是，如果你不知道什么激活函数最有效，你可以先试试在你的保留交叉验证数据集上跑跑，或者在开发集上跑跑，看看哪个参数效果更好，就用哪个。下一节将学到神经网络确实需要某种非线性激活函数。

3.7 神经网络为什么需要非线性激活函数

事实证明，要让你的神经网络能够计算出有趣的函数你必须使用非线性激活函数，如果去掉计算完z后面再使用激活函数计算z的那步，或者使用a = g(z) = z，即线性激活函数，或者说恒等激活函数，因为直接把输入值输出了。为了说明问题，我们看看会怎么样，如果你这么做，那么这个模型的输出y或不过是你输入特征的线性组合，可以得到下面推导：

![使用恒等激活函数的推导](<http://upload-images.jianshu.io/upload_images/1779926-c3910f5d8146ac13.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)



可以发现，如果使用线性或恒等激活函数，那么神经网络只是把输入线性组合再输出，我们稍后会说到神经网络有很多层的神经网络，很多隐藏层，事实证明，如果你使用线性激活函数或者没有激活函数，那么无论神经网络有所少层，一直在做的只是计算线性激活函数，那还不如直接去掉全部隐藏层。也就是说，在双层神经网络中，如果隐藏层使用线性激活函数，那么和没有任何隐藏层的标准逻辑回归是一样的，可以自己证明一下。所以要点就在于，线性隐藏层一点用都没有，因为两个线性函数的徐鹤本身就是线性函数，所以除非你引入非线性，那么你无法计算更有趣的函数，网络层数再多也不行。只有已个地方可以使用线性激活函数，g(z) =z就是如果你要机器学习学习的是回归问题，所以y是一个实数，比如说，你想预测房地产价格，所以y是一个实数，不是0和1。那么用线性函数也许可以，但这些隐藏单元不能用线性激活函数，可以使用ReLU或者tanh或者leaky ReLU或者别的函数，所以唯一可以用线性激活函数的地方通常是输出层，

除了这种情况，会在隐藏层用线性激活函数的可能除了与压缩有关的一些非常特殊的情况，在那之外，使用线性激活函数非常少见。下一节讨论梯度下降的基础，告诉你如何估计，如何计算，单个激活函数的导数，斜率。

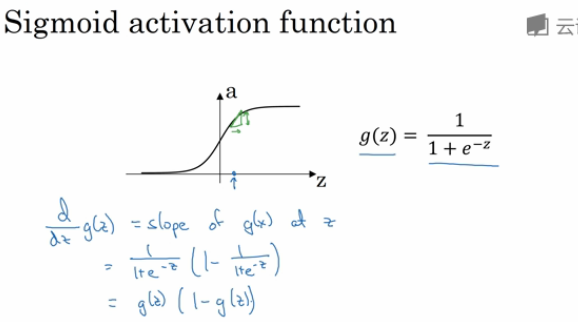
![有关回归问题输出可以使用线性激活函数的讨论](<http://upload-images.jianshu.io/upload_images/1779926-82ff247e7795106d.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

3.8 激活函数的导数

当你对你的神经网络使用反向传播时，需要计算激活函数的斜率或者导数，看看激活函数的选择以及如何计算这些函数的斜率。

1. Sigmoid函数

![sigmoid函数导数](<http://upload-images.jianshu.io/upload_images/1779926-af2a7bc49788b5cc.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)



我们看看这个式子是否合理，首先，如果z很大，z=10， 那么g(z)接近于1，那么，是符合的，因为当z很大时，斜率接近于0。相反如果z等于-10，那么g(z)接近于0，也是符合的，最后再z=0处，g(z)=1/2，，可以证明到这是z=0时正确的导数值。这就是sigmoid函数。这个函数的优点是，a=g(z), ，我们已经把a值计算出来了，所以直接带入得到结果。

1. Tanh函数

a=g(z)=tanh(z)，那么导数。

![tanh函数导数推导](<http://upload-images.jianshu.io/upload_images/1779926-2b36154ac84ede03.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

1. ReLU函数 and leaky ReLU函数

在ReLU函数中，在z=0处导数不存在，但是可以令该点导数为1或者0，折无关紧要，如果对优化术语很熟悉，就变成所谓的激活函数g(z)的次梯度，这样的梯度下降法仍然有效。但你又可以这么想，其实z=0的概率真的很小很小，所以你将z=0处的导数设成哪个值实际无关紧要。所以在实践中，人们一般这么定z的导数。最后如果你在训练自己的网络时，使用的leaky ReLU函数，说明一下，z精确为0时的梯度技术上是没定义的，但你可以写一段代码去定义这个梯度。

![ReLU函数和leaky ReLU函数导数](<http://upload-images.jianshu.io/upload_images/1779926-cb685c7da89ad89e.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

最后有了这些基础，就准备好如何在你的神经网络上实现梯度下降算法了。

* 1. 神经网络梯度中的下降法

这一节将学习梯度下降算法的具体实现，如何处理单隐层神经网络。会提供需要用

到的方程来实现反向传播，或者说梯度下降算法，所以对于一个双层神经网络，会有这些参数，和，，还有个，有时也用表示有那么多输入特征，个隐藏单元，个输出单元，在上面说到的例子中，我们只介绍过的情况，那么矩阵的维度就是（，），就是维向量，也就是一个列向量。的维度是（，），然后就是维，也是一个列向量。同样地，你还有一个神经网络的成本函数， 假设是在做一个二分分类的问题，那么成本关于各参数的成本函数公式是这样的：

，，，这个其实就是，其真实标签值为。如果是做二分分类，损失函数可能和之前的logistic回归完全一样，所以要训练参数，你的算法需要做梯度下降，在训练神经网络时，随机初始化参数很重要，而不是初始化全为0，后面会解释。当你把参数初始化成某些值之后，每个梯度下降循环都会计算预测值，所以基本上，你要计算i=1到m的值，然后需要计算导数，d即成本函数对参数的导数，以及d，并且还有对，的导数，然后梯度下降最后会更新，， ， 对，的更新也类似，以上就是梯度下降一次迭代循环需要做的工作。然后重复很多次，直到你的参数是在收敛。前面说过了如何计算预测值，如何计算输出，即正向传播的过程，后面给出你需要的公式，求这些导数的公式。

Forward propagation：

g^[1]

Back propagation：

这里有个细节，np.sum是pythron的numpy命令，用来对矩阵的一个维度求和，水平相加求和，而加上开关就是防止python直接输出这些古怪的秩为1的数组，它的维度是(n,)，确保输出的是个矩阵，对于这个向量输出的维度是（n，1），其实应该是（），

这是在二分分类例子中，所以就是（1,1）是个实数。但是不用担心，以后会看到真正需要考虑多维的情况。

到目前为止，我们所做的和logistic回归非常相似，但当你开始计算反向传播时，后面的差别会比较大。

Back propagation：

如果，那么可能没那么重要，这只是一个实数，而这里的是个（）维向量，所以如果你想用python用np.sum输出这个矩阵的维度的矩阵而不是形式很怪的一维数组，那样可能会把后面的计算步骤搞乱，还有一种办法是不需要用参数，但要显示的调用reshape把np.sum的输出结果写成矩阵形式就是你希望出现的db的矩阵形式。

所以正向传播有4个公式，而反向传播有六个方程，下一节讲如何导出反向传播算法的六个式子的。

![反向传播的手写版笔记](<http://upload-images.jianshu.io/upload_images/1779926-0680a9bb1a38cd7a.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

* 1. 理解反向传播过程的原理

![logistic梯度下降求导数对比图](<http://upload-images.jianshu.io/upload_images/1779926-6d284079a25a4e66.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

实现向后传播有个技巧，你必须确保矩阵的维度互相匹配，W和dW的维度应该是一致的，可以想想不同矩阵的维度都是怎样的，只要确定这些矩阵运算的维度互相匹配。这就已经可以消除反向传播实现中的很多bug了。

![双层神经网络二分分类梯度下降导数推导](<http://upload-images.jianshu.io/upload_images/1779926-dceaefa2d14d1f3a.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

![简单神经网络检查梯度下降公式维度匹配](<http://upload-images.jianshu.io/upload_images/1779926-f1301ed5e707a04b.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

关于神经网络梯度下降过程求导数，其实最不熟悉的就是关于参数为矩阵的导数怎么表示，只要把握住，参数求到后的维度与原参数维度相同，也不难理解六个步骤中转置是为什么了。我们有6个关键步骤，如下：

![单样本梯度下降导数计算6个公式](<http://upload-images.jianshu.io/upload_images/1779926-8e0c1a4078acb605.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

接下来必须写出反向传播过程，如果你每次训练单个样本的话，花费代价很大，我们需要将训练样本向量化，

![单个样本与所有样本向量化的梯度下降导数计算](<http://upload-images.jianshu.io/upload_images/1779926-4aa6434a493954f7.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

在所有机器学习领域，反向传播算法的推导，是最复杂的数学之一，矩阵的导数要用链式法则推出来，还要验证自己的推导是否正确。关于在实现神经网络，如何初始化你的神经网络的权重，事实证明，初始化你的参数，不是全都初始化为0，而是随机初始化，对于训练你的神经网络而言，这一点很重要。

* 1. 随机初始化参数

对于logistic回归，可以把权重初始化为0，但如果将神经网络的各参数数组全部初始化为0，再使用梯度下降算法，那会完全无效。探讨一下为什么呢？

假设你有一个有两个输入特征的神经网络，并且隐藏层有两个隐藏单元，那么，，所以和隐藏层相关的矩阵是(2,2)，假设全部初始化为0，那么，并且，将偏执项b初始化为0实际上是可行的，但是W全为0就有问题了，这种初始化形式的问题在于你给网络输入任何样本，你的和是一样的，并且两个隐藏单元做完全一样的计算，用的相同的激活函数，当你计算反向传播时，出于对称性，d和d也是相同的。假设输出的权重也是一样的，，如果以这种方式初始化那么两个隐藏单元就完全一样了，所以这就是所谓的完全对称，意味着节点计算完全一样的函数，我们可以通过归纳法证明，每次训练迭代之后，两个隐藏单元依然在进行完全一样的计算，因为很大一部分网络记忆dW是这样的一个矩阵，其中每一行的内容都是完全相同，然后我们执行一次权重更新，，你会发现每次迭代后的形式都是第一行和第二行的内容完全一样，通过归纳发现第二次第三次迭代也一样，因为节点都是在做一样的运算，并且两个隐藏单元对输出单元的影响也一样大，在一次迭代之后，同样的对称性依然存在，两个隐藏单元依然是对称的，无论你训练神经网络多长时间，两个隐藏单元依然在计算完全一样的函数，所以在这种情况下，多个隐藏单元真的没有意义，因为你需要两个不同的隐藏单元去计算不同的函数。

这个问题的解决方案是随机初始化所有参数，令，这可以产生参数为(2,2)的高斯分布随机变量（2,2），然后再乘上0.01，将权重初始化为很小的随机数，然后关于b，完全没有这个对称性问题，所以初始化为0是可以的，，因为只要W随机化，一开始就是用的不同的隐藏单元计算不同的函数。对于，也同理。关于这个常数是0.01，为什么100什么的呢，实际上我们通常喜欢把权重矩阵初始化为非常小的随机数，因为如果你用的tanh或者sigmoid函数，或者你在输出层有一个sigmoid函数，如果权重过大，计算激活函数值时，计算出来的z就会很大或很小，在这种情况，可能落在激活函数比较平缓部分，斜率接近于0，意味着梯度下降算法会非常慢，所以学习效率就会变低。如果你的神经网络中没有任何sigmoid函数或者tanh函数，那么问题不大，但如果你在做二分分类，那么久不希望参数过大，所以0.01是比较合理的。实际上，也有比0.01更好的常数，但当你训练一个单层神经单元或者相对较浅的神经网络，0.01是可以的。如果训练一个比较深的，可能需要试试其他0.01以外的参数。但不管怎样，初始化参数一般都很小。