1. 深度学习的实用层面

继续学习如何有效运作神经网络，内容涉及超参数调优，如何构建数据，以及如何确保优化算法快速运行，从而使学习算法在河里时间内完成自我学习。第一章课程首先说说神经网络机器学习中的问题，然后是随机神经网络，还会学习一些确保神经网络正确运行的技巧。

* 1. 训练/开发/测试集

在配置训练，验证和测试数据集的过程中做出正确决策会很大程度上帮助大家创建高效的神经网络，训练神经网络时，我们需要做出很多决策，例如，神经网络分多少层，每层有多少个隐藏单元，学习率是多少，各层采用哪些激活函数……创建新应用的过程中，我们不可能从一开始就准确的预测出这些信息和其他超参数，实际上，应用型机器学习是一个高度迭代的过程，idea->code->experiment->idea…… 现如今，深度学习已经在自然语言处理，计算机视觉，语音识别以及结构化数据应用等众多领域取得巨大成功。结构化数据无所不包，从广告到网络搜索，其中网络搜索不仅包括网络搜索引擎，还包括购物网站，从所有根据搜索栏词条传输结果的网站，再到计算机安全，物流，比如判断司机去哪接送货，范围之广，我发现可能有自然语言处理方面的人才想踏足计算机视觉领域或者经验丰富的语音识别专家想投身广告行业，又或者，有的人想从电脑安全领域跳到物流行业，在我看来，从一个领域或者应用领域得来的直觉经验通常无法转移到其他应用领域。最佳决策取决于你所拥有的数据量，计算机配置中出入特征的数量，用GPU训练还是CPU的具体配置以及其他诸多因素。目前为止，我觉得对于很多应用系统，即使是经验丰富的深度学习行家也不太可能一开始就预设出最匹配的超参数。因此这是一个迭代的过程，循环该过程的效率是决定项目进展速度的一个关键因素，而创建高质量的训练数据集，验证机和测试集也有助于提高循环效率。

假设一个训练数据，用长方形表示，通常将这些数据划分成几部分，一部分作为训练集，一部分作为简单交叉验证集（hold-out cross validation），有时候也称为验证集（dev set），一部分作为测试集，接下来，我们开始对训练集执行训练算法，通过验证集选择最好的模型，经过充分验证，我们选定了最终模型，然后就可以在测试集上进行评估了，为了无误差评估算法的运行状况。在机器学习发展的小数据量时代，常见做法是将数据三七分，70%训练集，30%测试集，如果没有明确设置验证集，也可以按照60%训练，20%验证，20%测试集来划分，这是前几年机器学习领域普遍认可的最好的实践方法，比如有100条10000条数据，这么划分是合理的。但是在大数据时代，我们现在的数据量可能是百万级别的，那么验证集和测试集占数据总量的比例会趋向于变得更小，因为验证集的目的就是验证不同的算法，检验哪种算法更有效，因此验证集要足够大才能评估，比如2个甚至10个不同算法，并迅速判断出哪种算法更有效，可能不需要拿出20%的数据作为验证集，比如有100万条数据，那么取一万条数据便足以进行评估，找出其中表现最好的1-2种算法，同样地根据最终选择的分类器，其中一万条作验证集，一万条作测试集，最后发现，训练集占98%，验证集和测试集各占1%，对于数据量过百万的应用，训练集可以占到99.5%，验证和测试集各占0.25%，或者验证集0.4%，测试集0.1%。

总结一下，在机器学习中，我们通常把数据集划分为三个部分，数据集规模相对较小的，适合传统的划分比例，数据集规模较大的，验证集和测试集可以占到数据总量的20%或10%以下，后面会给出如何划分验证集和测试集的具体指导。但现在我们看到的一个趋势是在深度学习中，越来越多的人在训练和测试集分布不匹配的情况下进行训练，假设你要构建一个用户可以上传大量图片的应用程序，目的是找出并呈现所有猫咪图片，训练集可能是网上下载的猫咪图片，而验证集和测试集是用户在这个应用上上传的猫的图片，就是说，训练集可能是从网络上抓下来的图片，结果许多网页上的猫咪图片分辨率很高，很专业，后期制作很精良，而用户上传的可能是用手机随意拍摄的，像素低比较模糊，这两类数据有所不同，针对这种情况，根据经验建议大家要确保验证集和测试集的数据来自同一分布，因为你要用验证集来评估不同的模型，尽可能的优化性能，因为训练集和测试集都来自于同一分布，但由于深度学习算法需要大量的训练数据，为了获取更大规模的训练数据集，可以采用大量的创意策略，比如网页抓取，代价就是训练集数据与验证集和测试集数据有可能不是来自同一分布，但只要遵循这个经验法则，就会发现机器学习算法会变得更快。最后一点，就算没有测试集也不要紧，测试集的目的是对最终所选定的神经网络系统做出无偏评估，如果不需要无偏评估，也可以不设置测试集。所以如果只有验证集没有测试集，我们要做的就是在训练集上训练，尝试不同的模型框架，在验证集上评估这些模型，然后迭代并选出适用的模型，因为验证集中已经涵盖测试及数据，其不再提供无偏性能估计，当然如果你不需要无偏估计，那就再好不过了。‘

在机器学习中如果只有一个训练集和一个验证集，而没有独立的测试集，这种情况验证集则被称为测试集，不过在实际应用中，人们只是把测试集当成简单交叉验证集在使用，因为把验证集数据过度拟合到了测试集。如果某团队跟你说他们只设置了一个训练集和一个测试集，改称为训练验证集。

所以说，搭建训练验证集和测试集能加速神经网络的集成，也可以更有效地衡量算法的偏差和方差，从而帮助我们更有效地选择合适的方法来优化算法。

* 1. 偏差/方差

这两个概念易学难静，即使你自认为已经理解了偏差和方差的基本概念，关于深度学习的误差问题，另一个趋势是对偏差和方差的权衡研究甚浅，总是分别考虑偏差和方差。

假设下面图中的就是数据集，如果图一中给这个数据集拟合一条直线，可能就得到一个逻辑回归拟合，但它并不能很好地拟合该数据集，偏差高，称为“欠拟合”（underfit the data，high bias），相反地，如果我们拟合一个非常复杂的分类器，比如深度神经网络或者含有隐藏单元的神经网络，可能就非常适用于这个数据集，如图三，但是这也不是一种很好的拟合方式，分类器方差较高，过度拟合（overfit the data，high variance），在两者之间，还有一种复杂程度适中，数据拟合适度的分类器，看起来更加合理，称为“适度拟合”（just right），是介于过拟合和欠拟合中的一种。

![平面数据集拟合效果三种情况](<http://upload-images.jianshu.io/upload_images/5355764-0030e328b3d73e30.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

在这样只有x1和x2两个特征的二维数据中，可以绘制数据，将偏差和方差可视化，在多维数据中，绘制数据和可视化分割边界无法实现，但我们可以通过几个指标来研究偏差和方差。理解偏差和方差的两个关键数据是训练集误差和验证集误差，为了方便论证，假设我们辨别图片中的小猫，肉眼识别是不会错的，假定训练集错误率是1%，验证集错误率是11%，可以看出，训练集设置的非常好，而验证集设置相对较差，可能过分拟合了训练集，某种程度上，验证集并没有充分利用较差验证集的作用，这种情况称为“高方差”，通过查看训练集误差和验证集误差，就可以诊断算法是否具有高方差。也就是说，衡量训练集误差和验证集误差得出不同结论。假设训练集误差15%，测试集误差16%，假设识别猫图案例中人的错误率为0%，算法并没有在训练集中得到很好训练，如果数据的拟合度不高，也就是欠拟合，这种情况“高偏差”，相反它对与验证集产生的结果却是合理的，验证集的错误率只比测训练多了1%。再举个例子，如果训练集错误率为15%，高偏差，但验证集错误率更高30%，这种情况认为算法偏差高，因为在训练集上结果不理想，方差也很高，这是方差偏差都很糟糕。如果分别为0.5%，1%，偏差和方差都很低。这些分析都是基于假设预测的，假设人眼识别的误差接近于0，一般来说最优误差也被称为贝叶斯误差，所以最优误差接近0%。如果最优误差特别高，比如15%，再看看（15%，16%）的错误率对训练集来说也是非常合理的，偏差不高，方差也很低。当所有分类器都不适合的时候，如何分析偏差和方差呢。比如图片很模糊，即使是人眼，或者没有系统可以准确无误的识别图片，这种情况下，最优误差会更高，那么分析过程就要改变了。暂时不讨论这些细微的差别，重点是通过查看训练集误差，我们可以判断数据拟合情况，至少对于训练数据是这样，可以判断偏差有多高查看错误率有多高，当完成训练集训练开始验证集验证时，可以判断方差是否过高。以上分析的前提都是假设基本误差很小，训练集和验证集数据都来自相同分布，如果没有这些前提，分析过程更加复杂。

![根据训练集误差和验证集误差分析偏差方差的例子](<http://upload-images.jianshu.io/upload_images/5355764-44ed3fd2d27d1a6a.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

前面介绍了高偏差和高方差的情况，对优质分类器有了一定的认识，偏差和方差都高是种什么情况呢，这种情对于两个衡量标准来说都是非常糟糕的。对于上面的平面数据分类的例子，线性回归一条直线会产生高偏差，因为它的数据拟合低，但如果稍微改变一下分类器，把那两个两侧的分出来，就会过度拟合部分数据，这样分分类器就绘具有高偏差和高方差，偏差高的原因是它近乎于一条直线，并且拟合数据，像中间图的情况的二次曲线能够很好的拟合数据，而紫色的高偏差高方差的曲线中间部分灵活性很高却过度拟合直线边上的那两个样本，所以这类分类器偏差很高，因为几乎是线性的，而采用曲线函数或者二次元函数会产生高方差，因为曲线灵活性太高，以致于过度拟合了这两个错误样本和中间这些活跃数据。对于二维数据看起来很不自然，但是对于高维数据，有些区域偏差高，有些区域方差高，在高维数据中采用这种分类器就相对比较合理。

![高方差高偏差紫色分类器](<http://upload-images.jianshu.io/upload_images/5355764-b115984660d6d777.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

这一节讲了如何通过分析训练集训练算法产生的误差和验证集验证产生的错误率来诊断算法是否存在高偏差高方差，或者都高，或都不高，根据算法偏差和方差的具体情况，决定接下来你要做的工作。

* 1. 机器学习基础

训练神经网络时用到的基本方法，初始模型训练完成后首先要知道算法的偏差高不高，如果偏差过高，试着评估训练集或训练数据的性能，如果偏差实在过高，甚至无法拟合数据集，要做的就是选择一个新网络，比如更多层或者更多隐藏单元的神经网络或者花费更多时间来训练网络，训练算法，或者尝试更先进的优化算法。可能有用也可能没用，后面我们也会看到许多不同的神经网络架构，或许你能找到一个更适合解决此问题的新网络架构。你必须去尝试不管不是有用，不过采用规模更大的网络通常都是有所帮助的，延长训练时间不一定有用但是也没什么坏处，训练学习算法时候，不断尝试这些方法，直到解决偏差问题，直到可以拟合数据为止，至少可以拟合数据集。如果你的网络足够大，通常可以很好地拟合训练集，只要你能扩大网络规模，如果图片很模糊，算法可能就拟合这张图片，但如果人可以识别，误差也不是很高，那么训练一个更大的网络，至少可以很好地拟合或者过拟合数据集。一旦偏差降低到可接受的数值，检查一下方差有没有问题，为了评估方差，要查看验证集性能，能从一个性能理想的训练集，推断出验证集的性能是否也理想。如果方差过高，最好的解决办法就是采用更多的数据，但有时候我们无法获得更多数据，也可以通过正则化来减少过拟合。不得不反复尝试或者找到更适合的神经网络架构，可能会同时减少方差和偏差。最终就是不断尝试直到找到一个低偏差，低方差的框架。

两个需要注意的，第一点，高偏差和高方差是两种不同的情况，要尝试的办法也可能完全不同，通常会用训练验证集来诊断算法是否偏差或方差问题，根据结果选择部分尝试方法。第二点，在早期机器学习阶段，关于所谓的偏差方差权衡的讨论，原因是可以尝试很多方法，可以减少偏差，增加方差，或者都减少，但是在早期，没有太多工具可以做到，只减少其中一个不影响另一个。但是现在大数据是到，只要持续训练一个更大的模型，有充足的数据，也并非只有这两种，那么只要正则适度，通常构建一个更大的神经网络就能在不影响方差的前提下，减少偏差，而采用更多数据通常可以在不过多影响偏差的同时减少方差。这两步实际要做的就是训练网络，选择网络或者准备更多的数据，现在我们有工具可以做到，在减少其中一个不对另一个产生过多影响。这就是深度学习对监督学习大有裨益的一个重要原因，也是我们不用过多关注如何平衡方差和偏差的一个重要原因。但有时有很多选择，减少一个不增加另一个，最终我们会得到一个比较规范化的网络，正规化，训练一个更大的网络几乎没有任何负面影响，而训练一个大型神经网络的主要代价就是训练时间，而前提就是网络是比较规范化。

* 1. 正则化

正则化是一种非常实用的减少方差的方法，正则化是会出现偏差方差权衡的问题，偏差可能略微增加，如果网络足够大，增幅不会太高。当你的神经网络过度拟合数据，即高方差的情况，首先想到的办法就是正则化，另一个方法就是准备更多的数据。但可能获取数据的成本很高不能随时都获取到，但正则化通常都有助于避免过度拟合或减少网络误差。

正则化原理：

使用逻辑回归来解释， 原先求成本函数J的最小值，成本函数为：,参数包含了一些训练数据和不同数据中个体预测的损失，w和b是逻辑回归的两个参数，w是一个多维度参数矢量，b是一个实数。在逻辑回归函数中加入正则化，只需要填入，也就是正则化参数，读作w平方的范数。

W欧几里得范数的的平方等于（j值从1到n\_x的平方的和）

理解，w范数平方就是w矩阵里的每一个数的平方求和，因为在逻辑回归里面w是一个（）的矩阵，就是（1，），所以（1，）\*（）最后得到一个数，刚好也是w矩阵里面每个数的平方之和。的表示是因为只有一行，所以i参数不用管。

这种方法就叫做L2正则化，因为这里用了欧几里德法则，被称为参数w的L2范数。

为什么只正则化参数w，不再加上参数b呢，在后面加上\*，你可以这么做，但是通常习惯省略不写，因为w通常是一个高维矢量，已经可以表达高偏差问题，w可能含有很多参数，我们不可能拟合所有参数，而b只是单个数字，所以w几乎涵盖所有参数，如果加了参数b，其实也没多大影响，就是很多参数中的一个，通常省略。

L2正则化是最常见的正则化类型，可能也听说过L1正则化，它在J后面加的不是L2范数，也被称为参数w向量的L1范数，如果用的是L1正则化，w最终会是稀疏的。也就是说w向量中会有很多0，有人说这样有利于压缩模型，因为集合中参数均为0，存储该模型所占用的内存更少，实际上，虽然L1正则化是模型变得稀疏，但是却没有降低太多存储内存，所以这并不是L1正则化的目的，至少不是为了压缩模型，人们在训练网络时都比较倾向于使用L2正则化。

所以是正则化参数，通常使用验证集或交叉验证来配置这个参数，尝试各种各样的数据，寻找最好的参数，要考虑训练集之间的权衡，把参数正常值设置为较小值，这样可以避免过拟合。所以也是一个需要设置的超参数，另外，在代码中，为了与python的lambda区分，我们把记为lambd。这就是在逻辑回归函数中实现L2正则化的过程。

![逻辑回归中的L2正则与L1正则](<http://upload-images.jianshu.io/upload_images/5355764-6339761e4251b195.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

在神经网络中，实现正则化如下”

），即每一层的W参数的范数平方和。

该矩阵范数被称作“弗罗贝尔尼乌斯范数”（Frobenius），鉴于线性代数一些什么晦涩的原因，不称为矩阵的L2范数，也表示一个矩阵的所有元素的平方和。

那如何使用该范数实现梯度下降呢，根据J对w求偏导，后面加上的范数部分只需要原来dWL加上后面部分求导，即原来的。使用重新计算好的，它的定义含有代价函数求导和相关参数再加上额外的正则化项，这也是L2正则化有时被称为“权重衰减”。将代入更新的步骤中，发现，只是在原来的更新操作的基础上，相当于把变为原来的（1-）倍。该正则项说明，不管是什么，我们都试图把它变得更小，实际上，相当于给矩阵乘以了（1-）倍的权重，这是一个小于1的数，因此L2范数正则化也被称为“权重衰减”。

![神经网络中的正则：Forbenius范数](<http://upload-images.jianshu.io/upload_images/5355764-9d418c7b73816cef.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

* 1. 为什么正则化可以减少过拟合

通过两个例子来说明正则化为什么可以预防过拟合以及减少方差：

还是用上一节的那个平面二维数据分类做例子，看一下这个庞大的深度拟合神经网络，事实上不够大深度也不够，但是可以想象这是一个过拟合的神经网络，代价函数含有参数w和b，并且添加上正则项则是

它可以避免数据权值矩阵过大，为什么压缩L2范数或者说是罗贝尼乌斯范数可以减少过拟合，直观上理解就是**如果正则化参数设置得足够大，权重矩阵被设置为接近0的值，就是把多隐藏单元的权重设置为0**，于是基本上消除了这些隐藏单元的许多影响，如果是这样，这个被大大简化了的神经网络会变成一个很小的神经网络，小到如同一个逻辑回归单元，可是深度却很大，它会使这个网络从过拟合状态更接近于左图的高偏差状态，但是会存在一个中间值，于是会有一个接近“just right”的中间状态。直观理解就是增加到足够大，W会接近于0，实际上是不会发生这种情况的，我们尝试消除或至少减少许多隐藏单元的影响，最终这个网络会变得简单，这个神经网络越来越接近逻辑回归。感觉上认为大量隐藏单元被完全消除了，实际上不是这样，实际上该神经网络所有的隐藏单元依然存在，只是影响变小了，从而使得神经网络变得简单了，这样不容易发生过拟合。不知道这样的只管理解有没有用，但是在编程过程中执行正则化时，会实际看到方差减少的结果。

关于加粗部分的理解，即为什么参数增加越大，w就会趋近于0,加入没有正则项，那么每一次更新w，就是通过，而加上正则项之后，d的值即是代价函数求偏导，那么对应的导数就应该增加部分，把新的d代入更新梯度值的计算，最终结果为。由于（1-）是个小于1的数，那么越大，整体这部分的值就越小，即w越趋近于0。

上面这段话是通过神经网络图示，w参数里的数越趋近0，隐藏单元的影响减小，或者近似的看成这些隐藏单元相当于不存在，因此把一个复杂的可能很高次的曲线函数变成一个简单的线性函数，从高方差状态近似转换成高偏差的状态，减少了过拟合。接下来我们从激活函数来理解，为什么正则参数增大，会使模型更趋向线性化。

假设我们使用的是tanh双曲激活函数，函数图像如图，该函数中，只要z非常小，如果z只涉及少量参数，即在图像上x轴原点附近的一段，利用双曲正切函数的线性状态，只要z可以扩展为更大的值或更小的值，即函数图像的左右两部分，激活函数就变得非线性。然而现在，正则化参数很大，激活函数的参数z会变小，因为代价函数中的参数被阻止变大了（penalized惩罚），即w参数变小了的，所以w很小时， 对应的z也就变得很小，特别是，当z小到在图像的中间部分范围内，大致呈线性，每层几乎都是线性的，和线性回归函数一样。在第一节讲过，为什么激活函数要选非线性的，就是因为如果每层都是线性的，那么整个网络就是线性网络，中间有再多的神经元都相当于没有，即使是一个非常深的深层网络，因具有线性激活函数的特征，最终都只是计算线性函数，因此它不适合非常复杂的决策以及过度拟合数据集的非线性决策边界。而本节中我们探讨的是减小模型的复杂程度，就是要防止过拟合，刚好可以应用线性这一特征。

![从激活函数角度理解正则化参数增大，模型会变简单](<http://upload-images.jianshu.io/upload_images/5355764-b9ace6f9a7038a1d.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

总结一下，如果正则化参数变得很大，w很小，z也相对变小，此时忽略b的影响，实际上，z的取值范围很小，这个激活函数会相对呈线性，整个网络就会计算离线性函数近的值，这个线性函数会很简单，不是一个极复杂的高度非线性函数，不会发生过拟合。

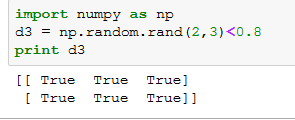
总结正则化之前，有个小建议，在增加正则化项时，应用之前定义的代价函数J通过增加一个L2正则项来修改它，目的是预防权重过大（penalized the weight being too large），如果你使用的梯度下降函数，在调试梯度下降时，其中一步就是把代价函数设计成图最下方那样，它表示梯度下降的调幅数量，即每次训练的时候画出的一个，迭代到多少次时对应的代价是多少的一个图像，可以看到代价函数对于梯度下降的每个调幅都单调递减。如果你实施的是正则化，牢记J已经有全新的定义，即新的公式，如果你用的原来的计算方式，可能看不到单调递减的现象，为了调试出单调递减，使用包含正则化后的代价函数J，否则函数J可能不会在所有调幅范围内都单调递减。

* 1. Dropout正则化

除了L2正则化，还有一个非常实用的正则化方法—dropout（随机失活）。工作原理如下：假设你训练如图的一个神经网络，它存在过拟合，dropout所要做的就是复制这个神经网络，遍历网络的每一层，并设置消除神经网络中节点的概率，假设网络中的每一层每个节点都以抛硬币得到方式设置概率，那么每个节点保留或消除的概率都是0.5，设置完节点概率，就消除一些节点，然后删除从该节点进出的连线，然后得到一个节点更少规模更小的网络，然后用backprop方法进行训练，图中右图就是网络节点精简后的一个样本，对于其他样本，照旧以抛硬币的方式设置概率，保留一类节点集合，删除其他类型的结点集合，对于每个训练样本都将采用一个精简后的神经网络来训练它，这种方法看似有点奇怪，单纯遍历节点，编码也是随机的，不过确实有效。不过可想而知，我们针对每个样本训练规模极小的网络，最后你可能会认识到为什么要正则化网络，因为我们要训练极小的网络。

![dropout正规化形象解释](<http://upload-images.jianshu.io/upload_images/5355764-43e383762b302b9b.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

如何实施dropout呢，方法有很多种，介绍一下最常用的一种：inverted dropout（反向随机失活），出于完整性考虑，使用一个三层网络来举例说明，编码中很多用到3的地方，这里只说明如何在某一层中实施dropout。首先声明向量d，表示一个三层的dropout向量， 然后看它是否小于某个数。是一个具体数字，比如在上面例子中就是0.5，在这个例子中是0.8，它表示某个隐藏单元的概率，意味着消除任意一个隐藏单元的概率是0.2，它的作用是生成随机矩阵，如果对进行因子分解效果也是一样的，是一个矩阵，每个样本和每个隐藏单元其在中的对应值为1的概率都是0.8，其对应值是0的概率都是20%。



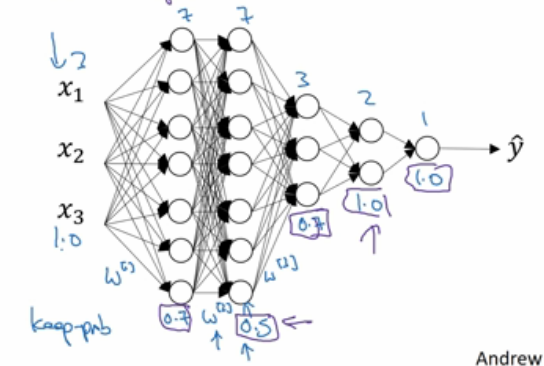
即矩阵随机数字小于0.8，那么是1的概率为80%，为0的概率为20%。接下来要做的就是从第三层中获取激活函数，含有要计算的激活函数，,这里是元素相乘，也可以写成它的作用就是过滤中所有等于0的数，而各个元素等于0的概率只有20%，乘法运算最终把中相应元素归0，如果用python实现，是一个布尔型数组，值为true 或false，而不是0和1，乘法运算依然有效，python会把true和false翻译成0和1。最后我们向外扩展，用它除以0.8或者参数，,解释这一步原因，假设第三隐层上有50个单元，那么是一个一维50的数，如果通过因子分解可以将它拆分成50\*m维，保留和删除它们的概率分别是80%和20%，这意味着最后被删除或者归零的单元平均有10个，，预期是减少20%，为了不影响的期望值，我们需要用，它将会修正或弥补我们所需的那20%，的期望值不会变。这就是反向随机失活方法，它的功能是不论keep\_prob的值是多少，0.8,0.9，甚至是1，如果设置为1，那么不存在dropout。反向随机失活方法通过除以keep\_prob，确保的期望值不会变。事实证明，在测试阶段，当我们评估一个神经网络时，反向随机失活方法使测试阶段变得更容易，因为它的数据扩展问题变少，下节课内容中讨论。目前实施的dropout最常用的方法就是inverted dropout。Dropout的早期迭代版本都没有除以keep\_prob，所以在测试阶段，平均值会变得越来越复杂，现在你用的d向量，你会发现，不同的训练样本清除不同的隐藏单元，实际上如果你通过相同训练集多次传递数据，每次训练数据的梯度不同，则随机对不同隐藏单元归零，有时也并非如此，比如需要将相同隐藏单元归零，第一次迭代梯度下降时，把一些隐藏单元归零，第二次梯度下降时，也就是第二次遍历训练集时，对不同类型的隐藏单元归零，向量d或d3用来决定对第三层中哪些单元归零，无论用正向传播还是反向传播，这里只介绍了正向。如何在测试阶段训练算法，在测试阶段，我们已经给出了x，或是想预测的变量，用的是标准计数法，为测试样本x，在测试阶段不使用dropout函数，尤其是像下列情况， ，，，，，直到最后的，显然在测试阶段，我们并未使用dropout，自然也就不用抛硬币来决定失活概率以及要消除哪些隐藏单元了，因为在测试阶段进行预测时，我们不期望输出结果是随机的，如果在测试阶段应用dropout函数，预测会受到干扰，理论上，你只需要多次运行预测处理过程，每一次不同的隐藏单元会被随机归零，预测处理遍历它们，但是计算效率低，得出的结果也几乎相同，与这个不同程序产生的结果极为相似。Inverted dropout函数在除以keep\_prob时可以记住上一步的操作，目的是确即使在测试阶段不执行dropout来调整数值范围，激活函数的预期结果也不会发生变化，所以没必要在测试阶段额外添加scaling parameter尺度参数，这与训练阶段不同。

* 1. 理解dropout

Dropout可以随机删除网络中的神经单元，做法有点疯狂，为什么可以通过正则化发挥这么大作用呢。

上一节中已经对dropout随机删除网络中神经单元有了个直观了解，好像每次迭代之后，神经网络都会变得比以前更小，因此采用一个较少神经网络好像与使用正则化的效果是一样的，第二个直观认识是，我们从单个神经元入手，这个单元的工作就是输入并产生一些有意义的输出，通过dropout该单元的输入几乎被消除，有可能其中的某两个，又或者是其他的，所以这个神经单元不能依赖任何一个特征，因为特征都有可能被随机清除，或者说该单元的输入也都可能被随机清除，不要把所有赌注都放在一个节点上，也不想给任何一个输入加上太多权重，因为它可能被消除，因此该单元将通过这种方式积极地传播开，并为该单元的每一个输入都增加一点权重，通过传播所有权重，dropout将产生收缩权重的平方范数的效果，和之前讲过的L2正则化类似，实施dropout的结果是压缩权重，并完成一些预防过拟合的外层正则化，事实证明，dropout被正式的作一种正则化的替代形式，与L2对不同权重的衰减还是不同的，它取决于倍增的激活函数的大小。总结一下，dropout的功能类似于L2正则化，与L2正则化不同的是，被应用的方式不同，dropout也会有所不同，甚至更适用于不同的输入范围。

实施dropout的另一个细节是，下图是一个拥有三个输入特征的网络，

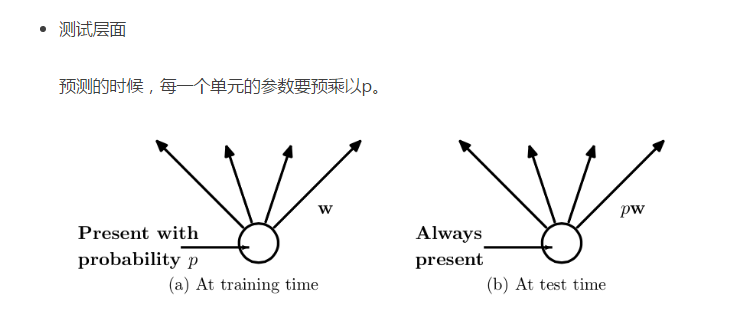


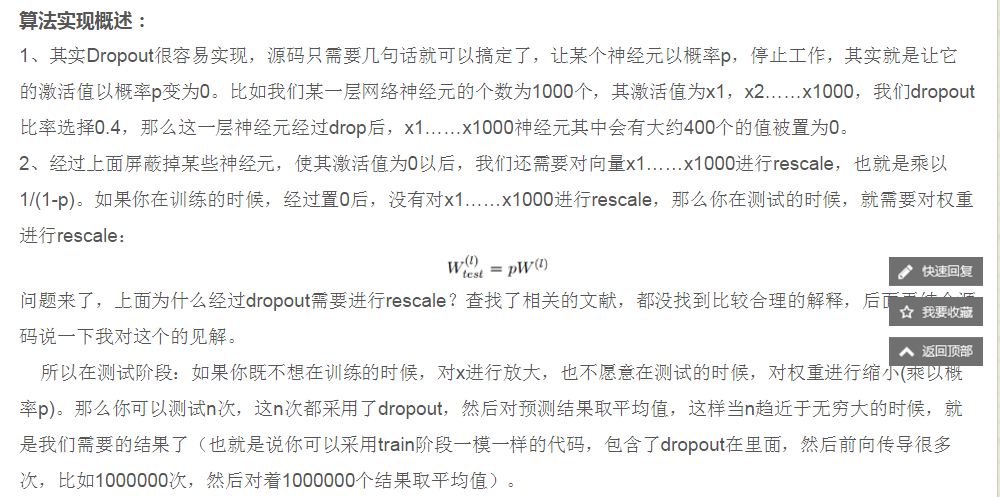
其中一个要选择的参数是keep\_prob，它代表每一层上保留单元的概率，所以不同层的keep\_prob也可以变化，很明显，W2维度为 7\*7是最大的权重矩阵，因为W2拥有最大参数集，即7\*7，为了预防矩阵的过拟合，对于这一层它的keep\_prob值应该会相对较低，对于其它层，过拟合的程度可能没有那么严重，keep\_prob可能会高一些，如果在某一层我们不必担心其过拟合的问题，那么keep\_prob可以为1，意味着保留所有单元并且不在这一层使用dropout。对于有可能出现过拟合且含有诸多参数的层，我们可以把keep\_prob设置成比较小的值以便应用更强大的dropout，有点像在处理L2正则化的正则化参数，我们尝试对某些层施行更多正则化。从技术上讲，我们也可以对输入层应用dropout，有机会删除一个或多个输入特征，虽然现实中通常不这样做，keep\_prob的值为1是非常常用的输入值，但是消除一半的输入特征是不太可能的。

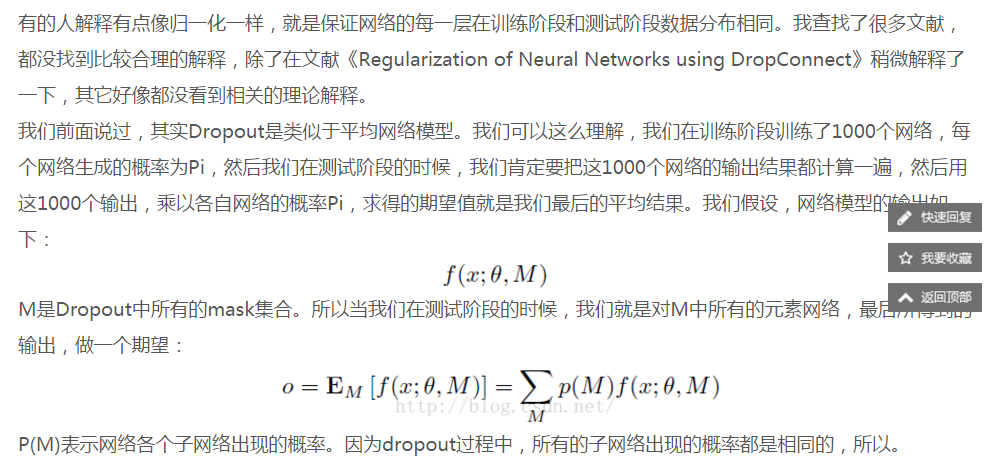
总结一下，如果你担心某些层比其它层更容易发生过拟合，可以把这些层的keep\_prob设置的低一些，缺点就是为了使用交叉验证，你要搜索更多的超级参数，另一个方案是在一些层上应用dropout，有些层不用，应用dropout的层只有一个超级参数，就是keep\_prob。

还有两个实施过程中的技巧，实施dropout在计算机视觉领域有很多成功的第一次，计算视觉中的输入量非常大，输入了太多像素，以致于没有足够的数据，所以dropout在计算机视觉中应用的比较频繁。牢记dropout是一种正则化方法，有助于预防算法过拟合，因此除非算法过拟合，不然是不会用dropout的，所以在其他领域用的比较少，经常用在计算机视觉，因为我们没有足够多的数据，所以一直存在过拟合，dropout的一大缺点就是代价函数不再被明确定义，每次迭代，都会随机移除一些节点，如果再三检查梯度下降的性能，实际上是很难进行复查的，定义明确的代价函数J每次迭代后都会下降，而dropout中我们所优化的代价函数J实际上并没有明确定义，或者在某种程度上很难计算，所以我们失去了调试工具来绘制这样一张图，即代价递减的图像，这个时候，一般先关闭dropout函数，将keep\_prob设为1，运行代码确保J函数单调递减，然后再打开dropout函数，在dropout过程中，代码并未引入bug。

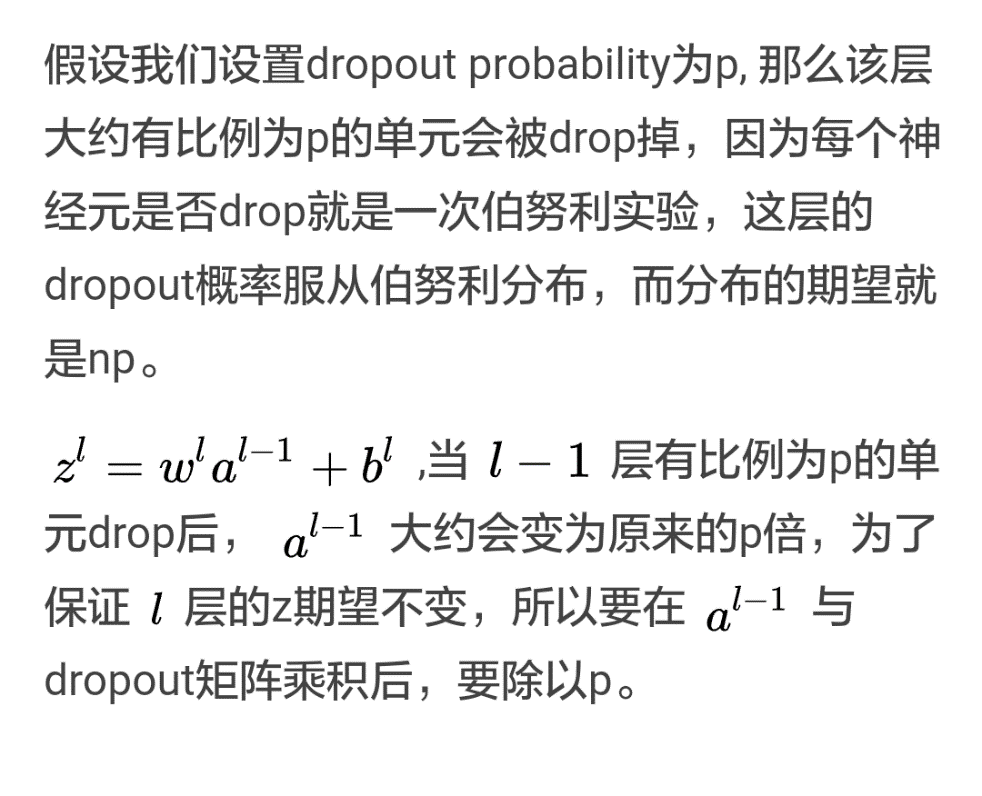
自己扩展的相关资料：

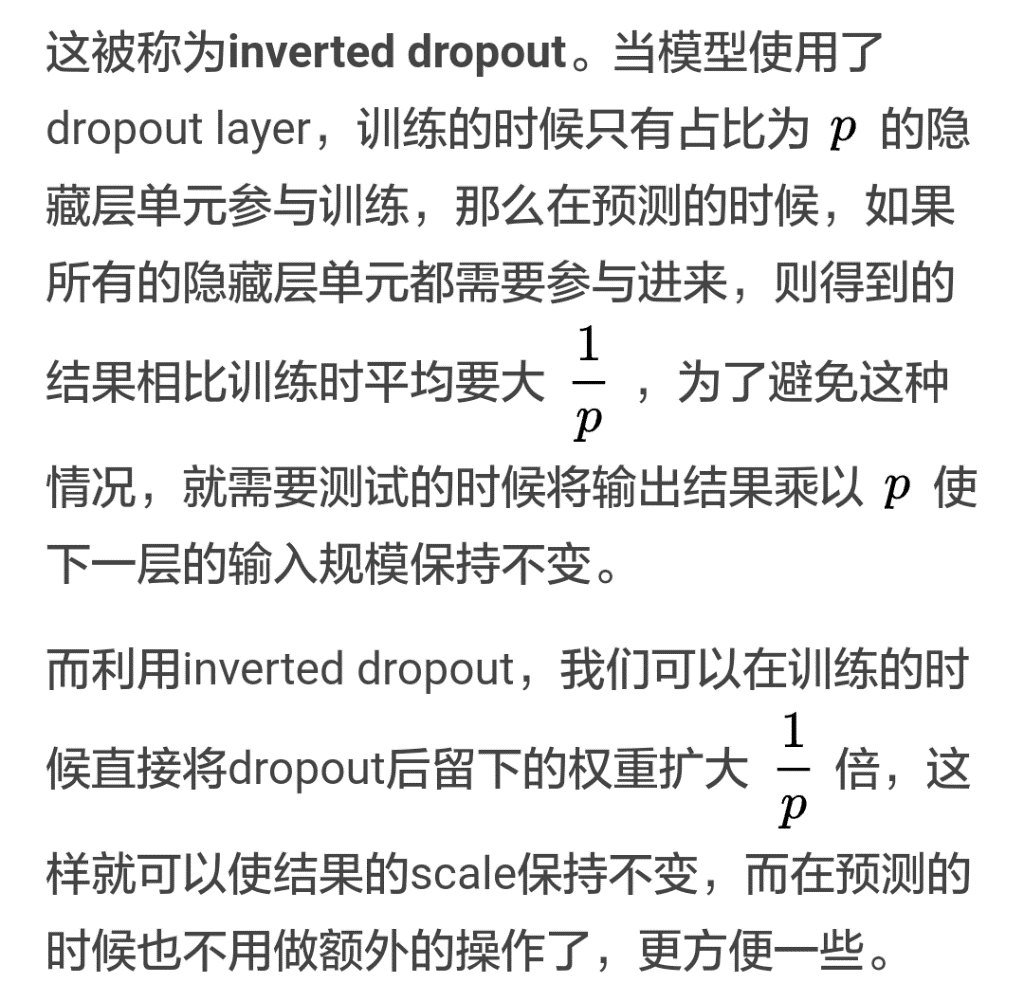






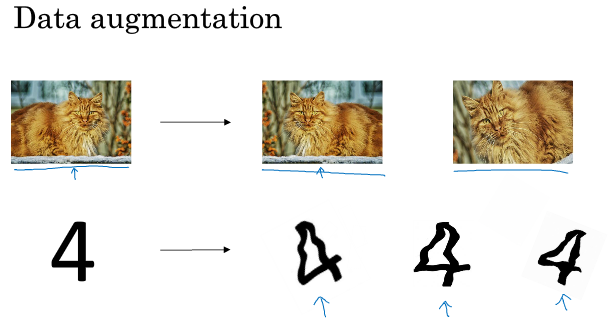
关于dropout过程，为什么dropout之后要进行rescale，从知乎找到两个解释：



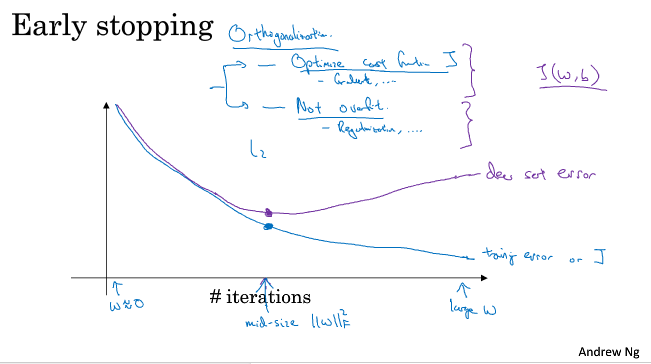


* 1. 其他正则化方法

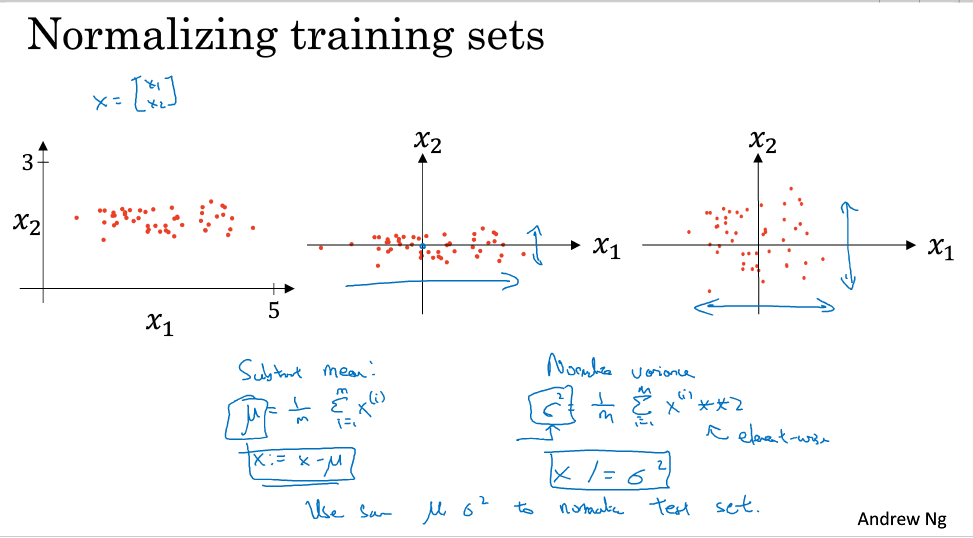
除了L2正则化和dropout正则化，还有几种方法可以减少神经网络中的过拟合，假设你正在拟合猫咪图片分类器，如果你想通过增加训练数据来解决过拟合，但扩增数据代价高，并且有时候也无法得到更多的数据，但我们可以通过添加这类图片来增加训练集，例如，水平翻转图片并把它添加到训练集，所以训练集中现在有原图和翻转后的这张图片，所以，通过水平翻转图片，训练集可以增大一倍，因为训练集有冗余，这虽然不如我们额外收集一组新图片那么好，但是这样做节省了获取更多猫咪图片的花费。除了水平翻转，也可以任意裁剪，可以增大数据集，额外生成假训练数据，和全新的，独立的猫咪图片数据相比，这些额外的数据无法提供那么多的数据，但是成本代价几乎为0，除了一些对抗性代价，以这种方式扩增算法数据，进而正则化数据集，减少过拟合比较廉价。像这样人工合成数据的话，我们要通过算法验证，告诉算法翻转后的图形仍然是猫。对于光学字符识别，我们还可以通过添加数字，随意旋转或扭曲数字来扩增数据，把这些数字添加到训练集。所以数据扩增可以作为正则化方法使用。



还有另外一种常用的方法叫做early stopping，提前终止（当验证集上的效果变差的时候），运行梯度下降时，可以绘制训练误差或只绘制代价函数J的优化过程，在训练集上用0-1记录分类误差次数，成单调递减趋势，因为在训练过程中，我们希望训练误差，代价函数J都在下降，通过early stopping我们不单可以绘制上面这些内容，还可以绘制验证集误差（dev set error），它可以是验证集上的分类误差，或验证集上的代价函数，logistic损失和log损失等，你会发现，验证集误差通常会呈先下降趋势，然后在某个节点上升，意思就是说，early stopping的作用是，你可能会说神经网络已经在这个迭代过程中表现的很好了，我们就此停止训练吧，得到验证集误差。是如何发挥作用的呢，当你还未在神经网络上运行太多迭代过程的时候，参数w接近0，因为随机初始化w时，它的值可能都是比较小的随机值，所以在你长期训练神经网络之前，w依然很小，在迭代和训练过程中，w的值会变得越来越大，所以提前停止要做的就是在中间点停止迭代过程，得到一个w值中等大小mid-size的弗罗贝尼乌斯范数。与L2正则化相似，选择参数w范数较小的神经网络，希望你的神经网络过度拟合不严重。经常会用到提前停止，但是它也有一个缺点。我认为机器学习过程包括几个步骤，其中一步是选择一个算法来优化代价函数J，如梯度下降，后面还会介绍其他，比如Momentum，RMSprop和Adam等。但是优化代价函数J之后，也不想发生过拟合，也有工具可以解决该问题，比如正则化，扩增数据等等。在机器学习中，超级参数激增，选出可行的算法也变得越来越复杂，我发现，如果我们用一组工具优化代价函数J，机器学习就会变得更简单，在重点优化代价函数J时，你只需要留意w和b，J(w,b)的值越小越好，你只需要想办法减小这个值，其他的都不用关注。然后预防过拟合还有其他任务，这一步我们用另外一套工具来实现，这个原理有时候被称为“正交化”Orthogonalization，思路就是在一个时间做一个任务。所以提前终止的重要缺点就是不能独立的处理这两个问题，因为提前停止梯度下降，也就是停止了优化代价函数J,所以代价函数的值不够小，同时又不希望出现过拟合，没有采取不同的方式来解决这两个问题，而是用一种方法同时解决两个问题。这样做的结果就是要考虑的东西变得更复杂。如果不用early stopping，另一种方法就是L2正则化，训练神经网络的时间就可能很长，发现这导致超级参数搜索空间更容易分解，也更容易搜索，但缺点就是你必须不停的尝试很多的lambad，这也导致搜索大量lambad值的计算代价太高。Early stopping的优点是只运行一次梯度下降，你就可以找出较小的W值，中间值或较大值，而无需尝试L2正则化超级参数很多值。



* 1. 正则化输入



训练神经网络，其中一个加速训练的方法就是归一化输入，假设我们有一个训练集，它有两个输入特征，所以输入特征x是二维的，下图是数据集的散点图，归一化输入需要两个步骤，第一步是零均值化，

是一个向量

意思是移动训练集，直到它完成零均值化，也就是让数据集的特征值在0左右，让数据集的平均值为0。

第二步是归一化方差，注意，在上面的例子中，可以看出来，特征x1的方差比特征x2的方差要大得多，方差数学中学过就是分布的均匀程度，我们要做的就是给赋值，

是一个向量，它的每个特征都有方差，注意我们已经完成零均值化， 就是方差，我们把所有数据都除以向量，图片最后就变成第三幅图。x1和x2的方差都等于1，提示一下如果你用它来调整训练数据，那么用相同的和来归一化测试集。尤其是你不希望训练集和测试集的归一化有所不同。不能在训练集和测试集上分别预估和，应该相同的这两个量定义的相同数据转换，这两个量是由训练集的数据计算出来的。

为什么要归一化输入特征：

回想一下定义的代价函数，如果使用非归一化的输入特征，那么代价函数有可能如图1，是一个非常细长狭窄的代价函数，你要找的最小值应该在红色箭头处，但如果特征值在不同范围，假如x1取值范围从1到1000，特征x2的取值范围是从0到1，结果是参数w1和w2值的范围和比率将会非常不同。如果你把该函数的部分轮廓画出来，也就是画在二维坐标轴上的函数轮廓，那么会是一个非常狭长的函数。然而如果归一化特征，代价函数平均起来看起来更对称，一个比较均匀的碗状，对应的轮廓图如图4。如果你在作图这样的代价函数上运行梯度下降法，找到代价最小的也就是碗底，那么就必须使用一个非常小的学习率，使每次迭代学习的步长小一点，如图中蓝色线条，如果在外圈位置，梯度下降法可能需要多次迭代过程，直到找到最小值，但如果函数是一个更圆的球形轮廓的话，那么不论从哪个位置开始，梯度下降法都能够更直接的找到最小值，也就是说，可以在梯度下降法中使用较大步长，不需要像左图那样找最小值步长小，并且反复执行。

![关于为什么要归一化特征](<http://upload-images.jianshu.io/upload_images/5355764-1ba50b9b3a0ff470.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

当然，实际上w是一个高维向量，我们只用了一个w和b只是为了方便说明，因此用二维绘制w并能正确地传达直观理解，但总的直观理解就是代价函数更均匀更圆一些，而且更容易优化，前提是体征都在相似范围内。不是很大范围内波动，而是在-1到1的范围内或有相似的偏差，这使得代价函数J优化起来更简单更快速。相似范围的意思比如说，一个特征1到2，一个0到1等，范围就很相似，效果就比较好，当它们在非常不同的取值范围内时，这对优化算法非常不利。但是仅将它们设置成为均化零值，假设方差为1，确保所有特征都在相似范围内，通常可以帮助学习算法运行的更快。

所以如果输入特征处于不同范围内，归一化特征值非常重要了。同理，如果特征值处于相似范围内，那么归一化就没那么重要。执行归一化并不会产生什么危害，所以经常都会做归一化处理尽管不确定它能否提高训练或算法速度。

* 1. 梯度消失和梯度爆炸

训练神经网络尤其是深度神经网络所面临的一个问题就是梯度消失和梯度爆炸（data vanishing/exploding gradients）。也就是说，当你训练深度网络时，导数和坡度有时会变得非常大，或非常小，甚至以指数方式变小，这增加了训练的难度。这节内容将明白梯度消失和梯度爆炸的真正含义，以及如何更明智地选择随机初始化权重从而避免这个问题。

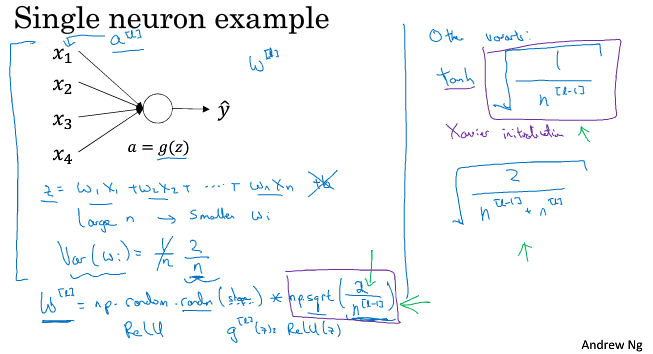
假设你在训练如图这样的一个极深的神经网络，有参数,,…，假设我们使用激活函数g(z)=z，也就是线性激活函数，忽略b，假设，如果这样的话，，可以发现最后两项乘就是，最后三项乘就是，所有这些矩阵数据传递的给出值，而不是y的值。假设每个权重矩阵（假设比1大一点）。从技术上讲，最后一项有不同维度，可能它就是余下的权重矩阵。忽略这个不同维度的权重矩阵，，因为我们假设所有的矩阵都等于它，它是1.5倍的单位矩阵，最后计算出来结果就是，忽略。对于一个神经网络来说，L值较大，那么也会非常大，实际上它呈指数级增长的，增长的比率是。因此对于一个深度神经网络，y的值将呈爆炸式的增长。相反地，如果权重是0.5，它比1小，增长的比率是，，再次忽略。因此每个矩阵都小于1，假设此时x1和x2都为1，那么根据深度神经网络计算第一层节点都是0.5,0.5，第二列就是0.25,0.25，依次类推，知道最后一项变成。所以作为自定义函数，激活函数的值将会以指数级下降，它是与网络层数量L相关的函数，在神经网络中，激活函数以指数级递减。

![激活函数爆炸和消失，同理梯度爆炸与消失](<http://upload-images.jianshu.io/upload_images/5355764-a652a8e4431d0865.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

所以理解应该是，权重矩阵比1大，或者说只比单位矩阵大一点，深度神经网络的激活函数将爆炸式增长，如果w比1略小一点，激活函数将以指数级递减。虽然我们只论述了激活函数关于L层会指数级增长或减小，但同样也适用于与层数L相关的导数和梯度函数，也是呈指数增长或递减。最近微软对152层的神经网络有了新的研究进展，在这样一个深度神经网络中，如果与L层相关呈指数增大和减小，将导致训练难度上升，尤其是梯度与L相差指数级，梯度下降算法的步长会非常非常小，梯度下降算法将花费很长时间来学习。虽然有一个不能彻底解决此问题的解决方案，但是已在如何选择初始化权重问题上提供了很大帮助。

* 1. 神经网络的权重初始化

针对神经网络产生的梯度消失和梯度爆炸问题，有一个不完整的解决方案，虽然不能彻底解决问题却很有用，有助于我们为神经网络更谨慎地选择随机初始化参数。为了更好地理解它，先举一个神经单元权重初始化的例子，然后再演变到整个深度神经网络。首先看看只有一个神经元的情况，然后才是深度网络。如下图，单个神经元可能有4个输入特征，从x1到x4，经过处理，最终得到，稍后将深度神经网络时，这些输入表示为，暂时用x表示。那么，假设b=0，先忽略b，为了预防z值过大或过小，可以看到n越大，你希望越小，因为z是的和，如果你把很多此类项相加，希望每项值更小。最合理的办法就是设置，n表示神经元的输入特征数量，实际上，你要做的就是设置某层权重矩阵后面部分表示该层每个神经元的特征数量分之一，是因为这是l层上拟合的单元数量。结果表明如果你用的是ReLU激活函数，方差设置为比的效果好。可以发现初始化时，尤其是使用ReLU激活函数时，，它取决于你对随机变量的熟悉程度，这是高斯随机变量，后面则是乘以它的平方根，也就是引用方差，这里用是因为在本例中逻辑回归的特征是不变的，就是输入了那么多个，但一般情况下，l层上的每个神经元都有个输入，如果激活函数的输入特征被零均值，标准方差化，方差为1，那么z也会调整到相似范围，这就没解决问题，但它确实降低了坡度消失和爆炸的问题，因为它给权重矩阵W设置了合理值，都知道不能比1大很多，也不能比1小很多，所以梯度没有爆炸或者消失太快。



提到了其他变体函数variants，刚刚提到的函数是ReLU激活函数，一篇由Herd等人撰写的论文曾介绍过，对于其他几个激活函数，如tanh函数，有篇论文提到，常量1比2效率更高，即对于tanh函数，应该是，即相当于，适用于tanh函数，这被称为Xavier初始化。

Yoshua Bengio和他的同事还提出另一种方法，使用的公式为，但如果你想用ReLU激活函数，也就是最常用的激活函数，那么就会用这个公式，，如果用tanh，就用。

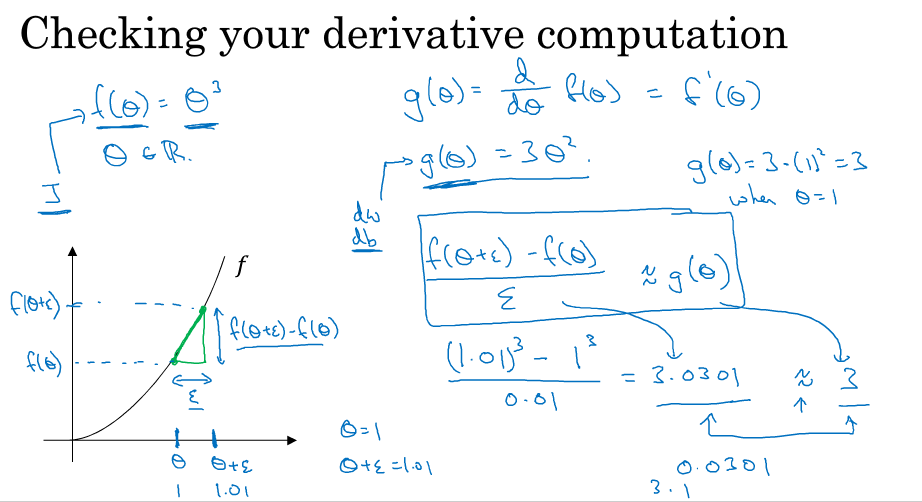
实际上，这些公式只是给你一些起点，它们给出初始化权重矩阵的方差的默认值，如果你想添加方差，方差参数则是另一个你需要调整的超级参数，可以给公式添加一个乘除参数，调优作为超级参数的一部分。有时候调优这个超级参数效果一般，这并不是想调优的首要超级参数，但一经发现条有过程中产生的问题，虽然调优该参数能起到一定的作用，但考虑到相比调优其他超级参数的重要性，通常把它优先级放到最低。

以上关于权重矩阵合理初始化也是加快训练速度的技巧。

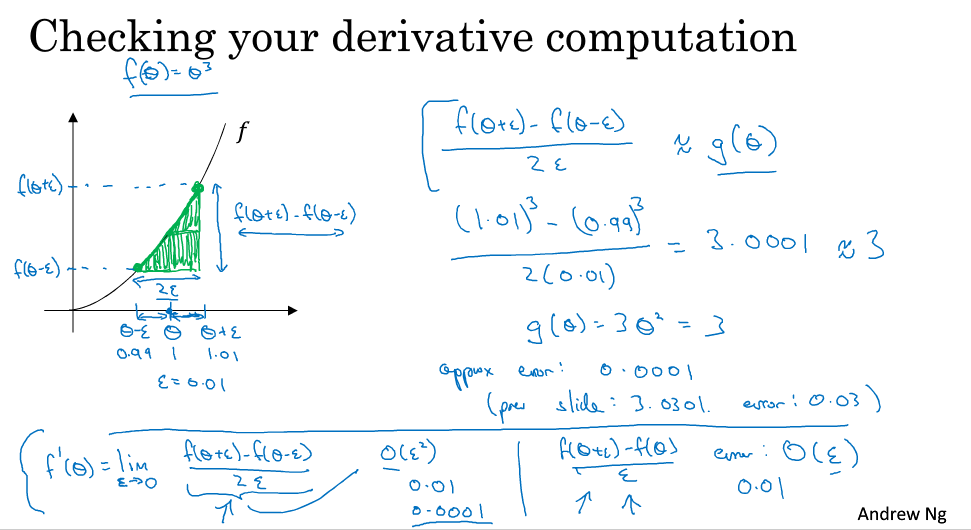
* 1. 梯度的数值逼近

在实施反向传播时，有一个测试叫做梯度检验，它的作用是确保反向传播正确实施，因为有时候你虽然写下了这些方程式，但是不能100%确定，执行backprop的细节都是正确的，为了逐渐实现梯度检验，首先说说如何对梯度做数值逼近。

首先画出函数f如下图，,假设，在的左右分别取一个很小值，以及，并且假设，可以根据小绿色三角形，计算高和宽的比值，就是更准确的坡度预估。如果用，以及两个点构成三角形，那么这个大三角形的高宽比值更接近于点的导数。其实都是关于微积分导数的知识。关于从到那么也有一个小三角形，通过绿色大三角形同时考虑了这两个小三角形。所以得到的不是一个单边公差，而是一个双边公差。其实这个意思就是关于点的导数，通过在左右取一个接近于0的范围，通过这个范围三角形的斜率作为该点的导数，即是高数中学习的导数定义，当接近于0时，求斜率的极限值，即为点的导数。通过具体的数字举例，也可以发现如果在的一边取范围，求得斜率作为导数与最后我们在点的导数求出来相差比在双边都取范围的斜率值作为导数相差大得多。因此，我们得到结论是，双边公差更能表示函数f在点的导数即梯度。



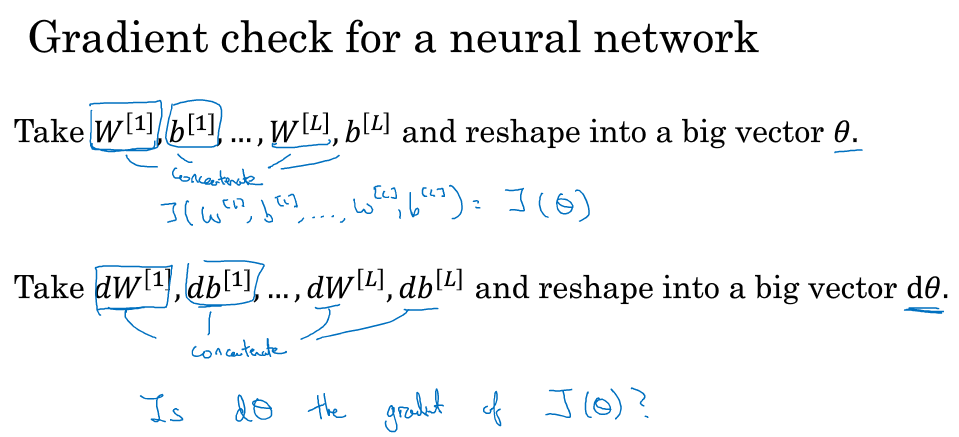
具体的解释如下：它的期望值接近,代入数字检验结果如下，误差为0.0001，而上一张图中，我们只考虑了单边误差，即从到之间的误差，计算出来斜率为3.0301，很明显误差更大，所以使用双边误差的方法更逼近导数。也更加确定可能是一个f导数的正确实现。在梯度检验和反向传播中使用该方法时，最终它与运行两次单边公差的速度一样，实际上，这种方法也还是很指的使用的，因为实际上它更准确。导数的官方定义是。对于一个非零的，的逼近误差可以写成,大写符号O的含义是指逼近误差其实是一些常量乘以，的确是很准确的逼近误差。然而，如果我们用另外一个公式，的逼近误差就是，当小于1时，实际上比大很多，所以近似值没有前者准确。所以在执行梯度检验时，我们应该使用双边误差，结果更准确。



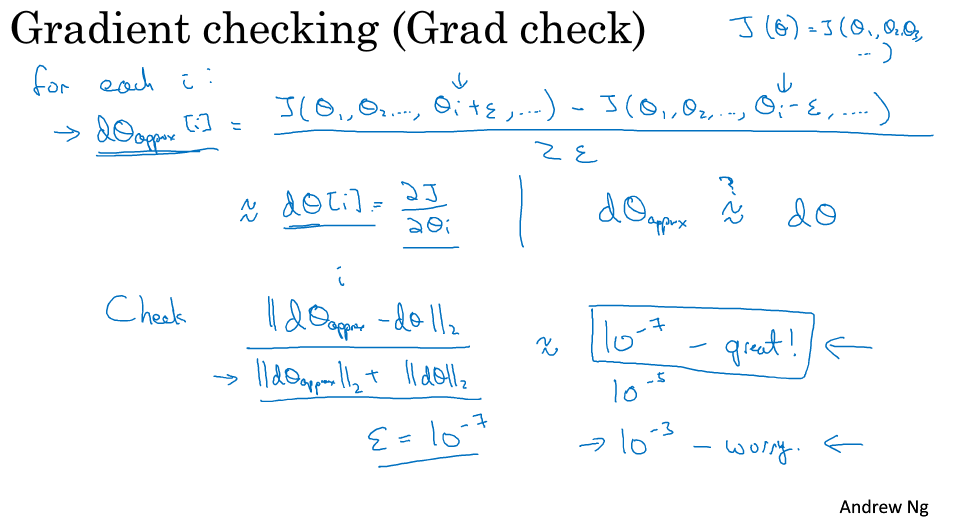
所以这一节讲了如何使用双边误差，来判断别人给你的函数是否正确的实现了函数f的偏导，现在可以使用这个方法来检验反向传播是否得以正确实施。如果不正确，可能有bug需要你解决。

* 1. 梯度检验

梯度检验可以帮忙节省很多时间以及发现backprop过程中的bug，接下来看看如何利用梯度检验来检验backprop的实施是否是正确的。



假设你的网络中有每一层的W和b参数，为了执行梯度检验，首先要做的就是把所有参数转换成一个巨大的向量数据。即把所有矩阵W转换成一个向量，依次做连接运算，得到一个巨型向量，代价函数J原本是所有W和b的函数，现在得到了一个关于的代价函数。

接着得到与W和b顺序相同的数据， ,,,用它们来初始化向量,它有与相同的维度。同样的把转成成向量，已经是一个向量了，直到把转换成向量，这样所有的都已经是向量，注意与具有相同维度，与具有相同维度，经过相同的转换和连接运算操作之后，可以把所有导数转换成一个大向量，它与具有相同维度，那么现在的问题是和代价函数J的梯度有什么关系。接下来就是实施梯度检验grad check的过程。

首先我们要清楚，J是超级参数的一个函数，也可以将J函数展开为，不论超级参数向量的维度是多少，为了实施梯度检验，要做的就是循环执行下面步骤：

For each i：（i也就是表示每个的组成元素）

其余都不变，只是在上进行调整，其余项都不变，使用的是双边误差。从上一节内容我们知道，这个值是个近似值，并且应该逼近，即J代价函数对这个元素的偏导数。所以要做的就是对每个i的每个值都执行这个运算，最后得到两个向量，的逼近值，它与有相同的维度。要验证的就是这些向量是否彼此阶级。

具体来说，如何定义两个向量是否真的彼此接近，做下列运算，计算这两个向量的距离，欧几里得范数，注意这里没有平方，这个式子表示误差平方之和，然后求平方根，得到欧氏距离。然后用向量长度做归一化，结果为使用长度向量的欧几里得范数，坟墓只是为了预防这些向量太大或者太小，分母使这个方程式变成比率，实际上我们执行这个式子，值可能为，使用这个范围内的，如果你发现计算这个得到的值比小就很好，这意味着导数逼近很有可能是正确的，它的值非常小。如果是在范围内，就要小心，也许这个值没问题，但会再次检查这个向量的所有项，确保没有一项的误差过大。如果有一项误差非常大， 可能有bug，如果计算出来结果是，就会担心是否有bug，就应该仔细检查所有项，看是否有一个具体的i值，使得与大不相同，并用它来追踪一些求导计算是否正确。经过调试最终结果会是这种非常小的值。那么你的实施可能是正确的。在实施神经网络时，经常需要执行foreprop和backprop，可能发现这个梯度检验有一个相对较大的值，就会怀疑有bug，然后就不断调试，最后得到一个很小的梯度检验值。

<http://blog.csdn.net/u012526120/article/details/48973497>

（csdn关于梯度检验的博客）

* 1. 关于梯度检验实施的注意事项和实施技巧

首先，第一点，不要在训练中使用梯度检验，它只适用于调试，意思是计算所有i值的是一个非常漫长的过程。为了实施梯度下降，必须使用backprop来计算以及计算导数，只有调试的时候，你才会就算它，来确认数值是否是接近，完成后，你会关闭梯度检验，梯度检验的每一个迭代过程都不执行这个过程，因为它太慢了。第二点，如果算法的梯度检验失败，要检查所有项，试着找出bug，意思就是如果与很大，就要查找不同的i值，看看是哪个导致的。举个例子，如果你发现相对于某些层或某层的或者以及b相差很大，但是得各项非常接近，注意的各项b和W的各项都是一一对应的，会发现在计算参数b的导数db的过程中存在bug。同理，如果是w相差很大，那么可以帮你定位bug的位置。虽然未必能准确定位，但是可以估测在哪些地方。第三点，在实施梯度检验时，如果使用正则化，注意正则项，如果代价函数

等于与相关的J函数的梯度，注意要包括正则项进去。第四点，梯度检验不能和dropout同时使用，因为在每次迭代过程中，dropout会随机消除隐层单元的不同子集，难以计算dropout在梯度下降上的代价函数J，因此dropout可以作为优化代价函数J的一种方法，但是代价函数J被定义为对所有指数极大的节点子集求和，而在任何迭代过程中，这些节点有可能被消除，所以很难计算J，你只是对成本函数做了抽样。所以很难用梯度检验来双重检验dropout的计算，所以一般不同时使用，你可以把keep\_prob设置为1，然后打开dropout，寄希望于dropout的实施是正确的。你还可以做点别的，比如修改节点丢失的模式确保梯度检验是正确的，但一般不用做，所以一般关闭dropout。最后一点，现实中几乎不出现这种情况，当W和b接近0时，梯度下降的实施是正确的。在随机初始化过程中，在运行梯度下降时，W和b变得更大，可能只有在W和b接近0时，backprop的实施才是正确的。但是当W和b变大时，它会变得越来越不准确。可以做但一般不做的就是在随机初始化过程中运行梯度检验，然后再训练网络，如果随机初始值比较小W和b会有一段时间远离0，反复训练网络后，再重新运行梯度检验。

Coursera无法观看课程解决方案

<https://jingyan.baidu.com/article/6f2f55a14059eeb5b93e6cab.html>

神经网络防止过拟合的方法：

1. Data augmentation扩增数据量
2. Weight decay权重缩减：即L1 L2正则化等
3. Early stop提前终止：训练集上错误率一般会一直降低，而验证集上一般先降后升，在验证集错误率最小时候终止迭代
4. Dropout 随机失活

<https://www.cnblogs.com/Allen-rg/p/7675914.html>

针对高偏差、高方差问题的解决方法：

1. 解决高方差（即过拟合）：增大训练样本量、缩小特征量、增大lambda值
2. 解决高偏差（即欠拟合）：增大特征量，增加多项式特征（即增大神经网络的节点数以及深度等）、减小lambda值

关于对lambda参数的理解，这个参数是指正则化参数，如果整个参数越大，调整后的W参数就越接近于0，那么可以理解为神经网络中节点影响力更小，神经网络趋于简单化，那么拟合效果就不会那么好，即解决过拟合。

隐藏层数的选择对于拟合效果的影响：

1. 隐藏层数过少，神经网络简单，参数少，容易出现欠拟合

2. 隐藏层数过多，神经网络复杂，参数多，容易出现过拟合，同时计算量也庞大。

