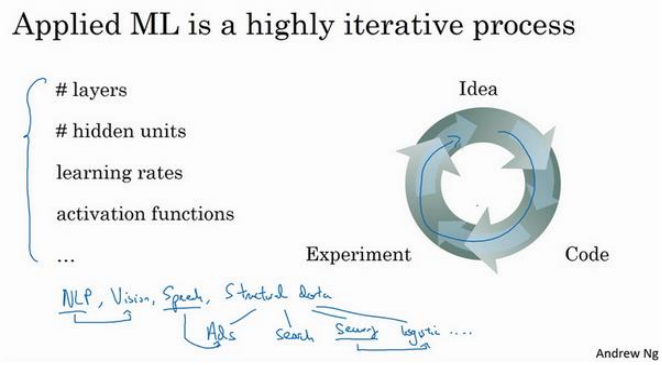
目录

第一周：深度学习的实践层面

1.1 训练，验证，测试集

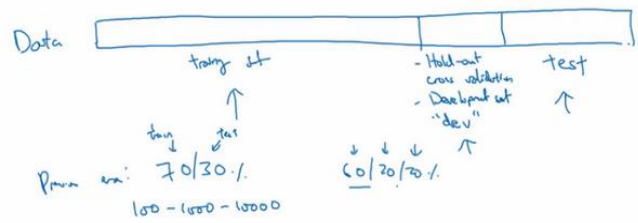
本周我们将继续学习如何有效运作神经网络，内容涉及超参数调优，如何构建数据，以及如何确保优化算法快速运行，从而使学习算法在合理时间内完成自我学习。

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地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；每层含有多少个隐藏单元；学习速率是多少；各层采用哪些激活函数。



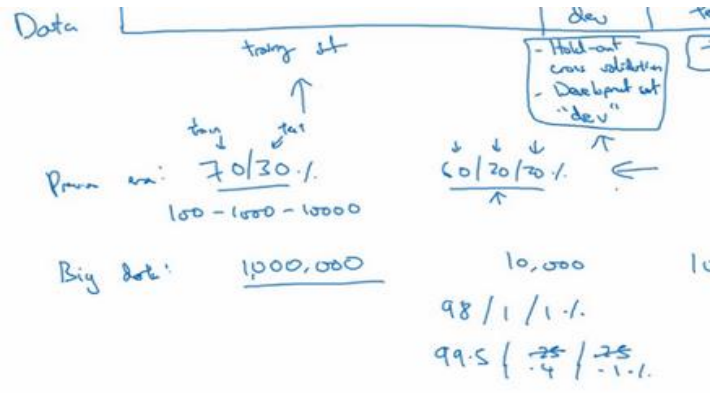
创建新应用的过程中，我们不可能从一开始就准确预测出这些信息和其他超级参数。实际上，应用型机器学习是一个高度迭代的过程。通常在项目启动时，我们会先有一个初步想法，比如构建一个含有特定层数，隐藏单元数量或数据集个数等等的神经网络，然后编码，并尝试运行这些代码，通过运行和测试得到该神经网络或这些配置信息的运行结果，你可能会根据输出结果重新完善自己的想法，改变策略，或者为了找到更好的神经网络不断迭代更新自己的方案。应用深度学习是一个典型的迭代过程，需要多次循环往复，才能为应用程序找到一个称心的神经网络，因此循环该过程的效率是决定项目进展速度的一个关键因素，而创建高质量的训练数据集，验证集和测试集也有助于提高循环效率。

假设这是训练数据，用一个长方形表示，通常这些数据将划分成三部分：一部分训练集，一部分简单交叉验证集，有时也称之为验证集，最后一部分则作为测试集。



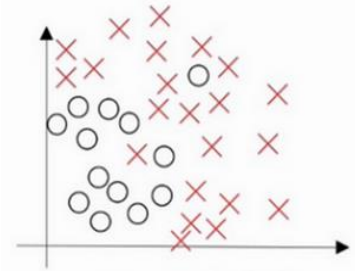
在机器学习发展的**小数据量时代（100 条，1000 条或者 1 万条数据）**，常见做法是将所有数据三七分，就是人们常说的 **70%验证集，30%测试集**，如果没有明确设置验证集，也可以按照 **60%训练，20%验证和 20%测试集**来划分。这是前几年机器学习领域普遍认可的最好的实践方法。

但在**大数据时代**，数据量可能是百万级别，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。验证集的目的就是验证不同的算法，检验哪种算法更有效，比如 2 个甚至 10 个不同算法，判断出哪种算法更有效。我们可能不需要拿出 20%的数据作为验证集。比如我们有 **100 万条数据，取 1 万条数据便足以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，测试集的主要目的是正确评估分类器的性能，所以，**假设有 100 万条数据，其中 1 万条作为验证集，1 万条作为测试集，即：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%，或者验证集占 0.4%，测试集占 0.1%**。



1.2 偏差，方差

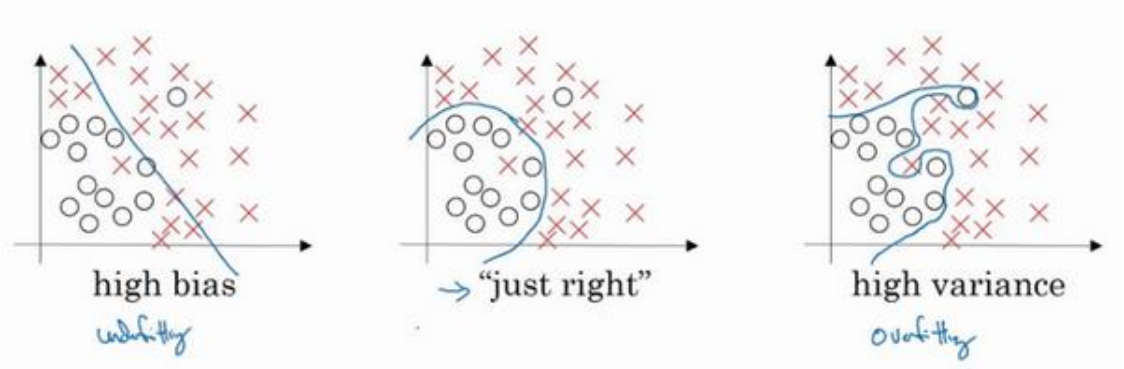
几乎所有机器学习从业人员都期望深刻理解偏差和方差，这两个概念易学难精，即使你自己认为已经理解了偏差和方差的基本概念，却总有一些意想不到的新东西出现。关于深度学习的误差问题，你可能听说过这两个概念，但深度学习的误差很少权衡二者，我们总是分别考虑偏差和方差，却很少谈及偏差和方差的权衡问题，下面我们来一探究竟。



假设这是数据集，如果给这个数据集拟合一条直线，可能得到一个逻辑回归拟合，但它并不能很好地拟合该数据，这是高偏差（**high bias**）的情况，称为“欠拟合”（**underfitting**）。

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含有隐藏单元的神经网络，可能就非常适用于这个数据集，但是这看起来也不是一种很好的拟合方式分类器方差较高（**high variance**），数据过度拟合（**overfitting**）。

在两者之间，可能还有一些像图中这样的，复杂程度适中，数据拟合适度的分类器，这个数据拟合更加合理，称之为“适度拟合”（**just right**）是介于过度拟合和欠拟合中间。



在只有x_1和x_2两个特征的二维数据集中，我们可以绘制数据，将偏差和方差可视化。在多维空间数据中，无法绘制数据和可视化分割边界，但可以通过几个指标来研究偏差和方差。

我们沿用猫咪图片分类这个例子，左边一张是猫咪图片，右边一张不是。理解偏差和方差的两个关键数据是训练集误差（**Train set error**）和验证集误差（**Dev set error**），为了方便论证，假设我们可以辨别图片中的小猫，用肉眼识别几乎是不会出错的（错误了 0%）。

假定训练集误差 1%，验证集误差 11%，可以看出训练集拟合非常好，而验证集相对较差，我们可能过度拟合了训练集，在某种程度上，验证集并没有充分利用交叉验证集的作用，**这种情况称之为“高方差”**。通过查看训练集误差和验证集误差，我们便可以诊断算法是否具有高方差，也就是说衡量训练集和验证集误差就可以得出不同结论。

假设训练集误差 15%，验证集误差 16%，算法并没有在训练集中得到很好训练，如果训练数据的拟合度不高，数据欠拟合，**就可以说这种算法“高偏差”**。相反它对于验证集产生的结果是合理的，验证集中的错误率只比训练集的多了 1%，所以这种算法方差低，偏差高。

再举一个例子，训练集误差是 15%，偏差相当高，但是，验证集的评估结果更糟糕，错误率达到 30%，在这种情况下，我会认为这种算法偏差高，因为它在训练集上结果不理想，而且方差也很高，**这是方差偏差都很糟糕的情况**。

再看最后一个例子，训练集误差是 0.5%，验证集误差是 1%，用户看到这样的结果会很开心，猫咪分类器只有 1%的错误率，**偏差和方差都很低**。

注意：这些分析都是基于假设预测的，假设人眼辨别的错误率接近 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%。如果最优误差或贝叶斯误差非常高，比如 15%。再看看这个分类器（训练误差 15%，验证误差 16%），15%的错误率对训练集来说也是非常合理的，偏差不高，方差也非常低。

总结，我们讲了如何通过分析在训练集上训练算法产生的误差和验证集上验证算法产生的误差来诊断算法是否存在高偏差和高方差，是否两个值都高，或者两个值都不高，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。

1.3 机器学习基础

这是我在训练神经网络时用到的基本方法，初始模型训练完成后，我首先要知道算法的偏差高不高，如果偏差较高，试着评估训练集或训练数据的性能。如果偏差的确很高，甚至无法拟合训练集，那么你要做的就是选择一个新的网络，比如含有更多隐藏层或者隐藏单元的网络，或者花费更多时间来训练网络，或者尝试更先进的优化算法。

如果网络足够大，通常可以很好的拟合训练集，只要你能扩大网络规模，如果图片很模糊，算法可能无法拟合该图片，但如果有人可以分辨出图片，如果你觉得基本误差不是很高，那么训练一个更大的网络，你就应该可以很好地拟合训练集，至少可以拟合或者过拟合训练集。一旦偏差降低到可以接受的数值，检查一下方差有没有问题，为了评估方差，我们要查看验证集性能，我们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，**如果方差高（过拟合），最好的解决办法就是采用更多数据**，但有时候，我们无法获得更多数据，此时可以尝试通过**正则化来减少过拟合**。有两点需要大家注意：

第一点，高偏差和高方差是两种不同的情况，我通常会用训练验证集来诊断算法是否存在偏差或方差问题，然后根据结果选择方法。举个例子，**如果算法存在高偏差问题，准备更多训练数据其实也没什么用处，至少这不是更有效的方法**，所以大家要清楚存在的问题是偏差还是方差，还是两者都有问题，明确这一点有助于我们选择出最有效的方法。

第二点，在机器学习的初期阶段，关于所谓的偏差方差权衡的讨论屡见不鲜，，减少方差，也可以减少偏差，增加方差；在深度学习的早期阶段，我们没有太多工具可以做到只减少偏差或方差却不影响到另一方；但在当前的深度学习和大数据时代，只要持续训练一个更大的网络，只要准备了更多数据，那么也并非只有这两种情况，只要正则适度，通常构建一个更大的网络便可以，在不影响方差的同时减少偏差，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。

这两步需要做的工作是：训练网络，选择网络或者准备更多数据，现在我们有工具可以做到在减少偏差或方差的同时，不对另一方产生过多不良影响。这就是深度学习对监督式学习大有裨益的一个重要原因，也是不用太过关注如何平衡偏差和方差的一个重要原因。

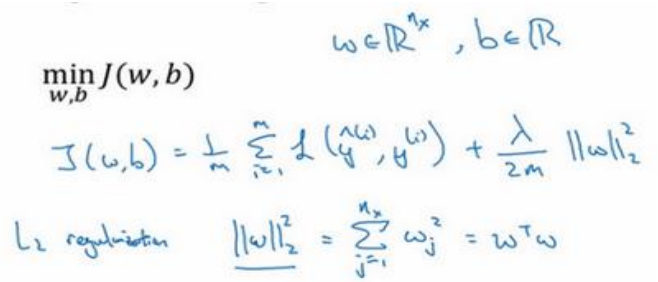
今天我们讲了如何通过组织机器学习来诊断偏差和方差的基本方法，然后选择解决问题的正确操作，希望大家有所了解和认识。我在课上不止一次提到了正则化，它是一种非常实用的减少方差的方法，正则化时会出现偏差方差权衡问题，偏差可能略有增加，如果网络足够大，增幅通常不会太高，我们下节课让大家更好理解如何实现神经网络的正则化。

1.4 正则化

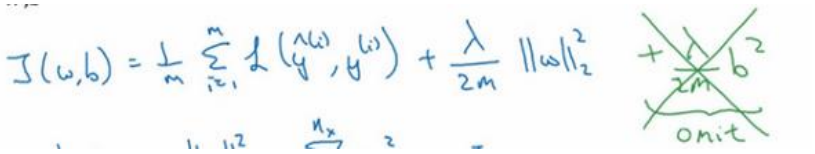
深度学习可能存在过拟合问题——高方差，有两个解决方法，一个是正则化，另一个是准备更多的数据，但可能无法时时刻刻准备足够多的训练数据或者获取更多数据的成本很高，但正则化通常有助于避免过拟合或减少你的网络误差。

下面来看看正则化的原理。首先看看逻辑回归，求成本函数J的最小值，它是我们定义的成本函数，参数包含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，只需添加参数 λ，也就是正则化参数。

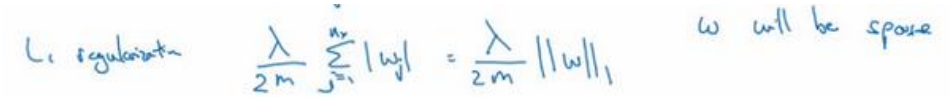
$\frac{\lambda}{2m}$乘以w范数的平方，w欧几里德范数的平方等于w_j（j 值从 1 到n_x）平方的和，也可表示为w^Tw，也就是向量参数w 的欧几里德范数（2 范数）的平方，被称为向量参数w的$L2$范数，此方法称为$L2$正则化。



为什么只正则化参数w？为什么不再加上参数 b 呢？你可以这么做，但w通常是一个高维参数矢量，可以表达高偏差问题，它包含有很多参数，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，其实也没太大影响，因为b只是众多参数中的一个，所以我通常省略不计，如果你想加上这个参数，完全没问题。

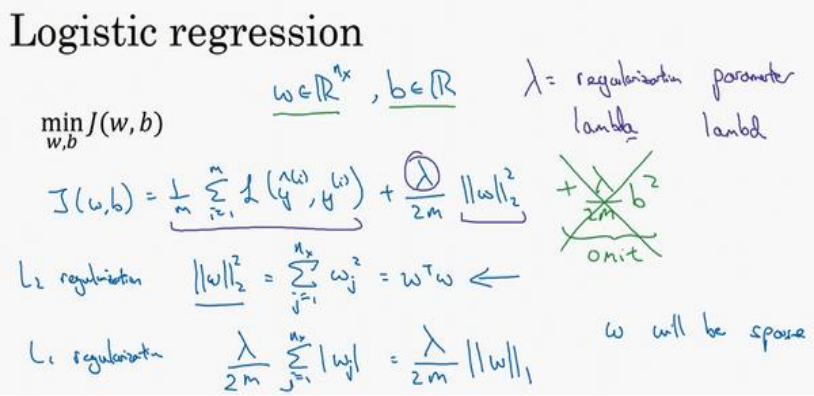


$L2$正则化是最常见的正则化类型，有人也可能听说过$L1$正则化，它是再成本函数后加上正则项$\frac{\lambda}{m}\sum_{j=1}^{n_x}|w|$，$\sum_{j=1}^{n_x}|w|$也被称为参数$w$向量的$L1$范数，无论分母是$m$还是$2m$，它都是一个比例常量。

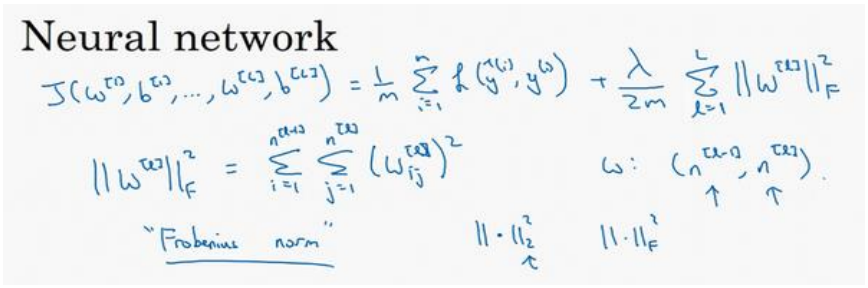


如果用的是$L1$正则化，w最终会是稀疏的，也就是说w向量中有很多 0，有人说这样有利于压缩模型，因为集合中参数均为 0，存储模型所占用的内存更少。实际上，虽然$L1$正则化使模型变得稀疏，却没有降低太多存储内存，所以我认为这并不是$L1$正则化的目的，至少不是为了压缩模型，人们在训练网络时，越来越倾向于使用$L2$正则化。

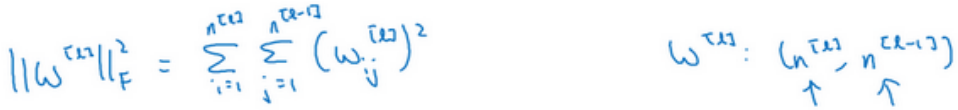
我们来看最后一个细节，λ是正则化参数，我们通常使用验证集或交叉验证集来配置这个参数，尝试各种各样的数据，寻找最好的参数，我们要考虑训练集之间的权衡，把参数设置为较小值，这样可以避免过拟合，所以 λ 是另外一个需要调整的超参数，顺便说一下，为了方便写代码，在 **Python** 编程语言中，λ是一个保留字段，编写代码时，我们删掉a，写成$lambd$，以免与 **Python** 中的保留字段冲突，这就是在逻辑回归函数中实现$L2$正则化的过程，如何在神经网络中实现$L2$正则化呢？



神经网络含有一个成本函数，该函数包含$W^{[1]}$，$b^{[1]}$到$W^{[l]}$，$b^{[l]}$所有参数，字母L是神经网络所含的层数，因此成本函数等于m个训练样本损失函数的总和乘以$\frac{1}{m}$，正则项为$\frac{\lambda}{2m}\sum_1^L |W^{[l]}|^2$，我们称$||W^{[l]}||^2$为范数平方，**这个矩阵范数$||W^{[l]}||^2$（即平方范数），被定义为矩阵中所有元素的平方求和**。



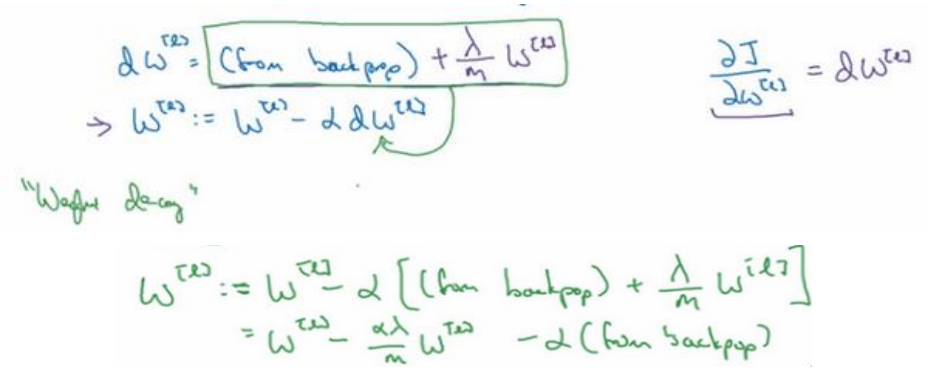
我们看下求和公式的具体参数，第一个求和符号其值 i 从 1 到$n^{[l-1]}$，第二个其 j 值从 1 到$n^{[l]}$，因为W是一个$n^{[l]} \times n^{[l-1]}$的多维矩阵，$n^{[l]}$表示 l 层单元的数量，$n^{[l-1]}$表示第$l-1$层隐藏单元的数量。



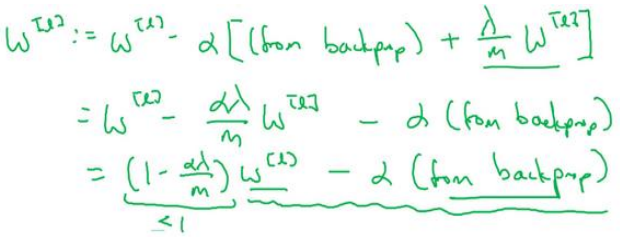
该矩阵范数被称作“弗罗贝尼乌斯范数”（**Frobenius norm**），用下标F标注，矩阵$L2$范数听起来更自然，但按照惯例，我们称之为“弗罗贝尼乌斯范数”，它表示一个矩阵中所有元素的平方和。

该如何使用该范数实现梯度下降呢？用 **backprop** 计算出dW的值，**backprop** 会给出 J 对 W的偏导数，实际上是把$W^{[l]}$替换为$W^{[l]}$减去学习率乘以dW。

现在要增加正则化项，我们要做的是给dW加上这一项$\frac{\lambda}{m}W^{[l]}$，然后计算这个更新项，使用新定义的$dW^{[l]}$，我们用$dW^{[l]}$的定义替换此处的$dW^{[l]}$，可以看到，$W^{[l]}$的定义被更新为$W^{[l]}$减去学习率a 乘以 **backprop** 再加上$\frac{\lambda}{m}W^{[l]}$。

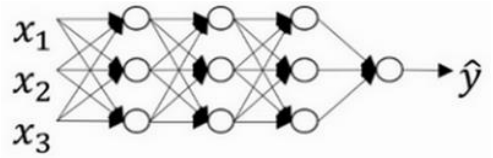


该正则项说明，不论$W^{[l]}$是什么，我们都试图让它变得更小，实际上，相当于我们给矩阵 W 乘以$(1-a\frac{\lambda}{m})$倍的权重，矩阵W减去$\alpha\frac{\lambda}{m}$倍的它，也就是用这个系数$(1-a\frac{\lambda}{m})$乘以矩阵W，该系数小于 1，因此$L2$范数正则化也被称为“权重衰减”，因为它就像一般的梯度下降，W被更新为少了a乘以 **backprop** 输出的最初梯度值，同时W也乘以了这个系数，这个系数小于 1，因此$L2$正则化也被称为“权重衰减”。



1.5 为什么正则化可以预防过拟合呢？

为什么正则化有利于预防过拟合呢？为什么它可以减少方差问题？我们通过两个例子来直观体会一下。

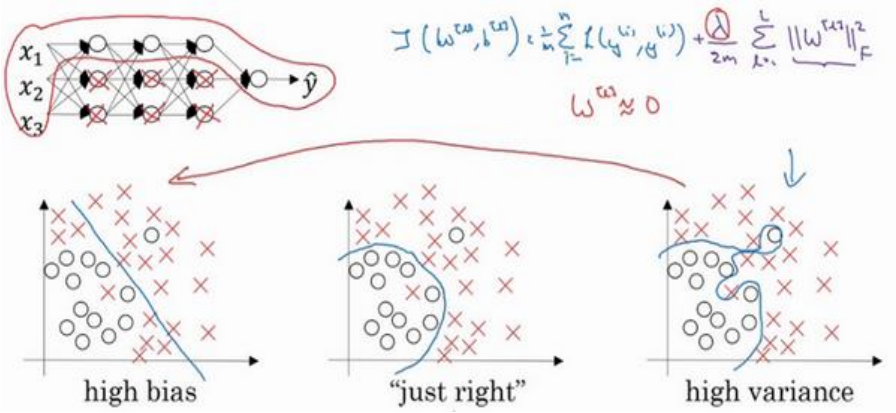


现在来看下这个庞大的深度拟合神经网络。你可以想象这是一个过拟合的神经网络。这是我们的代价函数J，含有参数W，b。我们添加正则项，它可以避免数据权值矩阵过大，这就是弗罗贝尼乌斯范数，为什么压缩$L2$范数，或者弗罗贝尼乌斯范数可以减少过拟合？

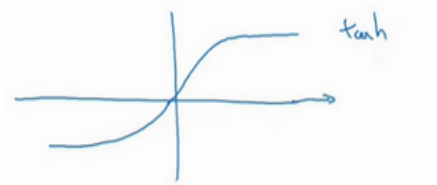
直观上理解就是如果正则化λ设置得足够大，权重矩阵W被设置为接近于 0 的值，直观理解就是把多隐藏单元的权重设为 0，于是基本上消除了这些隐藏单元的许多影响。如果是这种情况，这个被大大简化了的神经网络会变成一个很小的网络，小到如同一个逻辑回归单元，可是深度却很大，它会使这个网络从过度拟合的状态更接近高偏差状态。

但是λ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λ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更简单，这个神经网络越来越接近逻辑回归，我们直觉上认为大量隐藏单元被完全消除了，其实不然，实际上是该神经网络的所有隐藏单元依然存在，但是它们的影响变得更小了。神经网络变得更简单了，貌似这样更不容易发生过拟合。

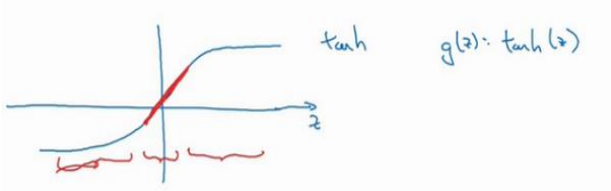


我们再来直观感受一下，正则化为什么可以预防过拟合，假设我们用的是这样的双曲线激活函数。

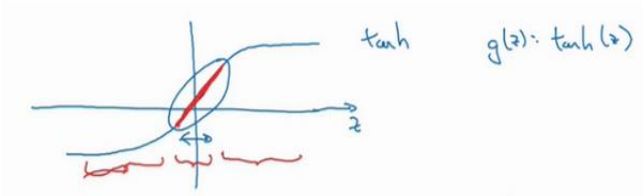


用$g(z)$表示$tanh(z)$，那么我们发现只要 z 非常小，如果 z 只涉及少量参数，这里我们利

用了双曲正切函数的线性状态，只要 z 可以扩展为这样的更大值或者更小值，激活函数开始变得非线性。



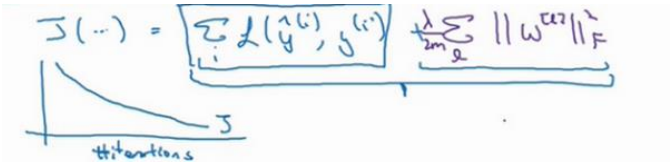
特别是，如果z的值最终在这个范围内，都是相对较小的值，$g(z)$大致呈线性，每层几乎都是线性的，和线性回归函数一样。



第一节课我们讲过，如果每层都是线性的，那么整个网络就是一个线性网络，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，最终我们只能计算线性函数，因此，它不适用于非常复杂的决策，以及过度拟合数据集的非线性决策边界，不容易过拟合。

总结一下，如果正则化参数变得很大，参数W很小，z也会相对变小，此时忽略b的影响，z会相对变小，实际上，z的取值范围很小，这个激活函数，也就是曲线函数$tanh$会相对呈线性，整个神经网络会计算离线性函数近的值，这个线性函数非常简单，并不是一个极复杂的高度非线性函数，不会发生过拟合。

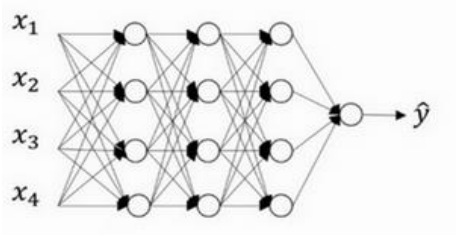
大家在编程作业里实现正则化的时候，会亲眼看到这些结果，总结正则化之前，我给大家一个执行方面的小建议，在增加正则化项时，应用之前定义的代价函数J，我们做过修改，增加了一项，目的是预防权重过大。



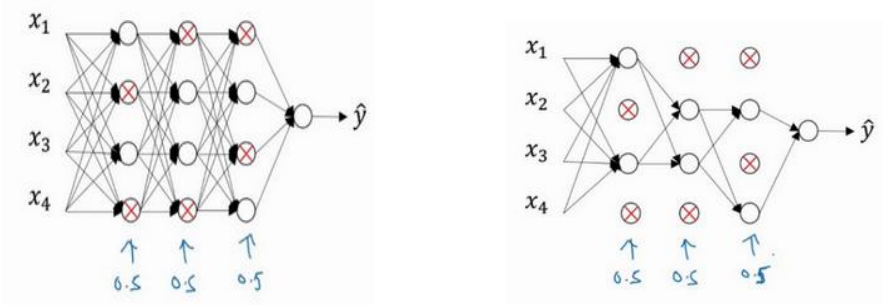
这就是$L2$正则化，它是我在训练深度学习模型时最常用的一种方法。在深度学习中，还有一种方法也用到了正则化，就是 **dropout** 正则化，我们下节课再讲。

1.6 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）”，我们来看看它的工作原理。



假设你在训练上图这样的神经网络，它存在过拟合，这就是 **dropout** 所要处理的，我们复制这个神经网络，**dropout** 会遍历网络的每一层，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的方式设置概率，每个节点得以保留和消除的概率都是 0.5，设置完节点概率，我们会消除一些节点，然后删除掉从该节点进出的连线，最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。



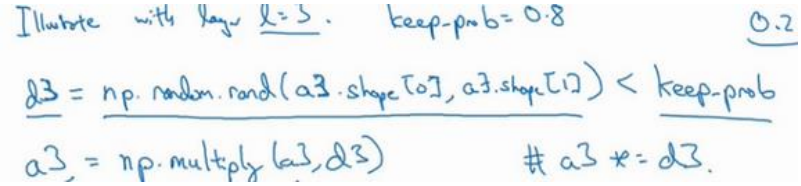
这是网络节点精简后的一个样本，对于其它样本，我们照旧以抛硬币的方式设置概率，保留一类节点集合，删除其它类型的节点集合。对于每个训练样本，我们都将采用一个精简后神经网络来训练它，这种方法似乎有点怪，单纯遍历节点，编码也是随机的，可它真的有效。不过可想而知，我们针对每个训练样本训练规模极小的网络，最后你可能会认识到为什么要正则化网络，因为我们在训练极小的网络。

如何实施 **dropout** 呢？方法有几种，接下来的是最常用的方法，即 **inverted dropout**（反向随机失活）。出于完整性考虑，我们用一个三层（$l=3$）网络来举例说明。编码中会有很多涉及到 3 的地方。我只举例说明如何在某一层中实施 **dropout**。

首先要定义向量d，$d^{[3]}$表示一个三层的 **dropout** 向量：

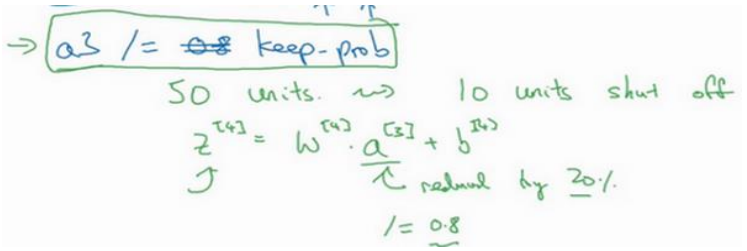
d3 = np.random.rand(a3.shape[0], a3.shape[1]) ，d3 中元素都是 0~1，假设 d3 维度是 4*1（也可能是 n*m），d3 = [[0.47148306], [0.79923346], [0.32820495], [0.86292134]]，然后看其中元素是否小于某数，我们称某数为 **keep-prob**，**keep-prob** 是一个具体数字，上个示例中它是 0.5，而本例中它是 0.8，它表示保留某个隐藏单元的概率，此处 **keep-prob** 等于 0.8，它意味着消除任意一个隐藏单元的概率是 0.2，也就是说 d3 中元素小于 0.8 则保留，大于 0.8 则消除，这就是生成随机矩阵的作用。

接下来要做的就是从第三层中获取激活函数，这里我们叫它$a^{[3]}$，$a^{[3]}$含有要计算的激活函数，$a^{[3]}$等于上面的$a^{[3]}$乘以$d^{[3]}$，a3 =np.multiply(a3,d3)，这里是元素相乘，也可写为$a3 *= d3$，它的作用就是让$d^{[3]}$中所有等于 0 的元素（输出），而各个元素等于 0 的概率只有 20%，乘法运算最终把$d^{[3]}$中相应元素输出，即让$d^{[3]}$中 0 元素与$a^{[3]}$中相对元素归零。



如果用 **python** 实现该算法的话，$d^{[3]}$则是一个布尔型数组，值为 **true** 和 **false**，而不是 1 和 0，乘法运算依然有效，**python** 会把 **true** 和 **false** 翻译为 1 和 0。我们向外扩展$a^{[3]}$，用它除以 0.8，或者除以 **keep-prob** 参数。$a3 \mathrel{/}= keep-prob$

下面解释一下为什么要这么做，为方便起见，假设第三隐藏层上有 50 个单元或 50 个神经元，在一维上$a^{[3]}$是 50，我们通过因子分解将它拆分成$50 \times m$维的，保留和删除它们的概率分别为 80%和 20%，这意味着最后被删除或归零的单元平均有 10（50×20%=10）个，现在我们看下$z^{[4]}$，$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$，我们的预期是$a^{[3]}$减少 20%，也就是说$a^{[3]}$中 20%的元素被归零，为了不影响$z^{[4]}$的期望值，我们需要用$w^{[4]}a^{[3]}/0.8$(此处除以 0.8 是数值上的变大，因为前面 d3 矩阵的元素都是 0~1 范围，被缩小了的)，它将会修正或弥补所需的 20%，$a^{[3]}$的期望值不会变，划线部分就是所谓的 **dropout** 方法。



反向随机失活（**inverted dropout**）方法通过除以 **keep-prob**，确保$a^{[3]}$的期望值不变。

事实证明，在测试阶段，当我们评估一个神经网络时，也就是用绿线框标注的反向随机失活方法，使测试阶段变得更容易，因为它的数据扩展问题变少。目前实施 **dropout** 最常用的方法就是 **Inverted dropout**。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。我用$a^{[0]}$，第 0 层的激活函数标注为测试样本x，我们在测试阶段不使用 **dropout** 函数，尤其是像下列情况：

$$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \cdots$$

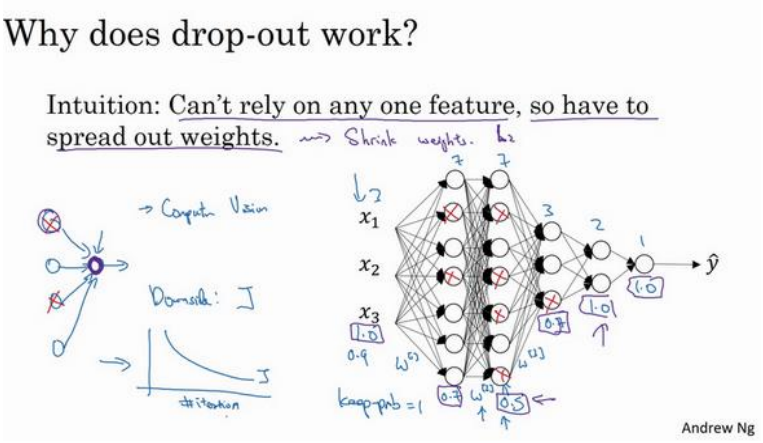
以此类推直到最后一层，预测值为$\hat{y}$。显然在测试阶段，我们并未使用 **dropout**，自然也就不用抛硬币来决定失活概率，以及要消除哪些隐藏单元了，因为在测试阶段进行预测时，我们不期望输出结果是随机的，如果测试阶段应用 **dropout** 函数，预测会受到干扰。

1.7 为什么 dropout 正则化可以防止过拟合？

Dropout 可以随机删除网络中的神经单元，他为什么可以通过正则化发挥如此大的作用呢？

直观上理解：不要依赖于任何一个特征，因为该单元的输入可能随时被清除，因此该单元通过这种方式传播下去，并为单元的四个输入增加一点权重，通过传播所有权重，**dropout** 将产生收缩权重的平方范数的效果，和之前讲的$L2$正则化类似；实施 **dropout** 的结果实它会压缩权重，并完成一些预防过拟合的外层正则化；$L2$对不同权重的衰减是不同的，它取决于激活函数倍增的大小。

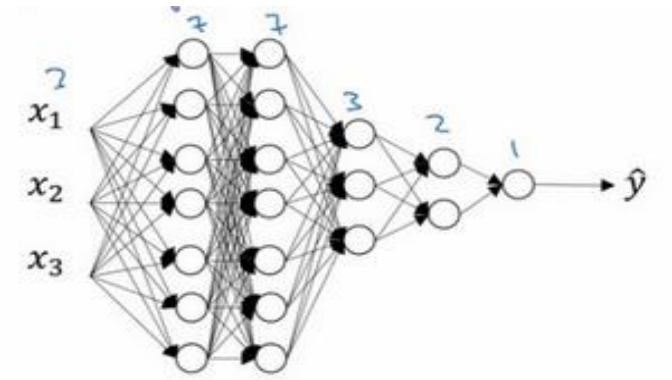
总结一下，**dropout** 的功能类似于$L2$正则化，与$L2$正则化不同的是应用方式不同会带来一点点小变化，甚至更适用于不同的输入范围。



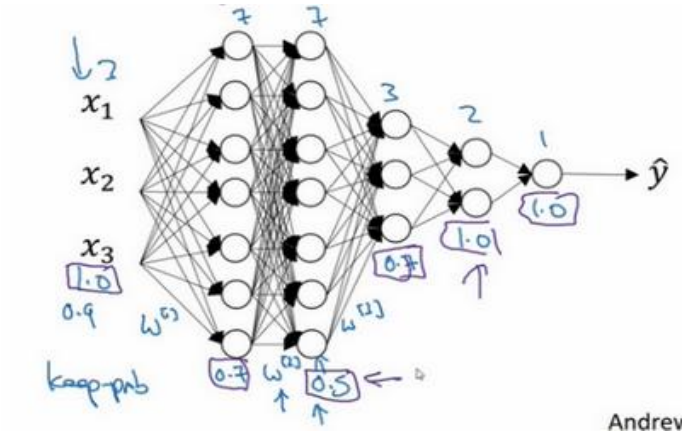
第二个直观认识是，我们从单个神经元入手，如图，这个单元的工作就是输入并生成一些有意义的输出。通过 **dropout**，该单元的输入几乎被消除，有时这两个单元会被删除，有时会删除其它单元，就是说，我用紫色圈起来的这个单元，它不能依靠任何特征，因为特征都有可能被随机清除，或者说该单元的输入也都可能被随机清除。我不愿意把所有赌注都放在一个节点上，不愿意给任何一个输入加上太多权重，因为它可能会被删除，因此该单元将通过这种方式积极地传播开，并为单元的四个输入增加一点权重，通过传播所有权重，**dropout** 将产生收缩权重的平方范数的效果，和我们之前讲过的$L2$正则化类似，实施 **dropout** 的结果是它会压缩权重，并完成一些预防过拟合的外层正则化。

事实证明，**dropout** 被正式地作为一种正则化的替代形式，$L2$对不同权重的衰减是不同的，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。



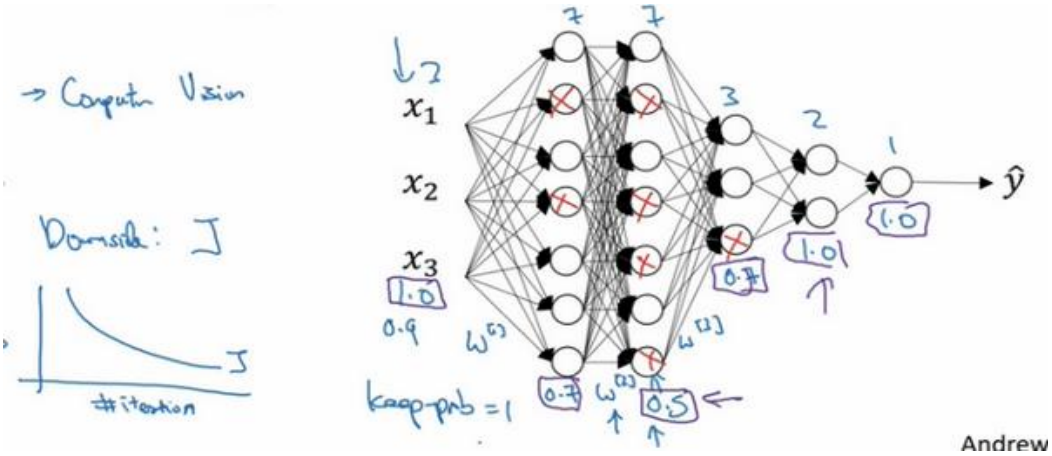
实施 **dropout** 的另一个细节是，这是一个拥有三个输入特征的网络，其中一个要选择的参数是 **keep-prob**，它代表每一层上保留单元的概率。所以不同层的 **keep-prob** 也可以变化。第一层，矩阵$W^{[1]}$是 7×3，第二个权重矩阵$W^{[2]}$是 7×7，第三个权重矩阵$W^{[3]}$是 3×7，以此类推，$W^{[2]}$是最大的权重矩阵，因为$W^{[2]}$拥有最大参数集，即 7×7，为了预防矩阵的过拟合，对于这一层，我认为这是第二层，它的 **keep-prob** 值应该相对较低，假设是 0.5。对于其它层，过拟合的程度可能没那么严重，它们的 **keep-prob** 值可能高一些，可能是 0.7，这里是 0.7。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，为了表达清除，我用紫色线笔把它们圈出来，每层 **keep-prob** 的值可能不同。



注意 **keep-prob** 的值是 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λ，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**，我们有机会删除一个或多个输入特征，虽然现实中我们通常不这么做，**keep-prob** 的值为 1，是非常常用的输入值，也可以用更大的值，或许是 0.9。但是消除一半的输入特征是不太可能的，如果我们遵守这个准则，**keep-prob** 会接近于 1，即使你对输入层应用 **dropout**。

总结一下，如果你担心某些层比其它层更容易发生过拟合，可以把某些层的 **keep-prob** 值设置得比其它层更低，缺点是为了使用交叉验证，你要搜索更多的超级参数，另一种方案是在一些层上应用 **dropout**，而有些层不用 **dropout**，应用 **dropout** 的层只含有一个超级参数，就是 **keep-prob**。

结束前分享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，输入太多像素，以至于没有足够的数据，所以 **dropout** 在计算机视觉中应用得比较频繁，有些计算机视觉研究人员非常喜欢用它，几乎成了默认的选择，但要牢记一点，**dropout** 是一种正则化方法，它有助于预防过拟合，因此除非算法过拟合，不然我是不会使用 **dropout** 的，所以它在其它领域应用得比较少，主要存在于计算机视觉领域，因为我们通常没有足够的数据，所以一直存在过拟合，这就是有些计算机视觉研究人员如此钟情于 **dropout** 函数的原因。直观上我认为不能概括其它学科。



dropout 一大缺点就是代价函数J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，因为我们所优化的代价函数 J 实际上并没有明确定义，或者说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的图片。我通常会关闭 **dropout** 函数，将 **keep-prob** 的值设为 1，运行代码，确保J函数单调递减。然后打开 **dropout** 函数，希望在 **dropout** 过程中，代码并未引入 bug。我觉得你也可以尝试其它方法，虽然我们并没有关于这些方法性能的数据统计，但你可以把它们与 **dropout** 方法一起使用。

1.8 其他正则化方法

除了$L2$正则化和随机失活（**dropout**）正则化，还有几种减少神经网络过拟合的方法:

一、数据扩增

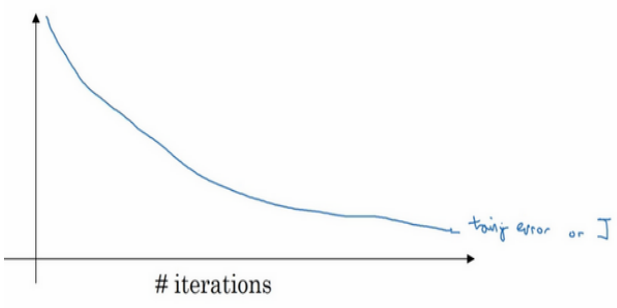
假设你正在拟合猫咪图片分类器，如果你想通过扩增训练数据来解决过拟合，但扩增数据代价高，而且有时候无法扩增数据，我们可以通过添加这类图片来增加训练集。例如，**水平翻转图片**，并把它添加到训练集。现在训练集中有原图，还有翻转后的这张图片，训练集则可以增大一倍，这虽然不如我们额外收集一组新图片那么好，但这样做节省了获取更多猫咪图片的花费。除了水平翻转图片，也可以**随意裁剪图片**，这张图是把原图旋转并随意放大后裁剪的，仍能辨别出图片中的猫咪。大家注意，我并没有垂直翻转，因为我们不想上下颠倒图片，也可以**随机选取放大后的部分图片**，猫可能还在上面。

通过随意翻转和裁剪图片，我们可以增大数据集，额外生成假训练数据。和全新的，独立的猫咪图片数据相比，这些额外的假的数据无法包含像全新数据那么多的信息，但我们这么做基本没有花费，代价几乎为零，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。

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，它们仍然是数字。

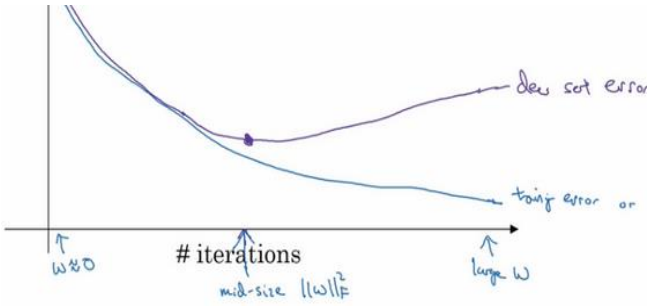
二、提前停止训练（**early stopping**）

另外一种常用的方法叫作 **early stopping**，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，呈单调下降趋势，在训练集上用 0-1 记录分类误差次数。

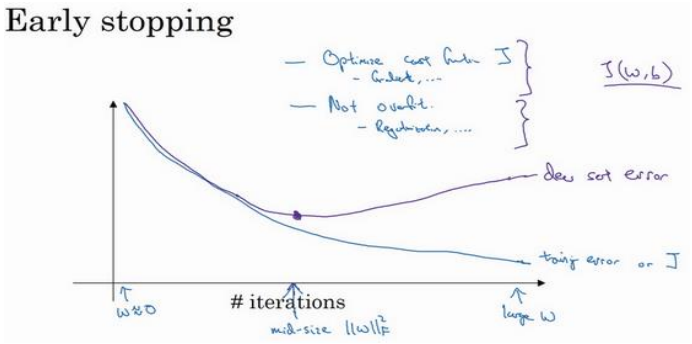


通过 **early stopping**，不但可以绘制上面这些内容，还可以绘制验证集误差，它可以是验证集上的分类误差，或验证集上的代价函数，逻辑损失和对数损失等，你会发现，验证集误差通常会先呈下降趋势，然后在某个节点处开始上升，**early stopping** 的作用是，神经网络已经在迭代过程中表现得很好了，我们在此停止训练吧，得到验证集误差，它是怎么发挥作用的？

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值时，它的值可能是较小的随机值，所以在长期训练神经网络之前w依然很小，在迭代过程和训练过程中w的值会变得越来越大，比如在这儿，神经网络中参数w的值已经非常大了，所以 **early stopping** 要做就是在中间点停止迭代过程，我们得到一个w值，中等大小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。



术语 **early stopping** 代表提早停止训练神经网络，训练神经网络时，我有时会用到 **early stopping**，但是它也有一个缺点。因为提早停止梯度下降，也就是停止了优化代价函数J，因为现在不再尝试降低代价函数J，所以代价函数J的值可能不够小，同时又希望不出现过拟合。机器学习过程包括几个步骤，其中一步是选择算法来优化代价函数J，如梯度下降，但是优化代价函数J之后，我也不想发生过拟合，也有一些工具可以解决该问题，比如正则化，扩增数据等等。



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。我发现，这导致超参数搜索空间更容易分解，也更容易搜索，但是缺点在于，你必须尝试很多正则化参数λ的值，这也导致搜索大量λ值的计算代价太高。

Early stopping 的优点是，只运行一次梯度下降，你可以找出w的较小值，中间值和较大值，而无需尝试$L2$正则化超级参数λ的很多值。

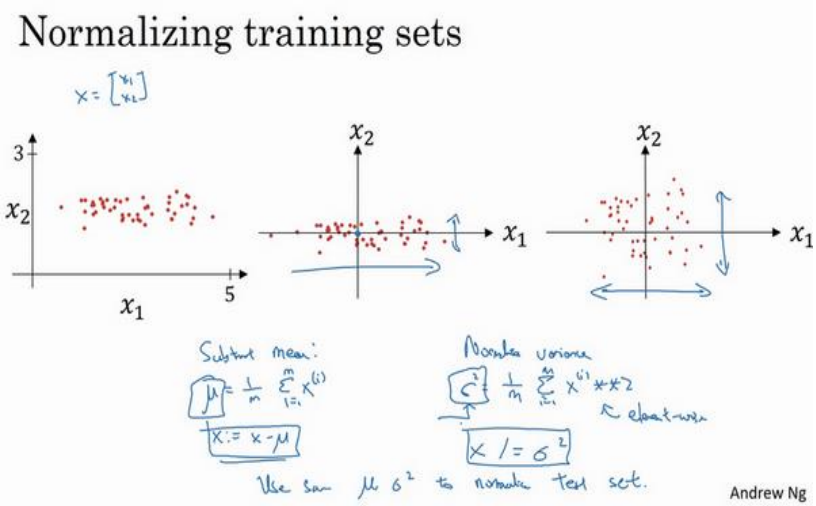
1.9 归一化输入

训练神经网络，其中一个加速训练的方法就是归一化。假设一个训练集有两个特征，输入特征为 2 维，归一化有两个步骤：

第一步是**零均值化**，$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$，它是一个向量，因为每个特征都有均值，$x$等于每个训练数据 x 减去 μ，意思是移动训练集，直到所有输入数据的均值是 0。

第二步是**归一化方差**，特征 x_1 的方差比特征 x_2 的方差要大得多，$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2$，这是节点 y 的平方，σ^2是一个向量，它的每个特征都有方差，注意，x 已经完成零值均化，把所有数据除以向量σ^2，最后变成上图形式。

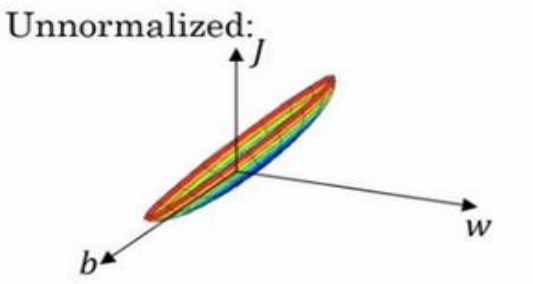
你不希望训练集和测试集的归一化有所不同，所以要用同样的方法，用在训练集上得到的 μ 和 σ^2 来调整测试集，而不是在训练集和测试集上分别计算 μ 和 σ^2。因为我们希望不论是训练数据还是测试数据，都是通过相同 μ 和σ^2定义的相同数据转换。



我们为什么要这么做呢？为什么想要归一化输入特征，回想一下所定义的代价函数。

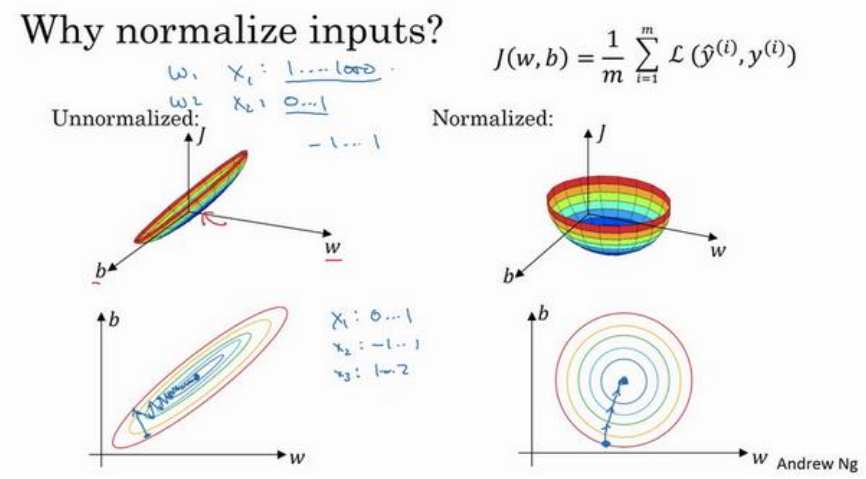
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

如果你使用非归一化的输入特征，代价函数会像这样：



这是一个非常细长狭窄的代价函数，有点像狭长的碗一样，最小值应该在碗底。但如果特征值在不同范围，假如x_1取值范围从 1 到 1000，特征x_2的取值范围从 0 到 1，结果是参数w_1和w_2值的范围或比率将会非常不同，这些数据轴应该是w_1和w_2，但直观理解，我标记为w和b，如果你能画出该函数的部分轮廓，它会是这样一个狭长的函数。

然而如果你归一化特征，代价函数平均起来看更对称，如果你在上图这样的代价函数上运行梯度下降法，你必须使用一个非常小的学习率。因为如果是在这个位置，梯度下降法可能需要多次迭代过程，直到最后找到最小值。但如果函数是一个更圆的球形轮廓，那么不论从哪个位置开始，梯度下降法都能够更直接地找到最小值，你可以在梯度下降法中使用较大步长，而不需要像在左图中那样反复执行。



实际上如果假设特征x_1范围在 0-1 之间，x_2的范围在-1 到 1 之间，x_3范围在 1-2 之间，它们是相似范围，所以会表现得很好。当它们在非常不同的取值范围内，如其中一个从 1 到 1000，另一个从 0 到 1，这对优化算法非常不利。但是仅将它们设置为均化零值，假设方差为 1，就像上一张幻灯片里设定的那样，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得更快。

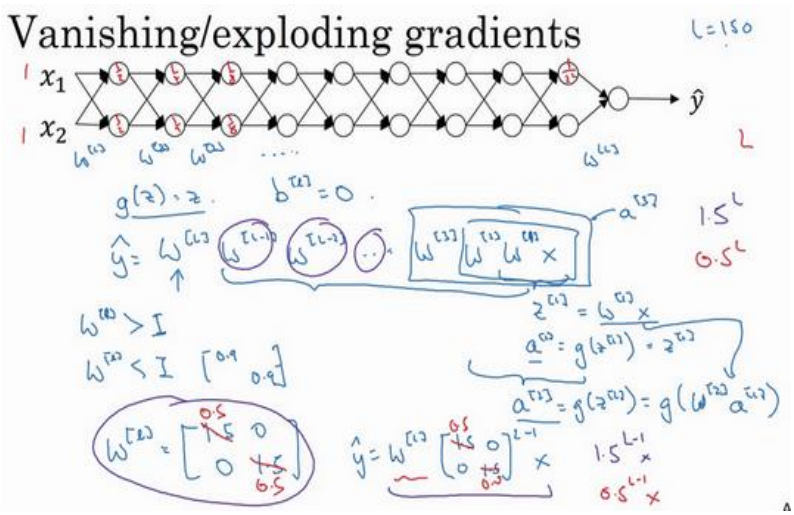
1.10 梯度消失/梯度爆炸

训练神经网络所面临的一个问题就是梯度消失或梯度爆炸，也就是训练神经网络时，导数有时会变得非常大或非常小，甚至于以指数方式变小，这加大了训练的难度。

假设你正在训练这样一个极深的神经网络，这个神经网络会有参数$W^{[1]}$，$W^{[2]}$，$W^{[3]}$，直到$W^{[l]}$，为了简单起见，假设我们使用激活函数$g(z)=z$，也就是线性激活函数，我们假设 $b^{[l]}=0$，如果这样的话，输出：

$$y=W^{[l]}W^{[L-1]}W^{[L-2]}\ldots W^{[3]}W^{[2]}W^{[1]}x$$

$W^{[1]}x=z^{[1]}$，因为$b=0$，所以$z^{[1]}=W^{[1]}x$，$a^{[1]}=g(z^{[1]})$，因为使用了线性激活函数，$a^{[1]}=z^{[1]}$，同理可得 $W^{[2]}W^{[1]}x=a^{[2]}$，……，$y=W^{[l]}W^{[L-1]}W^{[L-2]}\ldots W^{[3]}W^{[2]}W^{[1]}x$。



假设每个权重矩阵$W^{[l]}=\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}$，最后一项有不同维度，可能它就是余下的权重矩阵，$y=W^{[1]}\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}^{(L-1)}x$，最后计算结果就是$\hat{y}$，也就是$1.5^{(L-1)}x$。如果对于一个深度神经网络来说，$L$ 值较大，那么 $\hat{y}$ 值也会非常大，实际上它呈指数级增长的，因此对于一个深度神经网络，y的值将爆炸式增长。

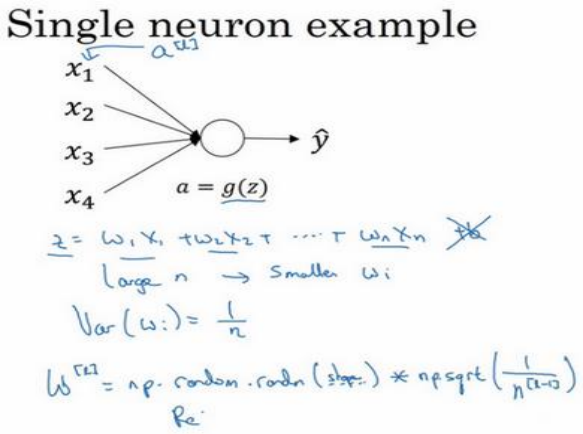
相反的，如果权重是 0.5，$W^{[l]}=\begin{bmatrix}0.5 & 0\\ 0 & 0.5\end{bmatrix}$，它比 1 小，这项也就变成了$0.5^L$，矩阵$y=W^{[1]}\begin{bmatrix}0.5 & 0\\ 0 & 0.5\end{bmatrix}^{(L-1)}x$，再次忽略$W^{[L]}$，因此每个矩阵都小于 1，假设$x_1$和$x_2$都是 1，激活函数将变成$\frac{1}{2}$，$\frac{1}{2}$，$\frac{1}{4}$，$\frac{1}{4}$，$\frac{1}{8}$，$\frac{1}{8}$等，直到最后一项变成$\frac{1}{2^L}$，激活函数的值将以指数级下降，它是与网络层数数量$L$相关的函数，在深度网络中，激活函数以指数级递减。

直观理解是，权重W只比 1 略大一点，或者说只是比单位矩阵大一点，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，深度神经网络的激活函数将爆炸式减小。

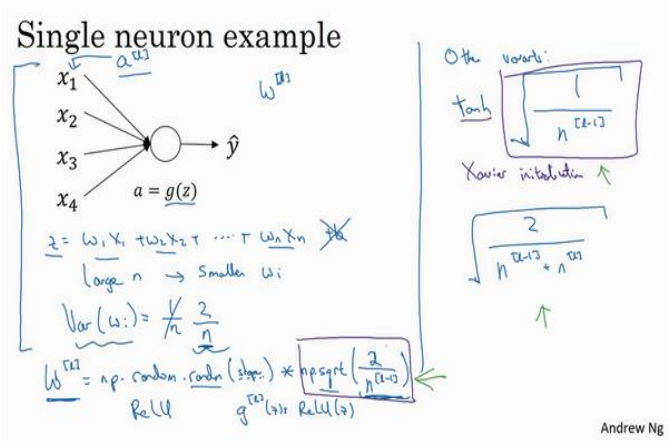
1.11 神经网络的权重初始化

我们学习了深度神经网络如何产生梯度消失和梯度爆炸问题，针对该问题，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，有助于为神经网络更谨慎地选择随机初始化参数，我们先举一个神经单元的例子，然后再演变到整个深度网络。

单个神经元可能有 4 个输入特征，从x_1到x_4，经过$a = g(z)$处理，最终得到$\hat{y}$，稍后讲深度网络时，这些输入表示为$a^{[l]}$。$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$，$b = 0$，暂时忽略$b$，为了预防$z$值过大或过小，看到 n 越大，你希望 w_i 越小，因为z是w_ix_i的和，如果把很多此类项相加，希望每项值更小，最合理的方法就是设置$w_i = \frac{1}{n}$，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$w^{[l]} = np.random.randn(\text{shape}) * \text{np.sqrt}(\frac{1}{n^{[l-1]}})$，$n^{[l-1]}$就是喂给第$l$层神经单元的数量（即第$l-1$层神经元数量）。



结果，如果你是用的是 **Relu** 激活函数，而不是$\frac{1}{n}$，方差设置为$\frac{2}{n}$，效果会更好。你常常发现，初始化时，尤其是使用 **Relu** 激活函数时，$g^{[l]}(z) = Relu(z)$，它取决于你对随机变量的熟悉程度，这是高斯随机变量，然后乘以它的平方根，也就是引用这个方差$\frac{2}{n}$。这里，我用的是$n^{[l-1]}$，因为本例中，逻辑回归的特征是不变的。但一般情况下l层上的每个神经元都有$n^{[l-1]}$个输入。如果激活函数的输入特征被零均值和标准方差化，方差是 1，z也会调整到相似范围，这没有真正解决问题（梯度消失和爆炸问题）。但它确实降低了梯度消失和爆炸问题，因为它给权重矩阵w设置了合理值，你也知道，它不能比 1 大很多，也不能比 1 小很多，所以梯度没有爆炸或消失过快。



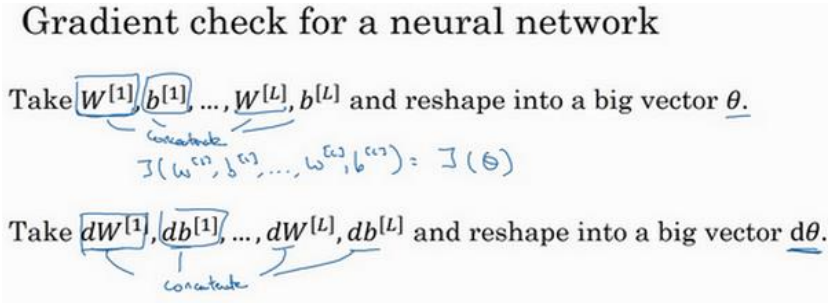
我提到了其它变体函数，刚刚提到的函数是 **Relu** 激活函数。对于几个其它变体函数，如 **tanh** 激活函数，有篇论文提到，常量 1 比常量 2 的效率更高，对于 **tanh** 函数来说，它是 $\sqrt{\frac{1}{n^{[l-1]}}}$，这里平方根的作用与这个公式作用相同($\text{np.sqrt}(\frac{1}{n^{[l-1]}})$)，它适用于 **tanh** 激活函数，被称为 **Xavier** 初始化。

实际上，我认为所有这些公式只是给你一个起点，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参数，可以给公式 $\text{np.sqrt}(\frac{2}{n^{[l-1]}})$添加一个乘数参数，调优作为超级参数激增一份子的乘子参数。有时调优该超级参数效果一般，这并不是我想调优的首要超级参数，但我发现调优过程中产生的问题，虽然调优该参数能起到一定作用，但考虑到相比调优，其它超级参数的重要性，我通常把它的优先级放得比较低。

1.12 梯度检验

假设网络中含有下列参数，$W^{[1]}$和$b^{[1]}$……$W^{[l]}$和$b^{[l]}$，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，你要做的就是把矩阵W转换成一个向量，把所有W矩阵转换成向量之后，做连接运算，得到一个巨型向量θ，该向量表示为参数θ，代价函数J是所有W和b的函数，现在你得到了一个θ的代价函数J（即$J(\theta)$）。接着，你得到与W和b顺序相同的数据，你同样可以把$dW^{[1]}$和$db^{[1]}$……$dW^{[l]}$和$db^{[l]}$转换成一个新的向量，用它们来初始化大向量$d\theta$，它与θ具有相同维度。

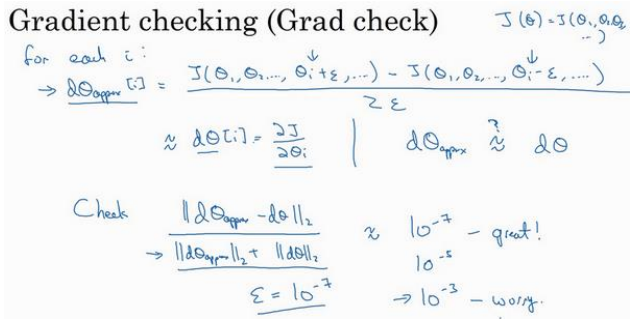
同样的，把$dW^{[1]}$转换成矩阵，$db^{[1]}$已经是一个向量了，直到把$dW^{[l]}$转换成矩阵，这样所有的dW都已经是矩阵，注意$dW^{[1]}$与$W^{[1]}$具有相同维度，$db^{[1]}$与$b^{[1]}$具有相同维度。经过相同的转换和连接运算操作之后，你可以把所有导数转换成一个大向量$d\theta$，它与θ具有相同维度，现在的问题是$d\theta$和代价函数J的梯度或坡度有什么关系？



这就是实施梯度检验，英语里简称为“**gradient check**”，首先，我们要清楚J是超参数θ的一个函数，你也可以将J函数展开为$J(\theta_1, \theta_2, \theta_3, \dots\dots)$，不论超级参数向量$\theta$的维度是多少，为了实施梯度检验，你要做的就是循环执行，从而对每个i也就是对每个θ组成元素计算$d\theta_{\text{approx}}[i]$的值，我使用双边误差，也就是

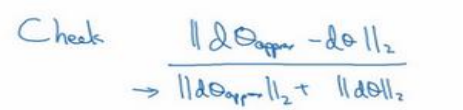
$$d\theta_{\text{approx}}[i] = \frac{J(\theta_1, \theta_2, \dots \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots \theta_i - \varepsilon, \dots)}{2\varepsilon}$$

只对θ_i增加ε，其它项保持不变，因为我们使用的是双边误差，对另一边做同样的操作，只不过是减去ε，θ其它项全都保持不变。

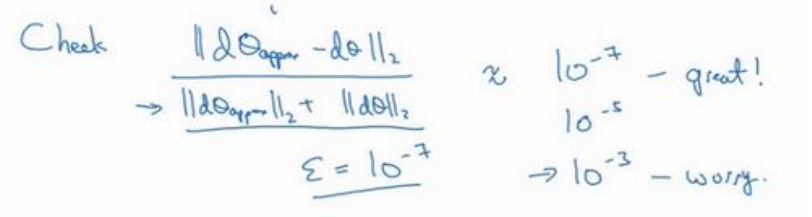


从上节课中我们了解到这个值（$d\theta_{\text{approx}}[i]$）应该逼近$d\theta[i]$=$\frac{\partial J}{\partial \theta_i}$，$d\theta[i]$是代价函数的偏导数，然后你需要对$i$的每个值都执行这个运算，最后得到两个向量，得到$d\theta$的逼近值

$d\theta_{\text{approx}}$，它与$d\theta$具有相同维度，它们两个与θ具有相同维度，你要做的就是验证这些向量是否彼此接近。



具体来说，如何定义两个向量是否真的接近彼此？我一般做下列运算，计算这两个向量的距离，$d\theta_{\text{approx}}[i] - d\theta[i]$的欧几里得范数，注意这里（$||d\theta_{\text{approx}} - d\theta||_2$）是误差平方之和，然后求平方根，得到欧式距离，然后用向量长度归一化，使用向量长度的欧几里得范数。分母只是用于预防这些向量太小或太大，分母使得这个方程式变成比率，我们实际执行这个方程式，ε可能为10^{-7}，使用这个取值范围内的ε，如果你发现计算方程式得到的值为10^{-7}或更小，这就很好，这就意味着导数逼近很有可能是正确的，它的值非常小。



如果它的值在10^{-5}范围内，我就要小心了，也许这个值没问题，但我会再次检查这个向量的所有项，确保没有一项误差过大，可能这里有 **bug**。

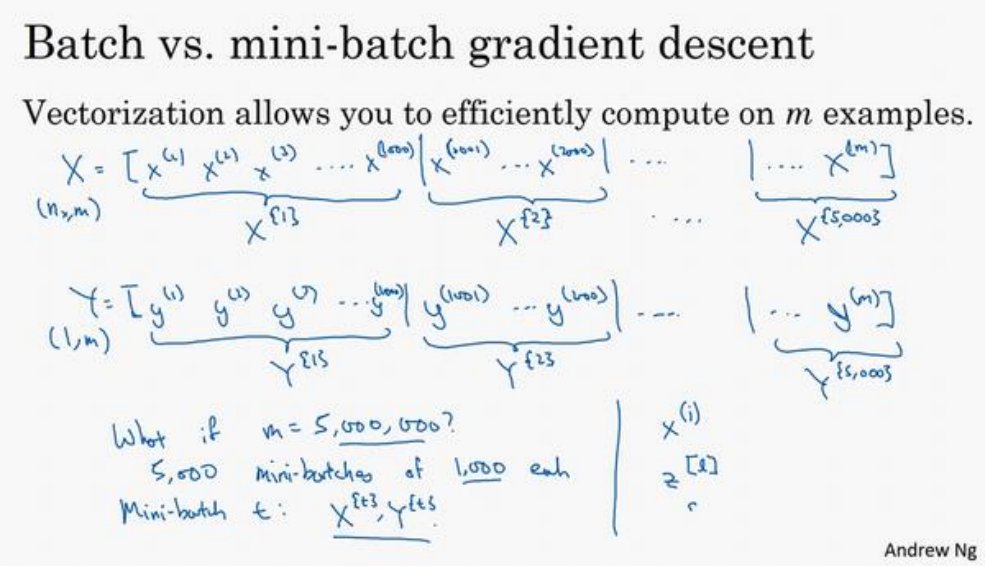
如果左边这个方程式结果是10^{-3}，就会很担心是否存在 **bug**。这时应该仔细检查所有θ项，看是否有一个具体的i值，使得$d\theta_{\text{approx}}[i]$与$d\theta[i]$大不相同，并用它来追踪一些求导计算是否正确，经过一些调试，最终结果会是这种非常小的值（10^{-7}）。

在实施神经网络时，经常需要执行 **foreprop** 和 **backprop**，然后我可能发现这个梯度检验有一个相对较大的值，我会怀疑存在 **bug**，然后开始调试，调试，调试，调试一段时间后，我得到一个很小的梯度检验值，现在我可以很自信的说，神经网络实施是正确的。

第二周：优化算法

2.1 Mini-batch 梯度下降

本周将学习优化算法，这能让你的神经网络运行得更快。机器学习的应用是一个高度依赖经验的过程，伴随着大量迭代的过程，你需要训练诸多模型，才能找到合适的那一个，所以，优化算法能够帮助你快速训练模型。其中一个难点在于，深度学习没有在大数据领域发挥最大的效果，我们可以利用一个巨大的数据集来训练神经网络，而在巨大的数据集上进行训练速度很慢。因此，使用快速的优化算法能够大大提高效率。

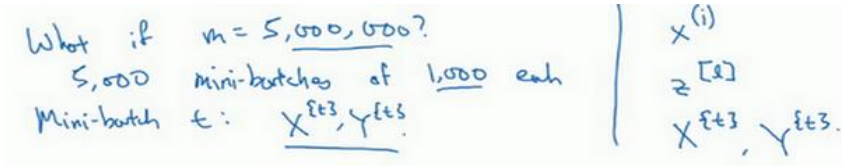


我们之前学过，向量化能够让你有效地对所有m个样本进行计算，而无需某个明确的公式，所以要把训练样本放大巨大的矩阵 X 中，$X = \left[x^{(1)}\ x^{(2)}\ x^{(3)} \ldots\ldots x^{(m)}\right]$，$Y$也是如此，$Y = \left[y^{(1)}\ y^{(2)}\ y^{(3)} \ldots\ldots y^{(m)}\right]$。$X$维数是$(n_x, m)$，$Y$的维数是$(1, m)$，向量化能够相对较快地处理所有$m$个样本，但如果$m$很大的话，处理速度仍然缓慢。比如说，如果$m$是 500 万或 5000 万，在对整个训练集执行梯度下降法时，必须处理整个训练集，然后才能进行一步梯度下降法，然后再重新处理 500 万个训练样本，才能进行下一步梯度下降法。

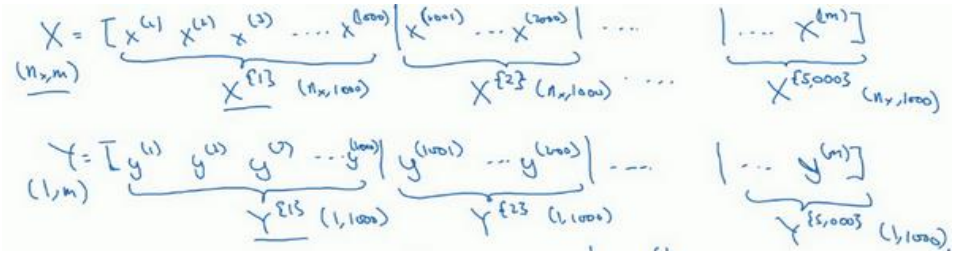
如果在处理完整个 500 万个样本的训练集之前，先让梯度下降法处理一部分，算法速度会更快。我们可以把训练集分割为子集训练，取名为 **mini-batch**，假设每个子集中只有 1000 个样本，那么把其中的$x^{(1)}$到$x^{(1000)}$取出来，将其称为第一个子训练集，也叫做 **mini-batch**，然后再取出接下来的 1000 个样本，从$x^{(1001)}$到$x^{(2000)}$，然后再取 1000 个样本，以此类推。

我们定义$x^{(1)}$到$x^{(1000)}$称为$X^{\{1\}}$，$x^{(1001)}$到$x^{(2000)}$称为$X^{\{2\}}$，如果训练样本一共有 500 万个，每个 **mini-batch** 都有 1000 个样本，也就是说有 5000 个 **mini-batch**，所以最后得到是$X^{\{5000\}}$，对Y也进行相同处理，相应地拆分Y的训练集，所以$y^{(1)}$到$y^{(1000)}$称为$Y^{\{1\}}$，然后从$y^{(1001)}$到$y^{(2000)}$，这个叫$Y^{\{2\}}$，一直到$Y^{\{5000\}}$。

mini-batch 的数量t组成了$X^{\{t\}}$和$Y^{\{t\}}$，这就是 1000 个训练样本，包含相应输入输出。



之前我们使用了上角小括号(i)表示训练集里的值，所以$x^{(i)}$是第i个训练样本。我们用了上角中括号$[l]$来表示神经网络的层数，$z^{[l]}$表示神经网络中第l层的z值，现在引入了大括号t来代表不同的 **mini-batch**，所以我们有$X^{\{t\}}$和$Y^{\{t\}}$。



$X^{\{t\}}$和$Y^{\{t\}}$的维数：如果$X^{\{1\}}$是一个有 1000 个样本的训练集，$(n_x, 1000)$，$X^{\{2\}}$的维数是$(n_x, 1000)$，以此类推。因此所有的子集维数都是$(n_x, 1000)$，而这些（$Y^{\{t\}}$）的维数都是$(1,1000)$。相比之下，**mini-batch** 梯度下降法，指的是每次处理$X^{\{t\}}$和$Y^{\{t\}}$，而不是同时处理全部的X和Y训练集。那么究竟 **mini-batch** 梯度下降法的原理是什么？

在训练集上运行 **mini-batch** 梯度下降法，你运行 `for t=1……5000`，因为我们有 5000 个各有 1000 个样本的组，在 **for** 循环里你要做得基本就是对$X^{\{t\}}$和$Y^{\{t\}}$执行一步梯度下降法。假设你有一个拥有 1000 个样本的训练集，而且假设你已经很熟悉一次性处理完的方法，你要用向量化去几乎同时处理 1000 个样本。

首先对输入也就是$X^{\{t\}}$，执行前向传播，然后执行$z^{[1]} = W^{[1]}X + b^{[1]}$，处理第一个 **mini-batch**，在处理 **mini-batch** 时它变成了$X^{\{t\}}$，即$z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$，然后执行$A^{[1]k} = g^{[1]}(Z^{[1]})$，以此类推，直到$A^{[L]} = g^{[L]}(Z^{[L]})$，这就是预测值。注意这里需要用到一个向量化的执行命令，一次性处理 1000 个而不是 500 万个样本。

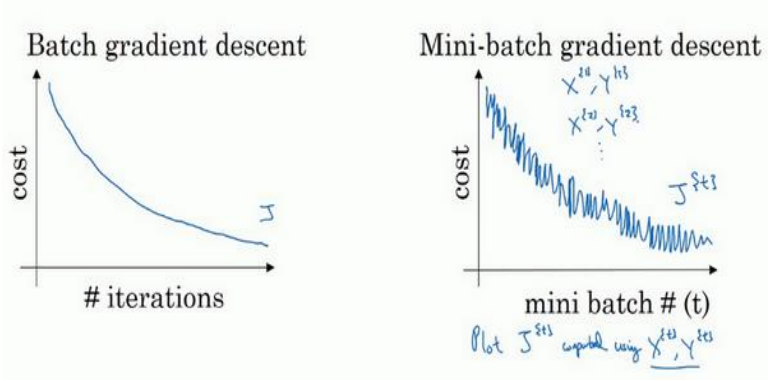
接下来要计算损失成本函数J，因为子集规模是 1000，$J = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)})$，这里的（$L(\hat{y}^{(i)}, y^{(i)})$）是指来自于 **mini-batch** $X^{\{t\}}$和$Y^{\{t\}}$中的样本。如果加上正则化，$J = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*1000}\sum_{l} ||w^{[l]}||_F^2$，因为这是一个 **mini-batch** 的损失，所以我将J损失记为上角标t，放在大括号里（$J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*1000}\sum_{l} ||w^{[l]}||_F^2$）。

我们做的似曾相识，其实跟之前的梯度下降法如出一辙，除了现在对象不是X，Y，而是$X^{\{t\}}$和$Y^{\{t\}}$。接下来，执行反向传播来计算$J^{\{t\}}$的梯度，然后更新加权值，W实际上是$W^{[l]}$，更新为$W^{[l]} := W^{[l]} - adW^{[l]}$，对$b$做相同处理，$b^{[l]} := b^{[l]} - adb^{[l]}$。这是使用 **mini-batch** 梯度下降法训练样本的一步，也可被称为进行“一代”（**1 epoch**）的训练。一代这个词意味着只是一次遍历了训练集。使用 **batch** 梯度下降法，一次遍历训练集只能让你做一个梯度下降，使用 **mini-batch** 梯度下降法，一次遍历训练集，能让你做 5000 个梯度下降。

2.2 理解 mini-batch 梯度下降法

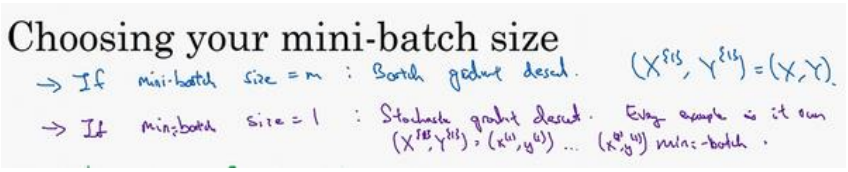
使用 **batch** 梯度下降法时，每次迭代都需要遍历整个训练集，可以预期每次迭代成本都会下降，所以成本函数 J 是迭代次数的一个函数，它应该会随着每次迭代而减少，如果J在某次迭代中增加了，那肯定出了问题，也许学习率太大。

使用 **mini-batch** 梯度下降法，如果作出成本函数在整个过程中的图，则并不是每次迭代都是下降的，特别是在每次迭代中，你要处理的是$X^{\{t\}}$和$Y^{\{t\}}$，如果要作出成本函数$J^{\{t\}}$的图，而$J^{\{t\}}$只和$X^{\{t\}}$，$Y^{\{t\}}$有关，也就是每次迭代下你都在训练不同的样本集或者说训练不同的 **mini-batch**，所以很可能会看到这样的结果，走向朝下，但有更多的噪声，没有每次迭代都下降，但走势应该向下，噪声产生的原因在于也许$X^{\{1\}}$和$Y^{\{1\}}$是比较容易计算的 **mini-batch**，因此成本会低一些。不过也许出于偶然，$X^{\{2\}}$和$Y^{\{2\}}$是比较难运算的 **mini-batch**，或许你需要一些残缺的样本，这样一来，成本会更高一些，所以才会出现这些摆动。



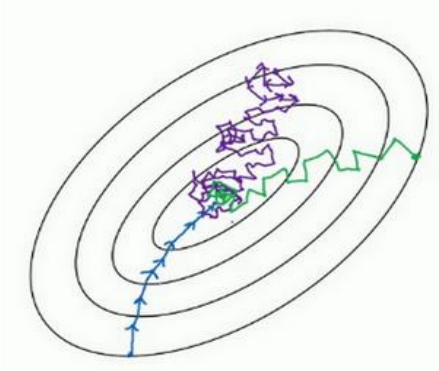
需要决定的变量之一是 **mini-batch** 的大小，m是训练集的大小，这里有两种极端情况：

①如果 **mini-batch** 的大小等于m，就是 **batch** 梯度下降法，在这种极端情况下，就有了 **mini-batch** $X^{\{1\}}$和$Y^{\{1\}}$，并且该 **mini-batch** 等于整个训练集，所以把 **mini-batch** 大小设为m可以得到 **batch** 梯度下降法。此法特点是：**batch** 梯度下降法从某处开始，相对噪声低些，精度比较高，所以幅度可以设置大一些，可以继续找最小值，尽管如此，速度也很慢。



②如果 **mini-batch** 大小为 1，就是随机梯度下降法，每个样本都是独立的 **mini-batch**，一次只处理一个。当你看第一个 **mini-batch**，也就是$X^{\{1\}}$和$Y^{\{1\}}$，它就是第一个训练样本。接着再看第二个 **mini-batch**，也就是第二个训练样本，采取梯度下降步骤，然后是第三个训练样本，以此类推。此法特点是：每次只计算一个样本，迭代速度快，但不一定每次都朝着收敛的方向，因此随机梯度下降法是有很多噪声的，平均来看，它最终会靠近最小值，不过有时候也会方向错误，因为随机梯度下降法永远不会收敛，而是会一直在最小值附近波动，但它并不会在达到最小值并停留在此。

而 **mini-batch** 大小在 1 和m之间，而 1 太小了，m太大了，原因在于如果使用 **batch** 梯度下降法，**mini-batch** 的大小为m，每个迭代需要处理大量训练样本，该算法的主要弊端在于特别是在训练样本数量巨大的时候，单次迭代耗时太长。如果训练样本不大，**batch** 梯度下降法运行地很好。用 **mini-batch** 梯度下降法，我们从这里开始，一次迭代，两次，三次，四次，它不会总朝向最小值靠近，但它比随机梯度下降要更持续地靠近最小值的方向，它也不一定在很小的范围内收敛或者波动，如果出现这个问题，可以慢慢减少学习率。



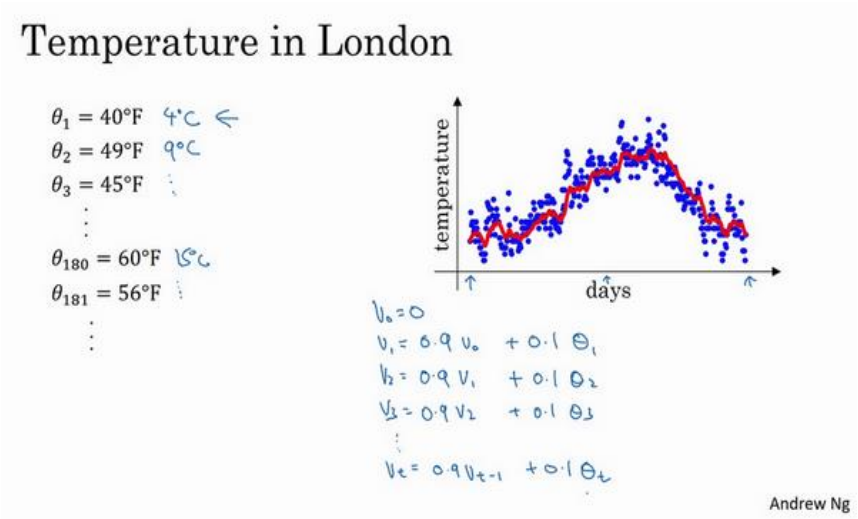
如果 **mini-batch** 大小既不是 1 也不是m，应该取中间值，那应该怎么选择呢？其实是有指导原则的。

首先，如果训练集较小，直接使用 **batch** 梯度下降法，可以快速处理整个训练集，所以使用 **batch** 梯度下降法也很好，这里的少是说小于 2000 个样本，这样比较适合 **batch** 梯度下降法。不然，样本数目较大的话，一般的 **mini-batch** 大小为 64 到 512，考虑到电脑内存设置和使用的方式，如果 **mini-batch** 大小是 2 的n次方，代码会运行地快一些，64 就是 2 的 6 次方，以此类推，128 是 2 的 7 次方，256 是 2 的 8 次方，512 是 2 的 9 次方。所以我经常把 **mini-batch** 大小设成 2 的次方。在上一个视频里，我的 **mini-batch** 大小设为了 1000，建议你可以试一下 1024，也就是 2 的 10 次方。也有 **mini-batch** 的大小为 1024，不过比较少见，64 到 512 的 **mini-batch** 比较常见。

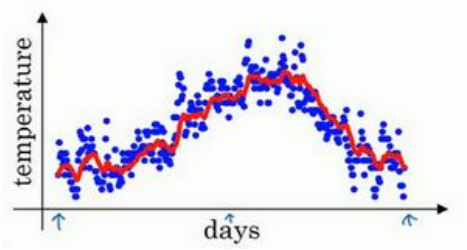
最后需要注意的是，在 **mini-batch** 中，要确保$X^{\{t\}}$和$Y^{\{t\}}$要符合 **CPU/GPU** 内存，取决于你的应用方向以及训练集的大小。如果你处理的 **mini-batch** 和 **CPU/GPU** 内存不相符，不管你用什么方法处理数据，你会注意到算法的表现急转直下变得惨不忍睹，所以我希望你对一般人们使用的 **mini-batch** 大小有一个直观了解。事实上 **mini-batch** 大小是另一个重要的变量，你需要做一个快速尝试，才能找到能够最有效地减少成本函数的那个，一般会尝试几个不同的值，几个不同的 2 次方，然后看能否找到一个让梯度下降优化算法最高效的大小。

2.3 指数加权平均数

有几个优化算法，它们比梯度下降法快，要理解这些算法，需要用到指数加权平均（Exponentially weighted averages），我们首先学这个，然后再学更复杂的优化算法。



这里是 1 月 1 号，年中接近夏季的时候，随后就是年末的数据，看起来有些杂乱，如果要计算趋势的话，也就是温度的局部平均值，或者说移动平均值。要做的是，首先使$v_0 = 0$，每天，需要使用 0.9 的加权数之前的数值加上当日温度的 0.1 倍，即$v_1 = 0.9v_0 + 0.1\theta_1$，所以这里是第一天的温度值。第二天，又可以获得一个加权平均数，0.9 乘以之前的值加上当日的温度 0.1 倍，即$v_2 = 0.9v_1 + 0.1\theta_2$，以此类推。第二天值加上第三日数据的 0.1，如此往下。大体公式就是某天的v等于前一天v值的 0.9 加上当日温度的 0.1。如此计算，然后用红线作图的话，便得到这样的结果。

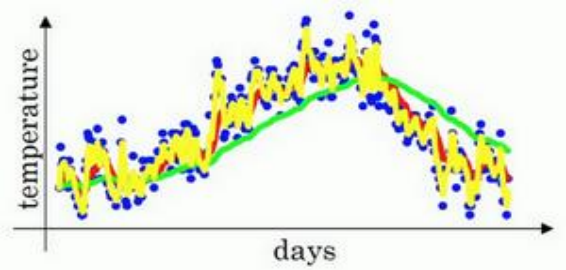


你得到了移动平均值，每日温度的指数加权平均值。看一下这里的公式，$v_t = 0.9v_{t-1} + 0.1\theta_t$，我们把 0.9 这个常数变成$\beta$，将之前的 0.1 变成$(1-\beta)$，即$v_t = \beta v_{t-1} + (1-\beta)\theta_t$。由于以后我们要考虑的原因，在计算时可视$v_t$大概是$\frac{1}{(1-\beta)}$×每日温度：

①如果β=0.9，这是十天的平均值，也就是红线部分。

②如果β=0.98，计算$\frac{1}{(1-0.98)} = 50$，粗略平均过去 50 天的温度，作图可得绿线。这个大的β要注意几点，得到的曲线平坦一些，原因在于多平均了几天的温度，所以曲线波动更小，更加平坦，缺点是曲线进一步右移，会出现一定延迟，因为$\beta = 0.98$，相当于给前一天的值加了太多权重，只有 0.02 的权重给了当日的值，因此此曲线缓慢适应温度变化。

③如果β=0.5，根据右边的公式（$\frac{1}{(1-\beta)}$），这是平均了两天的温度，作图运行后得到黄线。由于仅平均了两天的温度，平均的数据太少，所以得到的曲线有更多的噪声，有可能出现异常值，但是这个曲线能够更快适应温度变化。



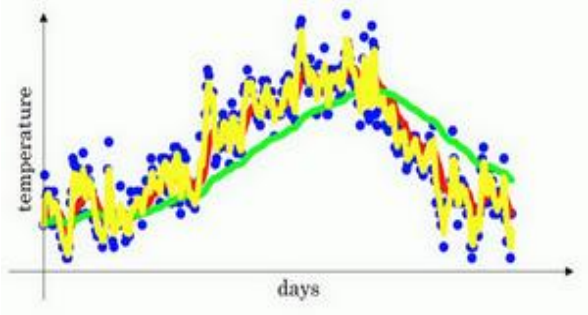
指数加权平均数经常被使用，通过调整这个参数（β），或者说后面的算法学习，你会发现这是一个很重要的参数，可以取得稍微不同的效果，往往中间有某个值效果最好，β为中间值时得到的红色曲线，比起绿线和黄线更好地平均了温度。

2.4 理解指数加权平均数

本视频中，我们进一步探讨算法的本质作用，回忆一下这个计算指数加权平均数的关键方程。

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9$的时候，得到的结果是红线，如果它更接近于 1，比如 0.98，结果就是绿线，如果β小一点，如果是 0.5，结果就是黄线。

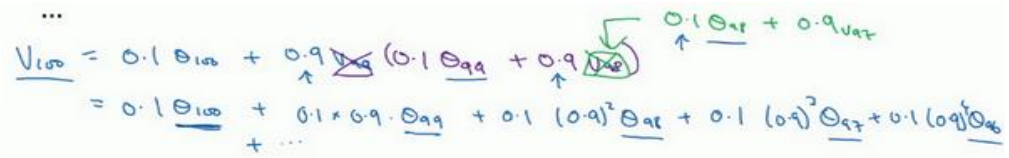


我们进一步地分析，来理解如何计算出每日温度的平均值。使$\beta = 0.9$，写下相应的几个公式，所以在执行的时候，t从 0 到 1 到 2 到 3，t的值在不断增加，为了更好地分析，我写的时候使得t的值不断减小，然后继续往下写。

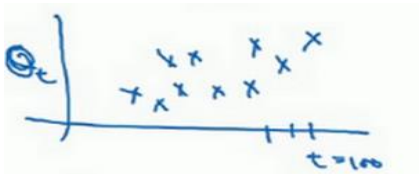
$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$
$$\cdots$$

理解v_{100}是什么？$v_{100} = 0.1\theta_{100} + 0.9v_{99}$。那么$v_{99}$是什么？我们就代入这个公式（$v_{99} = 0.1\theta_{99} + 0.9v_{98}$），所以：$v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98})$。那么$v_{98}$是什么？你可以用这个公式计算（$v_{98} = 0.1\theta_{98} + 0.9v_{97}$），把公式代进去，所以：$v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9v_{97}))$。以此类推，如果你把这些括号都展开，

$$v_{100} = 0.1\theta_{100} + 0.1 \times 0.9\theta_{99} + 0.1 \times (0.9)^2\theta_{98} + 0.1 \times (0.9)^3\theta_{97} + 0.1 \times (0.9)^4\theta_{96} + \cdots$$

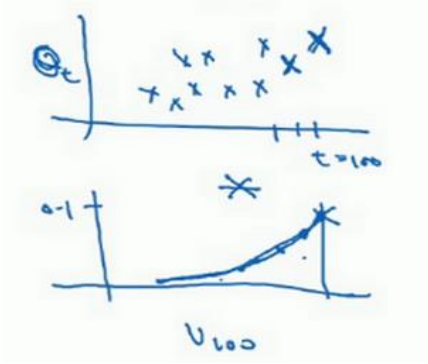


所以这是一个加和并平均，包括 100 号数据，99 号数据，97 号数据，…，1 号数据，从而得到 100 号数据，也就是当日温度。画图的一个办法是，假设我们有一些日期的温度，所以这是数据，这是t，所以 100 号数据有个数值，99 号数据有个数值，98 号数据等等，t为 100，99，98 等等，这就是数日的温度数值。



然后我们构建一个指数衰减函数，从 0.1 开始，到0.1×0.9，到$0.1 \times (0.9)^2$，以此类推，所以就有了这个指数衰减函数。

计算v_{100}是通过，把两个函数对应的元素相乘，然后求和，用这个数值 100 号数据值乘以 0.1，99 号数据值乘以 0.1 乘以$(0.9)^2$，这是第二项，以此类推，所以选取的是每日温度，将其与指数衰减函数相乘，然后求和，就得到了v_{100}。



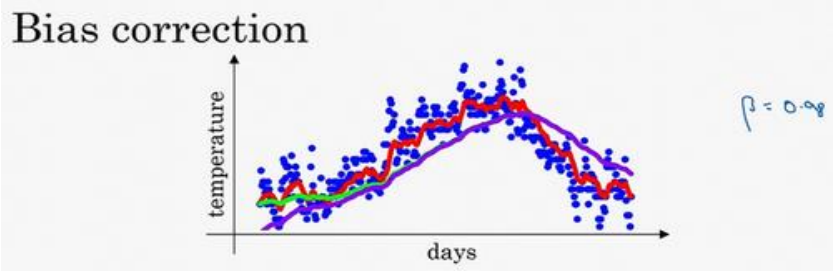
结果是，稍后我们详细讲解，不过所有的这些系数（$0.1 + 0.1 \times 0.9 + 0.1 \times (0.9)^2 + 0.1 \times (0.9)^3$ ...），相加起来为 1 或者逼近 1，我们称之为偏差修正，下一节会涉及。

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \beta v_0 + (1-\beta)\,\theta_1 \\
v_2 &= \beta v_1 + (1-\beta)\,\theta_2 \\
v_3 &= \beta v_2 + (1-\beta)\,\theta_3 \\
&\cdots
\end{aligned}
$$

指数加权平均数公式的好处之一在于，它占用极少内存，电脑内存中只占用一行数字而已，然后把最新数据代入公式，不断覆盖就可以了，正因为这个原因，它基本上只占用一行代码，计算指数加权平均数也只占用单行数字的存储和内存，当然它并不是最好的，也不是最精准的计算平均数的方法。如果你要计算移动窗，你直接算出过去 10 天的总和，过去 50 天的总和，除以 10 和 50 就好，如此往往会得到更好的估测。但缺点是，如果保存所有最近的温度数据，和过去 10 天的总和，必须占用更多的内存，执行更加复杂，计算成本也更加高昂。

2.5 指数加权平均的偏差修正

你学过了如何计算指数加权平均数，有一个技术名词叫做偏差修正（Bias correction in exponentially weighted averages），可以让平均数运算更加准确，来看看它是怎么运行的。

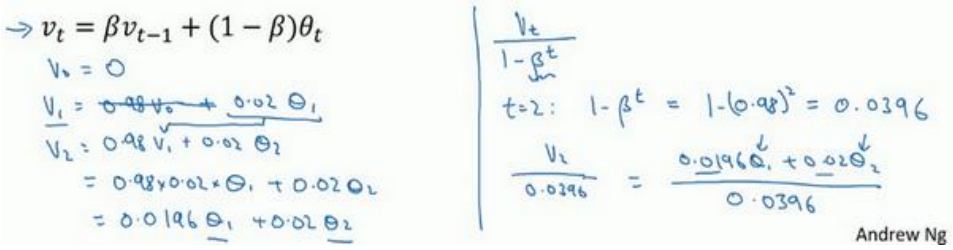


$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

在上一个视频中，这个（红色）曲线对应β的值为 0.9，这个（绿色）曲线对应的β=0.98，如果你执行写在这里的公式，在β等于 0.98 的时候，得到的并不是绿色曲线，而是紫色曲线，你可以注意到紫色曲线的起点较低，我们来看看怎么处理。

计算移动平均数的时候，初始化$v_0 = 0$，$v_1 = 0.98v_0 + 0.02\theta_1$，但是$v_0 = 0$，所以这部分没有了（$0.98v_0$），所以$v_1 = 0.02\theta_1$，所以如果一天温度是 40 华氏度，那么$v_1 = 0.02\theta_1 = 0.02 \times 40 = 8$，因此得到的值会小很多，所以第一天温度的估测不准。

$v_2 = 0.98v_1 + 0.02\theta_2$，如果代入$v_1$，然后相乘，所以$v_2 = 0.98 \times 0.02\theta_1 + 0.02\theta_2 = 0.0196\theta_1 + 0.02\theta_2$，假设$\theta_1$和$\theta_2$都是正数，计算后$v_2$要远小于$\theta_1$和$\theta_2$，所以$v_2$不能很好估测出这一年前两天的温度。



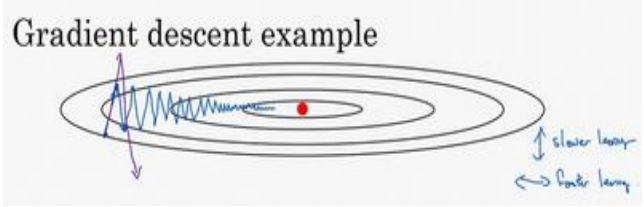
有个办法可以修改这一估测，让估测变得更好，更准确，特别是在估测初期，也就是不用v_t，而是用$\frac{v_t}{1-\beta^t}$，t 就是现在的天数。举个例子，当$t = 2$时，$1-\beta^t = 1-0.98^2 = 0.0396$，因此对第二天温度的估测变成了$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$，也就是$\theta_1$和$\theta_2$的加权平均数，并去除了偏差。你会发现随着$t$增加，$\beta^t$接近于 0，所以当$t$很大的时候，偏差修正几乎没有作用，因此当$t$较大的时候，紫线基本和绿线重合了。不过在开始学习阶段，才开始预测热身练习，偏差修正可以更好预测温度，偏差修正可以使结果从紫线变成绿线。

在机器学习中，在计算指数加权平均数时，很多人不在乎执行偏差修正，因为大部分人宁愿熬过初始时期，拿到具有偏差估测，然后继续计算下去。如果你关心初始时期的偏差，在刚开始计算指数加权移动平均数的时候，偏差修正能帮助你在早期获取更好的估测。

2.6 动量梯度下降法

还有一种算法叫做 **Momentum**，或者叫做动量梯度下降法（**Gradient descent with Momentum**），运行速度几乎总是快于标准的梯度下降算法，简而言之，基本的想法就是计算梯度的指数加权平均数，并利用该梯度更新权重。

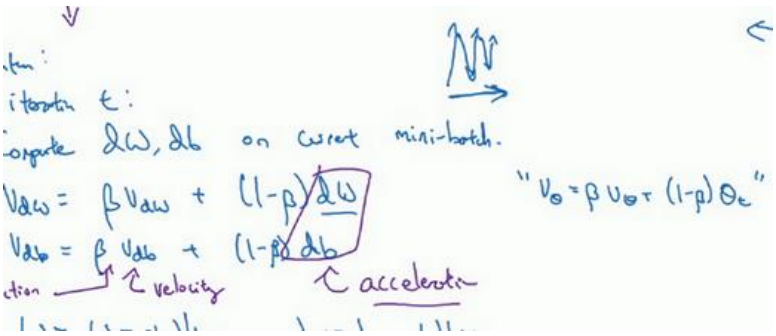
例如，如果要优化成本函数，函数形状如图，红点代表最小值的位置，假设我们从这里（蓝色点）开始梯度下降法，如果进行梯度下降法的一次迭代，无论是 **batch** 或 **mini-batch** 下降法，慢慢摆动到最小值，这种上下波动减慢了梯度下降法的速度，就无法使用更大的学习率，如果要用较大的学习率（紫色箭头），结果可能会偏离函数的范围，为了避免摆动过大，就要用一个较小的学习率。



另一个看待问题的角度是，在纵轴上，我们希望学习慢一点，因为不想要这些大的摆动，但是在横轴上，我们希望加快学习，希望快速从左向右移，移向最小值，移向红点。为了实现上述要求，我们可以使用动量梯度下降法。具体步骤是，在每次迭代中，确切来说在第t次迭代的过程中，你会计算微分dW，db，我会省略上标$[l]$，你用现有的 **mini-batch** 计算dW，db。如果你用 **batch** 梯度下降法，现在的 **mini-batch** 就是全部的 **batch**，对于 **batch** 梯度下降法的效果是一样的。如果现有的 **mini-batch** 就是整个训练集，效果也不错，你要做的是计算$v_{dW} = \beta v_{dW} + (1-\beta)dW$，这跟我们之前的计算相似，也就是$v = \beta v + (1-\beta)\theta_t$，$dW$的移动平均数，接着同样地计算$v_{db}$，$v_{db} = \beta v_{db} + (1-\beta)db$，然后重新赋值权重，$W := W - av_{dW}$，同样$b := b - av_{db}$，这样就可以减缓梯度下降的幅度。

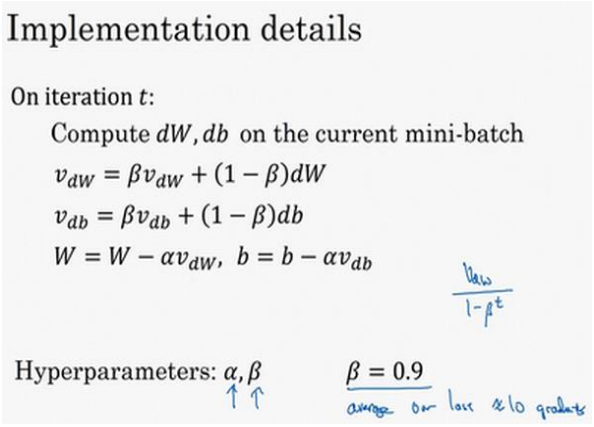
例如，在上几个导数中，你会发现这些纵轴上的摆动平均值接近于零，所以在纵轴方向，你希望放慢一点，平均过程中，正负数相互抵消，所以平均值接近于零。但在横轴方向，所有的微分都指向横轴方向，因此横轴方向的平均值仍然较大，因此用算法几次迭代后，你发现动量梯度下降法，最终纵轴方向的摆动变小了，横轴方向运动更快，因此你的算法走了一条更加直接的路径，在抵达最小值的路上减少了摆动。

动量梯度下降法的一个本质，这对有些人而不是所有人有效，就是如果你要最小化碗状函数，它们能够最小化碗状函数，这些微分项，想象它们为从山上往下滚的一个球，提供了加速度，**Momentum** 项相当于速度。



想象你有一个碗，你拿一个球，微分项给了这个球一个加速度，此时球正向山下滚，球因为加速度越滚越快，而因为β 稍小于 1，表现出一些摩擦力，所以球不会无限加速下去，所以不像梯度下降法，每一步都独立于之前的步骤，你的球可以向下滚，获得动量，可以从碗向下加速获得动量。

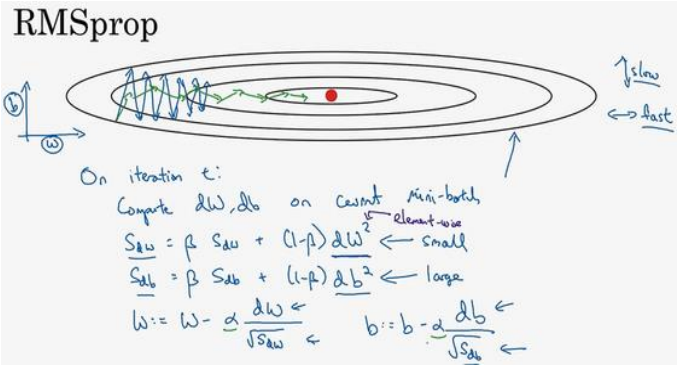
所以这里有两个超参数，学习率a以及参数β，β控制着指数加权平均数。β最常用的值是 0.9，我们之前平均了过去十天的温度，现在平均了前十次迭代的梯度。实际上β为 0.9，效果不错。那么关于偏差修正，所以你要拿v_{dW}和v_{db}除以$1-\beta^t$，实际上人们不这么做，因为 10 次迭代之后，因为移动平均已经过了初始阶段。实际中，在使用梯度下降法或动量梯度下降法时，人们不会受到偏差修正的困扰。当然v_{dW}初始值是 0，要注意到这是和dW拥有相同维数的零矩阵，也就是跟W拥有相同的维数，v_{db}的初始值也是向量零，所以和db拥有相同的维数，也就是和b是同一维数。



最后要说一点，如果你查阅了动量梯度下降法相关资料，你经常会看到一个被删除了的专业词汇，$1-\beta$被删除了，最后得到的是$v_{dW} = \beta v_{dW} + dW$。用紫色版本的结果就是，所以$v_{dW}$缩小了$1-\beta$倍，相当于乘以$\frac{1}{1-\beta}$，所以你要用梯度下降最新值的话，$a$要根据$\frac{1}{1-\beta}$相应变化。实际上，二者效果都不错，只会影响到学习率a的最佳值。我觉得这个公式用起来没有那么自然，因为有一个影响，如果你最后要调整超参数β，就会影响到v_{dW}和v_{db}，你也许还要修改学习率a，所以我更喜欢左边的公式，而不是删去了$1-\beta$的这个公式，所以我更倾向于使用左边的公式，也就是有$1-\beta$的这个公式，但是两个公式都将β设置为 0.9，是超参数的常见选择，只是在这两个公式中，学习率a的调整会有所不同。

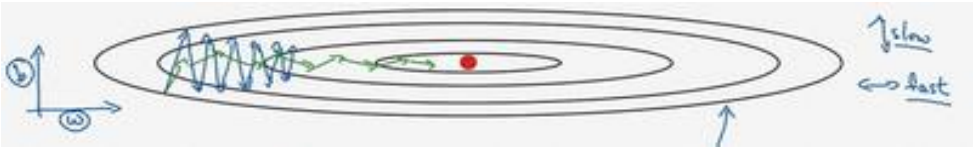
2.7 RMSprop

我们已学了动量（**Momentum**）可以加快梯度下降，现在学习 **RMSprop**（**root mean square prop**）算法，它也可以加速梯度下降，来看看它是如何运作的。回忆一下之前的例子，如果执行梯度下降，横轴方向推进的同时，纵轴方向会有大幅度摆动，假设纵轴代表参数b，横轴代表参数W，可能有W_1，W_2或者其它参数，为了便于理解，被称为b和W。所以我们想减缓b方向的学习，同时加快，至少不是减缓横轴方向的学习，**RMSprop** 算法可以实现这一点。



在第t次迭代中，该算法会照常计算当下 **mini-batch** 的微分dW，db，所以我会保留这个指数加权平均数，我们用到新符号S_{dW}，而不是v_{dW}，因此$S_{dW} = \beta S_{dW} + (1-\beta)dW^2$，澄清一下，这个平方是针对这一整个符号，相当于$(dW)^2$，这样做能够保留微分平方的加权平均数，同样$S_{db} = \beta S_{db} + (1-\beta)db^2$。

接着 **RMSprop** 会这样更新参数值，$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$，$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$。我们希望横轴方向（w）的学习速度快，故要除以一个较小的数S_{dW}，这样更新步幅才大；另一方面，我们希望减缓纵轴方向（b）的摆动，故要除以一个较大的数S_{db}，这样更新步幅才小，从而减缓摆动。本质上讲，由于函数的倾斜程度可知，在纵轴（b）方向上要大于在横轴（W）方向，故微分在垂直方向（db）要比水平方向（dW）大得多，所以$S_{db} < S_{dw}$。

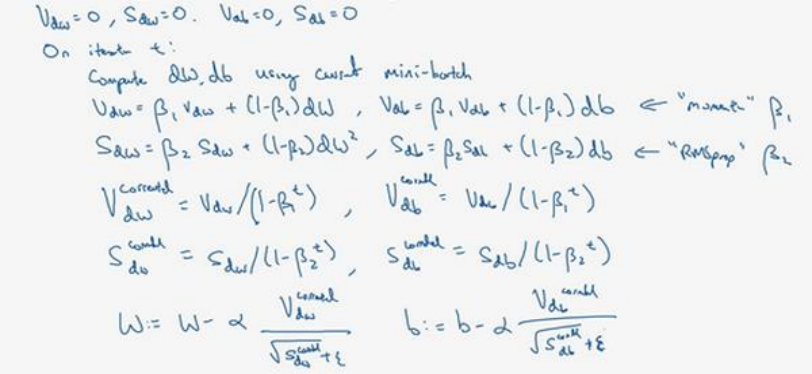


RMSprop 可以更新为绿色线，纵轴方向上摆动较小，而横轴方向继续推进。还有个影响就是用一个更大学习率α，然后加快学习，而无须在纵轴上垂直方向偏离。

最后要注意的是确保算法不会除以 0。如果S_{dW}的平方根趋近于 0 怎么办？得到的答案就非常大，为了确保数值稳定，在实际操练的时候，要在分母上加上一个很小的ε，ε是多少没关系，10^{-8}是个不错的选择，这只是保证数值能稳定一些，无论什么原因，都不会除以一个很小很小的数。所以 **RMSprop** 跟 **Momentum** 有很相似的一点，可以消除梯度下降中的摆动，包括 **mini-batch** 梯度下降，并允许使用更大的学习率α，从而加快算法学习速度。我们讲过了 **Momentum** 和 **RMSprop**，如果二者结合起来，就得到一个更好的优化算法。

2.8 Adam 优化算法

Adam（**Adaptive Moment Estimation**）优化算法是将 **Momentum** 和 **RMSprop** 结合在一起，是一种极其常用的学习算法，被证明能有效适用于不同神经网络，适用于广泛的结构，来看看如何使用 **Adam** 算法。

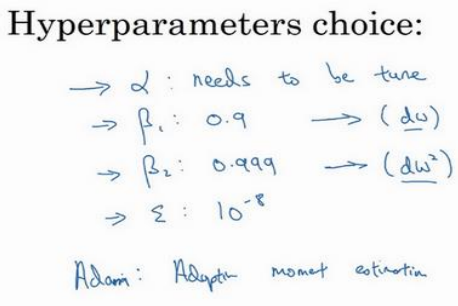


首先初始化$v_{dW}=0$，$S_{dW}=0$，$v_{db}=0$，$S_{db}=0$，在第t次迭代中计算微分，用当前的 **mini-batch** 计算dW，db，一般用 **mini-batch** 梯度下降法。然后计算 **Momentum** 指数加权平均数，$v_{dW}=\beta_1 v_{dW}+(1-\beta_1)dW$，同样 $v_{db}=\beta_1 v_{db}+(1-\beta_1)db$。

接着用 **RMSprop** 进行更新，$S_{dW}=\beta_2 S_{dW}+(1-\beta_2)(dW)^2$，再说一次，这里是对整个微分$dW$进行平方处理，$S_{db}=\beta_2 S_{db}+(1-\beta_2)(db)^2$。

相当于 **Momentum** 更新了超参数β_1，**RMSprop** 更新了超参数β_2。一般使用 **Adam** 算法的时候，要计算偏差修正，$v_{dW}^{\text{corrected}}$，修正也就是在偏差修正之后，$v_{dW}^{\text{corrected}}=\frac{v_{dW}}{1-\beta_1^t}$，同样$v_{db}^{\text{corrected}}=\frac{v_{db}}{1-\beta_1^t}$，$S$也使用偏差修正，也就是$S_{dW}^{\text{corrected}}=\frac{S_{dW}}{1-\beta_2^t}$，$S_{db}^{\text{corrected}}=\frac{S_{db}}{1-\beta_2^t}$。

最后更新权重，$W:=W-\alpha\frac{v_{dW}^{\text{corrected}}}{\sqrt{S_{dW}^{\text{corrected}}}+\varepsilon}$，$b:=b-\alpha\frac{v_{\text{db}}^{\text{corrected}}}{\sqrt{S_{\text{db}}^{\text{corrected}}}+\varepsilon}$。

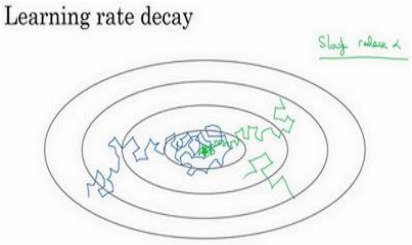


本算法中有很多超参数，超参数学习率α很重要，也经常需要调试可以尝试一系列值，然后看哪个有效。

- β_1：常用的缺省值为 0.9，这是 dW 的移动平均数，也就是dW的加权平均数。
- β_2：**Adam** 算法的发明者推荐使用 0.999，计算$(dW)^2$以及$(db)^2$的移动加权平均值。
- ε：其实没那么重要，**Adam** 作者建议ε为10^{-8}，一般不用调整。

2.9 学习率衰减

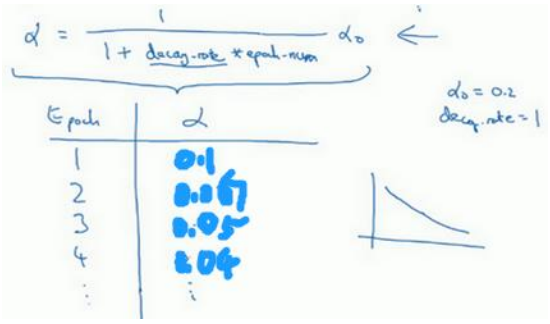
假设要使用 **mini-batch** 梯度下降法，**mini-batch** 数量不大，大概 64 或者 128 个样本，在迭代过程中会有噪音（蓝色线），下降朝向这里的最小值，但是不会精确地收敛，所以算法最后在附近摆动，并不会真正收敛，因为α是固定值，不同的 **mini-batch** 中有噪音。

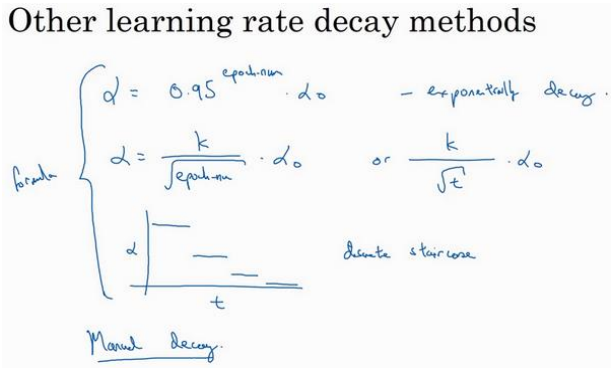


但要慢慢减少学习率α的话，在初期的时候，α学习率还较大，学习还是相对较快，但随着α变小，步伐也会变慢变小，所以最后曲线（绿色线）会在最小值附近的一小块区域里摆动，而不是在训练过程中，大幅度在最小值附近摆动。

所以慢慢减少α的本质在于，在学习初期，能承受较大的步伐，但当开始收敛的时候，小一些的学习率能让步伐小一些。可以这样做到学习率衰减，记得一代要遍历一次数据，如果你有以下这样的训练集:

拆分成不同的 **mini-batch**，第一次遍历训练集叫做第一代。第二次就是第二代，依此类推，可以将α学习率设为$\alpha = \frac{1}{1+decay_rate * \text{epoch_num}}\alpha_0$（**decay-rate** 称为衰减率，**epoch-num** 为代数，α_0为初始学习率），注意这个衰减率是另一个需要调整的超参数。

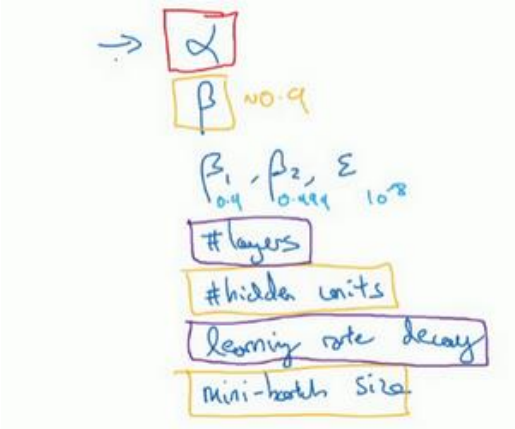




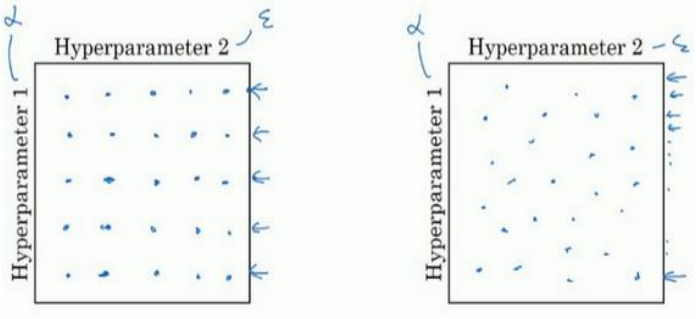
第三周 超参数调试、Batch 正则化和程序框架

3.1 调试处理

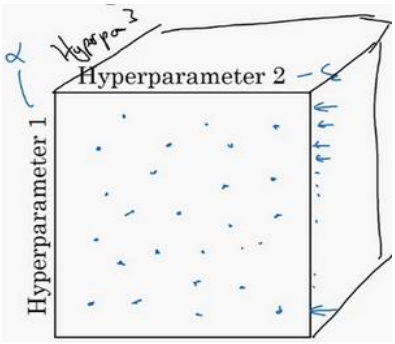
对于超参数而言，如何找到一套好的设定呢？结果证实一些超参数比其它的更为重要，最为广泛的学习应用是α，学习率是需要调试的最重要的超参数。其次，还有一些参数需要调试，例如 **Momentum** 参数β，0.9 就是个很好的默认值；还会调试 **mini-batch** 的大小，以确保最优算法运行有效；还会经常调试隐藏单元。最后是其他因素，层数有时会产生很大的影响，学习率衰减也是如此。当应用 **Adam** 算法时，事实上，不怎么调试β_1，β_2和ε，分别设置为 0.9，0.999 和10^{-8}即可。注意，以上分类并不是绝对严格的标准，知识经验之谈。



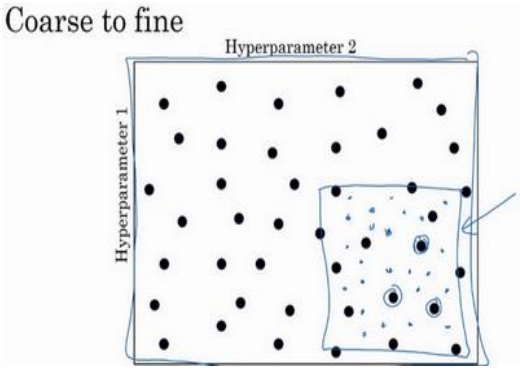
在深度学习领域，我们常做的，我推荐你采用下面的做法，随机选择点，所以你可以选择同等数量的点，对吗？我们可以以此来判断哪些超参数更为重要。25 个点，接着，用这些随机取的点试验超参数的效果，来判断哪些超参数的确要比其它的更重要。举个例子，假设超参数 1 是α（学习速率），假设超参数 2 是 **Adam** 算法中，分母中的ε。在这种情况下，α的取值很重要，而ε取值则无关紧要。如果在网格中取点，接着试验了α的 5 个取值，那么无论ε取何值，结果基本上都是一样的。所以，调试学习率α更为重要。



我已经解释了两个参数的情况，实践中搜索的超参数可能不止两个。假如有三个超参数，这时搜索的不是一个方格，而是一个立方体，超参数 3 代表第三维。

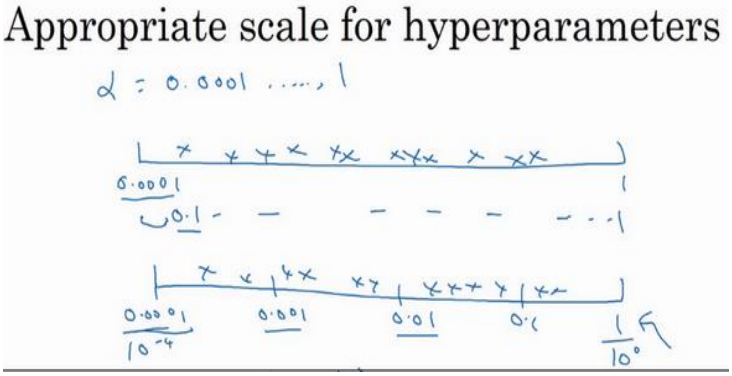


当给超参数取值时，另一个惯例是采用由粗糙到精细的策略。比如在二维的那个例子中，首先大步幅取值，也许会发现效果最好的某个点，或者效果更优，然后我们在效果更好的附近区域（小蓝色方框内）进行精细取值，小步幅地在其中更密集得取值或随机取值。



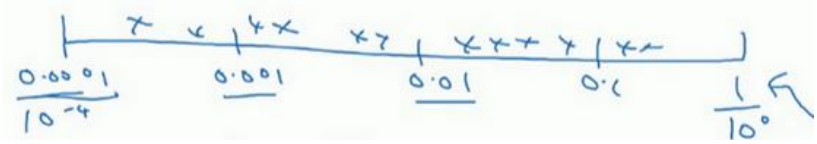
3.2 为超参数选择合适的范围

之前我们已经看到了在超参数范围中，随机取值可以提升搜索效率。但随机取值并不是在有效范围内的随机均匀取值，而是选择合适的标尺，用于探究这些超参数，这很重要。

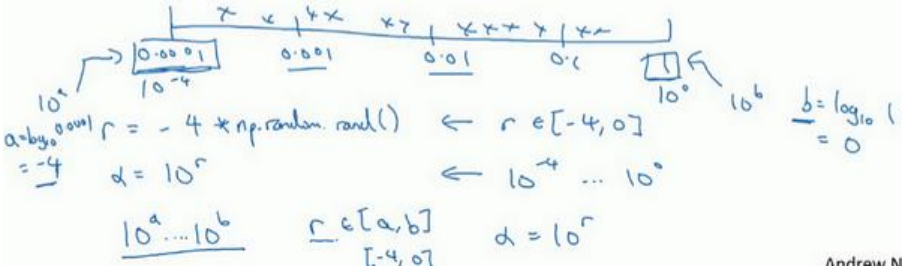


假设搜索超参数a，其值最小是 0.0001 或最大是 1。如果画一条从 0.0001 到 1 的数轴，沿其随机均匀取值，那 90%的数值将会落在 0.1 到 1 之间，结果就是在 0.1 到 1 之间，应用了 90%的资源，而在 0.0001 到 0.1 之间，只有 10%的搜索资源，这看上去不太对。

反而，用对数标尺搜索超参数的方式会更合理，因此这里不使用线性轴，分别依次取 0.0001，0.001，0.01，0.1，1，在对数轴上均匀随机取点，这样，在 0.0001 到 0.001 之间，就会有更多的搜索资源可用，还有在 0.001 到 0.01 之间等等。在 **Python** 中，你可以这样做，使 r=-4*np.random.rand()，然后a随机取值，$a = 10^r$，所以，第一行可以得出$r \in [4,0]$，那么$a \in [10^{-4}, 10^0]$，所以最左边的数字是10^{-4}，最右边是10^0。



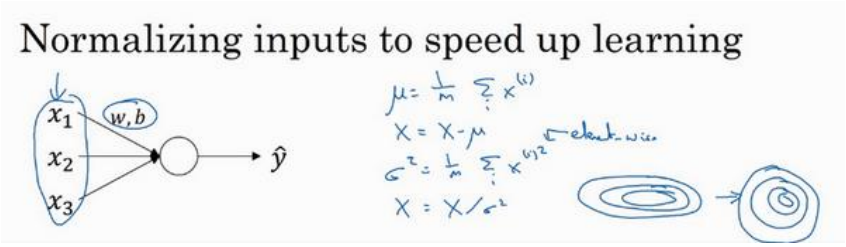
更常见的情况是，如果你在10^a和10^b之间取值，在此例中，这是10^a（0.0001），你可以通过0.0001算出a的值，即-4，在右边的值是10^b，你可以算出b的值1，即 0。你要做的就是在$[a, b]$区间随机均匀地给r取值，这个例子中$r \in [-4,0]$，然后你可以设置a的值，基于随机取样的超参数$a = 10^r$。



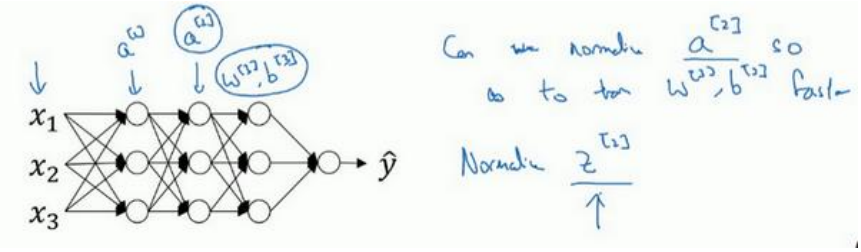
总结一下，在对数坐标下取值，取最小值的对数就得到a的值，取最大值的对数就得到b值，所以现在你在对数轴上的10^a到10^b区间取值，在a，b间随意均匀的选取r值，将超参数设置为10^r，这就是在对数轴上取值的过程。

3.3 归一化网络的激活函数

在深度学习兴起后，最重要的一个思想是 **Batch** 归一化，由 **Sergey Ioffe** 和 **Christian Szegedy** 两位研究者创造。**Batch** 归一化会使参数搜索问题变得很容易，使神经网络对超参数的选择更加稳定，超参数范围会更加庞大，工作效果也很好，也会使训练更加容易，甚至是深层网络。让我们来看看 **Batch** 归一化是怎么起作用的吧。



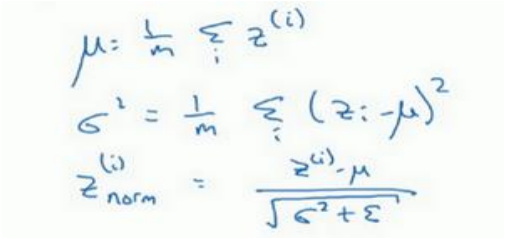
当训练一个模型，比如 **logistic** 回归时，归一化输入特征可以加快学习过程。计算平均值，从训练集中减去平均值，计算方差，接着根据方差归一化数据集，这是如何把学习问题的轮廓，从很长的东西，变成更圆的东西，更易于算法优化。所以这是有效的，对 **logistic** 回归和神经网络的归一化输入特征值而言。



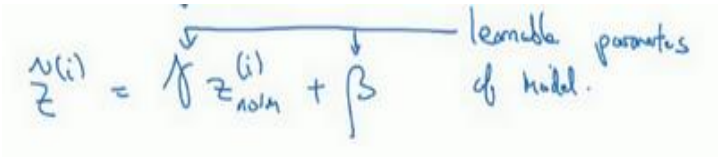
那么更深的模型呢？不仅输入特征值x，而且有激活值$a^{[1]}$，$a^{[2]}$等等。如果想训练这些参数，比如$w^{[3]}$，$b^{[3]}$，那归一化$a^{[2]}$的平均值和方差岂不是很好？以便使$w^{[3]}$，$b^{[3]}$的训练更有效率。在 **logistic** 回归的中，我们看到了如何归一化x_1，x_2，x_3，会更有效的训练w和b。

所以问题来了，对任何一个隐藏层而言，我们能否归一化a值，在此例中，比如说$a^{[2]}$的值，以更快的速度训练$w^{[3]}$，$b^{[3]}$，因为$a^{[2]}$是下一层的输入值，所以就会影响$w^{[3]}$，$b^{[3]}$的训练。尽管严格来说，我们真正归一化的不是$a^{[2]}$，而是$z^{[2]}$，深度学习文献中有一些争论，关于在激活函数之前是否应该将值$z^{[2]}$归一化，或是否应该在应用激活函数$a^{[2]}$后再规范值。实践中，经常做的是归一化$z^{[2]}$，下面就是 **Batch** 归一化的使用方法。

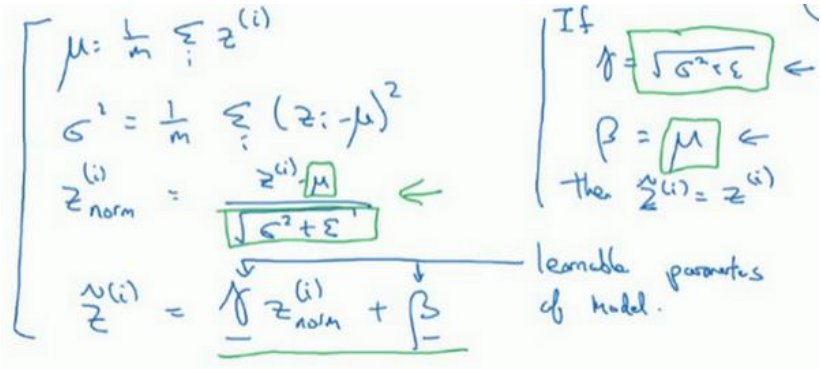
在神经网络中，已知一些中间值，假设你有一些隐藏单元值，从$z^{(1)}$到$z^{(m)}$，这些来源于隐藏层，计算方法如下，减去均值再除以标准偏差，为了使数值稳定，通常将ε作为分母，以防$\sigma = 0$的情况。



现在我们已把这些z值标准化，化为含平均值 0 和标准单位方差，所以z的每一个分量都含有平均值 0 和方差 1，但我们不想让隐藏单元总是含有平均值 0 和方差 1，也许隐藏单元有了不同的分布会有意义，所以我们所要做的就是计算$\tilde{z}^{(i)}$，$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$，这里$\gamma$和$\beta$是模型的学习参数，所以我们使用梯度下降或一些其它类似梯度下降的算法，比如 **Momentum** 或者 **Adam**，你会更新γ和β，正如更新神经网络的权重一样。



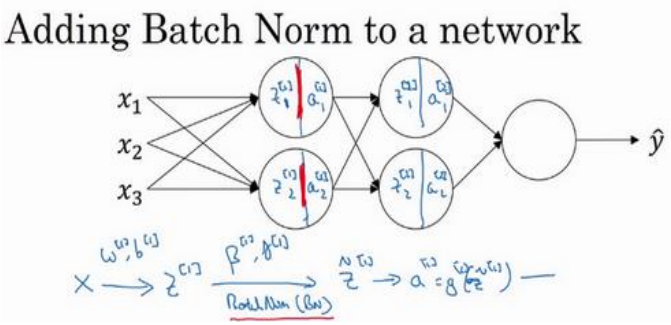
请注意γ和β的作用是，可以随意设置$\tilde{z}^{(i)}$的平均值，事实上，如果$\gamma = \sqrt{\sigma^2 + \varepsilon}$（$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$中的分母），$\beta$等于$\mu$（$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$中的$\mu$），那么$\gamma z_{\text{norm}}^{(i)} + \beta$的作用在于，它会精确转化这个方程，如果这些成立（$\gamma = \sqrt{\sigma^2 + \varepsilon}, \beta = \mu$），那么$\tilde{z}^{(i)} = z^{(i)}$。通过对$\gamma$和$\beta$合理设定，规范化过程，即这四个等式，从根本来说，只是计算恒等函数，通过赋予γ和β其它值，可以使你构造含其它平均值和方差的隐藏单元值。



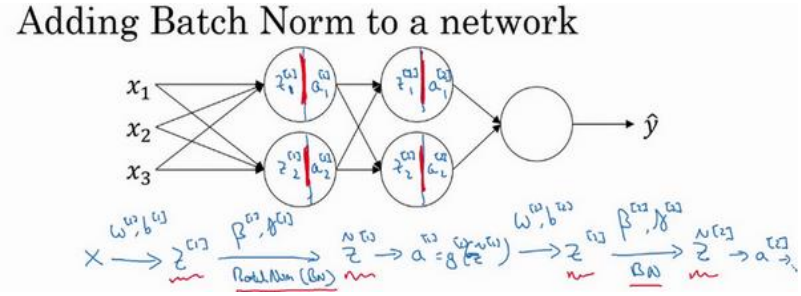
在网络匹配这个单元的方式，之前可能是用$z^{(1)}$，$z^{(2)}$等等，现在则会用$\tilde{z}^{(i)}$取代$z^{(i)}$，方便神经网络中的后续计算。如果想放回$[l]$，以清楚的表明它位于哪层，可以把它放这。

3.4 将 Batch Norm 拟合进神经网络

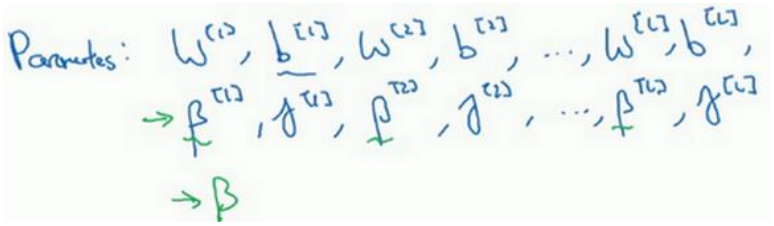
你已经看到那些等式，它可以在单一隐藏层进行 **Batch** 归一化，接下来，让我们看看它是怎样在深度网络训练中拟合的吧。



假设有一个这样的神经网络，可以认为每个单元负责计算两件事。第一，它先计算z，然后应用其到激活函数中再计算a。同样的，对于下一层而言，那就是$z_1^{[2]}$和$a_1^{[2]}$等。如果没有应用 **Batch** 归一化，把输入X拟合到第一隐藏层，然后首先计算$z^{[1]}$，这是由$w^{[1]}$和$b^{[1]}$两个参数控制的。接着，通常而言，你会把$z^{[1]}$拟合到激活函数以计算$a^{[1]}$。但 **Batch** 归一化的做法是将$z^{[1]}$值进行 **Batch** 归一化，简称 **BN**，此过程将由$\beta^{[1]}$和$\gamma^{[1]}$两参数控制，这一操作会给一个规范化的$z^{[1]}$值（$\tilde{z}^{[1]}$），然后将其输入激活函数中得到$a^{[1]}$，即$a^{[1]} = g^{[1]}(\tilde{z}^{[l]})$。



需要强调的是 **Batch** 归一化是发生在计算z和a之间的。与其应用没有归一化的z值，不如用归一过的$\tilde{z}$，这是第一层（$\tilde{z}^{[1]}$），第二层同理。所以网络的参数就会是$w^{[1]}$，$b^{[1]}$，$w^{[2]}$和$b^{[2]}$等等，现在我们将另一些参数加入到此新网络中$\beta^{[1]}$，$\beta^{[2]}$，$\gamma^{[1]}$，$\gamma^{[2]}$等等。这里的这些β（$\beta^{[1]}$，$\beta^{[2]}$等等）和超参数β没有任何关系，后者是用于 **Momentum** 或计算各个指数的加权平均值。**Adam** 论文的作者，在论文里用β代表超参数。**Batch** 归一化论文的作者，则使用β代表此参数（$\beta^{[1]}$，$\beta^{[2]}$等等），但这是两个完全不同的β。

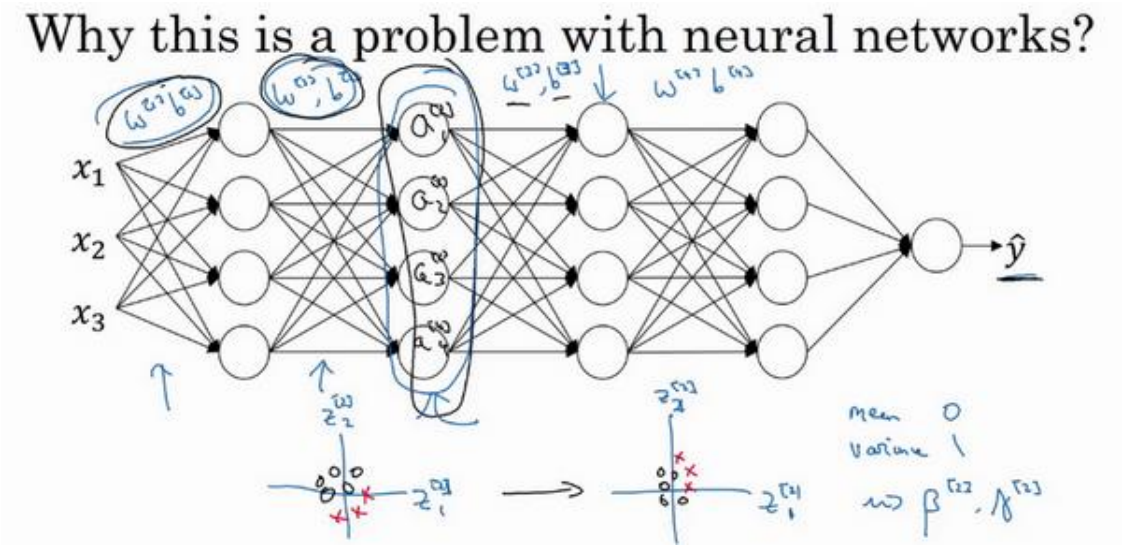


现在，以防 **Batch** 归一化仍然看起来有些神秘，尤其是你还不清楚为什么其能如此显著的加速训练，我们进入下一个视频，详细讨论 **Batch** 归一化为何效果如此显著。

3.5 Batch Norm 为什么奏效？

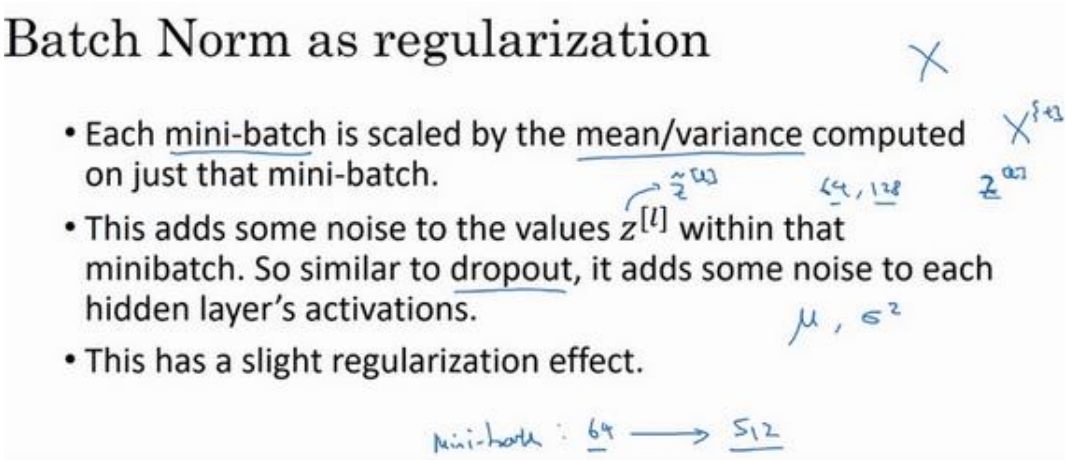
为什么 **Batch** 归一化会起作用呢？一个原因是，已经看到如何归一化输入特征值x，使其均值为 0，方差 1，它又是怎样加速学习的，有一些从 0 到 1 而不是从 1 到 1000 的特征值，通过归一化所有的输入特征值x，以获得类似范围的值，可以加速学习。所以 **Batch** 归一化起作用的原因就是，它在做类似的工作，但不仅仅对于这里的输入值，还有隐藏单元的值，这只是 **Batch** 归一化作用的冰山一角，还有些深层的原理，它会有助于对 **Batch** 归一化的作用有更深的理解。**Batch** 归一化有效的第二个原因是，它可以使权重比网络更滞后或更深层，比如第 10 层的权重更能经受得住变化，相比于神经网络中前层的权重，比如第 1 层。

Batch 归一化减少了输入值改变的问题，它的确使这些值变得更稳定，神经网络的之后层就会有更坚实的基础。即使使输入分布改变了一些，它会改变得更少。它做的是当前层保持学习，当改变时，迫使后层适应的程度减小了，你可以这样想，它减弱了前层参数的作用与后层参数的作用之间的联系，它使得网络每层都可以自己学习，稍稍独立于其它层，这有助于加速整个网络的学习。



所以，希望这能带给你更好的直觉，重点是 **Batch** 归一化的意思是，尤其从神经网络后层之一的角度而言，前层不会左右移动的那么多，因为它们被同样的均值和方差所限制，所以，这会使得后层的学习工作变得更容易些。

Batch 归一化还有一个作用，它有轻微的正则化效果，**Batch** 归一化中非直观的一件事是，每个 **mini-batch**，我会说 **mini-batch**$X^{\{t\}}$的值为$z^{[l]}$，$z^{[l]}$，在 **mini-batch** 计算中，由均值和方差缩放的，因为在 **mini-batch** 上计算的均值和方差，而不是在整个数据集上，均值和方差有一些小的噪声，因为它只在你的 **mini-batch** 上计算，比如 64 或 128 或 256 或更大的训练例子。因为均值和方差有一点小噪音，因为它只是由一小部分数据估计得出的。缩放过程从$z^{[l]}$到$\tilde{z}^{[l]}$，过程也有一些噪音，因为它是用有些噪音的均值和方差计算得出的。

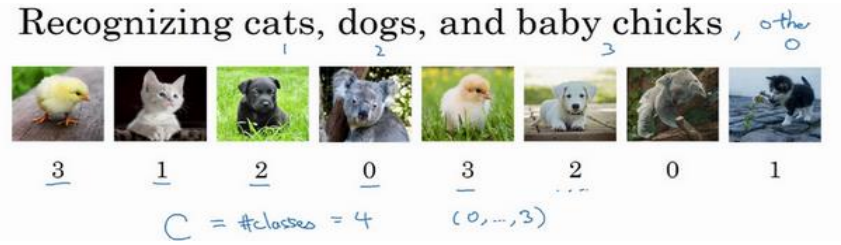


所以和 **dropout** 相似，它往每个隐藏层的激活值上增加了噪音，**dropout** 有增加噪音的方式，它使一个隐藏的单元，以一定的概率乘以 0，以一定的概率乘以 1，所以你的 **dropout** 含几重噪音，因为它乘以 0 或 1。

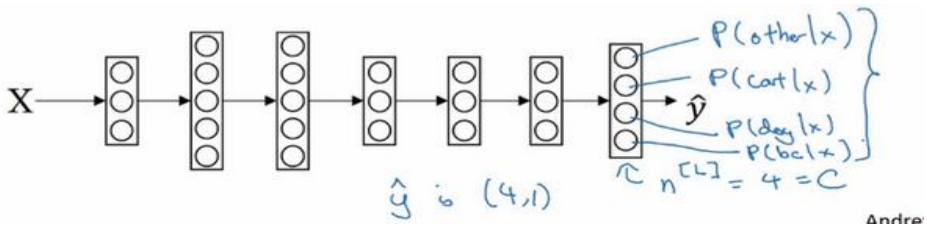
对比而言，**Batch** 归一化含几重噪音，因为标准偏差的缩放和减去均值带来的额外噪音。这里的均值和标准差的估计值也是有噪音的，所以类似于 **dropout**，**Batch** 归一化有轻微的正则化效果，因为给隐藏单元添加了噪音，这迫使后部单元不过分依赖任何一个隐藏单元，类似于 **dropout**，它给隐藏层增加了噪音，因此有轻微的正则化效果。因为添加的噪音很微小，所以并不是巨大的正则化效果，你可以将 **Batch** 归一化和 **dropout** 一起使用，如果你想得到 **dropout** 更强大的正则化效果。

3.6 Softmax 回归

到目前为止，我们讲到过的分类的例子都使用了二分分类，这种分类只有两种可能的标记 0 或 1，是一只猫或者不是一只猫，如果我们有多种可能的类型的话呢？有一种 **logistic** 回归的一般形式，叫做 **Softmax** 回归，识别多种分类，不只是识别两个分类。



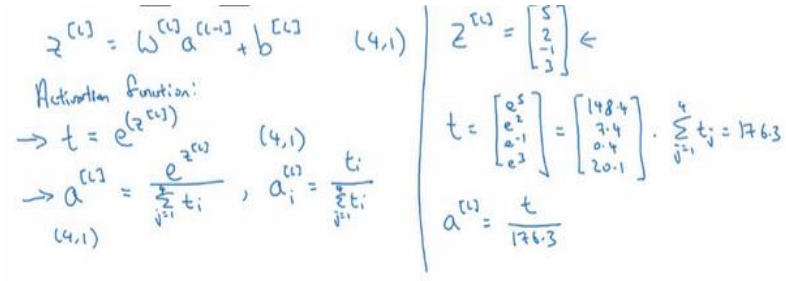
假设你不单需要识别猫，而是想识别猫，狗和小鸡，我把猫加做类 1，狗为类 2，小鸡是类 3，如果不属于以上任何一类，就分到“其它”这一类，我把它叫做类 0。我们将会用符号表示，我会用大写的C来表示输入类别总个数，在这个例子中，我们有 4 种可能的类别，指示类别的数字就是从 0 到$C-1$，换句话说就是 0、1、2、3。



在这个例子中，我们将建立一个神经网络，其输出层有 4 个，或者说C个输出单元，因此n，即输出层也就是L层的单元数量，等于 4，或者一般而言等于C。我们想要输出层单元的数字告诉我们这 4 种类型中每个的概率有多大，

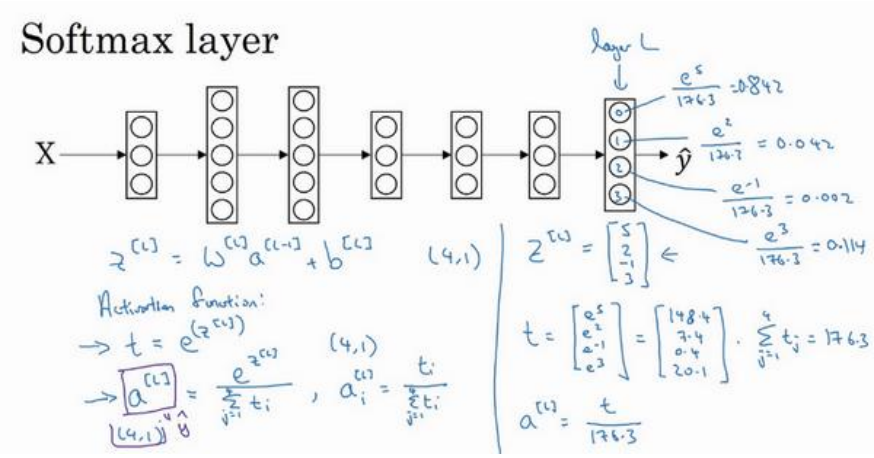
- 在输入X的情况下，第一个节点(最后输出的第 1 个方格+圆圈)输出的是“其他类”的概率。
- 在输入X的情况下，第二个节点（最后输出的第 2 个方格+圆圈）会输出猫的概率。
- 在输入X的情况下，第三个节点（最后输出的第 3 个方格+圆圈）会输出狗的概率。
- 在输入X的情况下，第四个节点（最后输出的第 4 个方格+圆圈）会输出小鸡的概率。

因此这里的$\hat{y}$将是一个4×1维向量，因为它必须输出四个数字，给你这四种概率，因为它们加起来应该等于 1，输出中的四个数字加起来应该等于 1。让网络做到这一点的标准模型要用到 **Softmax** 层，以及输出层来生成输出，公式如下：



在神经网络的最后一层，像往常一样计算各层的线性部分，$z^{[l]}$这是最后一层的z变量，记住这是大写L层，计算方法是$z^{[l]} = W^{[l]}a^{[L-1]} + b^{[l]}$，算出了$z$之后，需要应用 **Softmax** 激活函数，这个激活函数对于 **Softmax** 层而言有些不同，它是这样的，$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^{4} t_i}$，换句话说，$a^{[l]}$也是一个 4×1 维向量，而这个四维向量的第$i$个元素。

我们来看一个例子，详细解释，假设你算出了$z^{[l]}$，$z^{[l]}$是一个四维向量，假设为$z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$，我们要做的就是用这个元素取幂方法来计算$t$，所以$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$，$t$得到向量$a^{[l]}$就只需要将这些项目归一化，使总和为 1。如果你把$t$的元素都加起来，把这四个数字加起来，得到 176.3，最终$a^{[l]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$)。



3.9 训练一个 Softmax 分类器

上一个视频中我们学习了 **Softmax** 层和 **Softmax** 激活函数，在这个视频中，你将更深入地了解 **Softmax** 分类，并学习如何训练一个使用了 **Softmax** 层的模型。

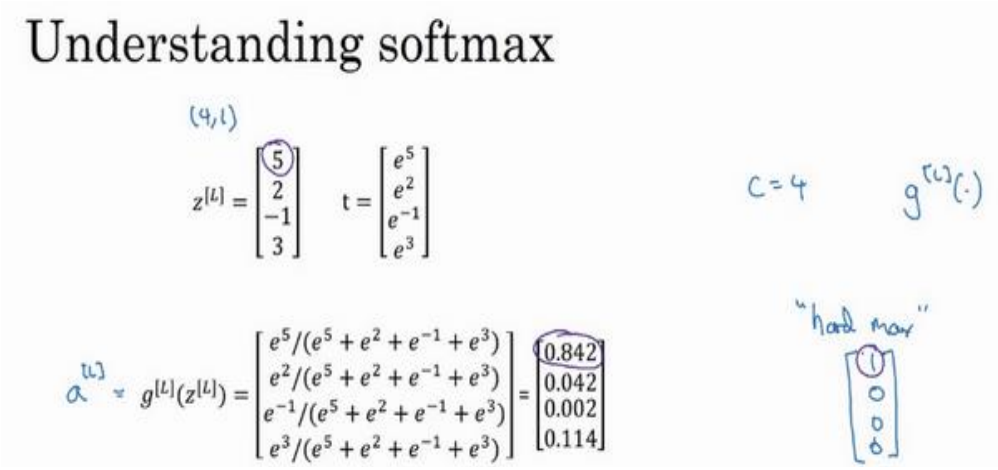
$$(4,1)$$

$$z^{[L]}=\begin{bmatrix}5\\2\\-1\\3\end{bmatrix} \qquad t=\begin{bmatrix}e^{5}\\e^{2}\\e^{-1}\\e^{3}\end{bmatrix}$$

回忆一下我们之前举的的例子，输出层计算出的$z^{[l]}$如下，$z^{[l]}=\begin{bmatrix}5\\2\\-1\\3\end{bmatrix}$我们有四个分类$C=4$，$z^{[l]}$可以是 4×1 维向量，我们计算了临时变量$t$，$t=\begin{bmatrix}e^{5}\\e^{2}\\e^{-1}\\e^{3}\end{bmatrix}$，对元素进行幂运算，最后，如果输出层的激活函数$g^{[L]}()$是 **Softmax** 激活函数，那么输出就会是这样的：

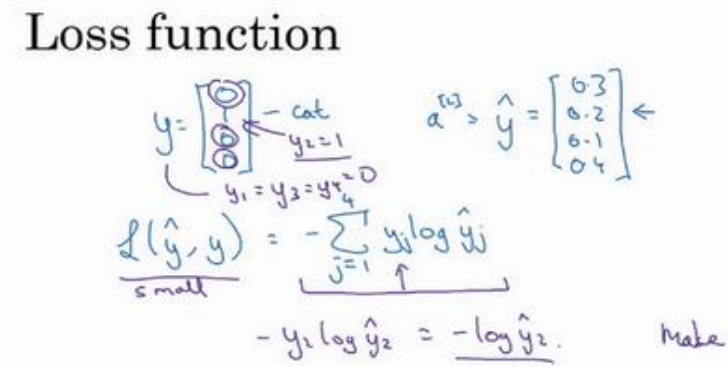
$$g^{[L]}(z^{[L]})=\begin{bmatrix}e^{5}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{2}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{-1}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{3}/(e^{5}+e^{2}+e^{-1}+e^{3})\end{bmatrix}=\begin{bmatrix}0.842\\0.042\\0.002\\0.114\end{bmatrix}$$

简单来说就是用临时变量t将它归一化，使总和为 1，于是这就变成了$a^{[L]}$，注意到向量z中，最大的元素是 5，而最大的概率也就是第一种概率。



Softmax 的来源是与 **hardmax** 对比，**hardmax** 会把向量z变成这个向量$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$，**hardmax** 函数会观察$z$的元素，然后在$z$中最大元素的位置放上 1，其它位置放上 0，这是一个 **hard max**，也就是最大的元素的输出为 1，其它的输出都为 0。与之相反，**Softmax** 所做的从z到这些概率的映射更为温和，这就是 **softmax** 背后所包含的想法，与 **hardmax** 正好相反。

接下来我们来看怎样训练带有 **Softmax** 输出层的神经网络，具体而言，我们先定义训练神经网络使会用到的损失函数。举个例子，我们来看看训练集中某个样本的目标输出，真实标签是$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$，这表示这是一张猫的图片，因为它属于类 1，现在假设神经网络输出的是$\hat{y}$，$\hat{y}$是一个包括总和为 1 的概率的向量，$y=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$，你可以看到总和为 1，这就是$a^{[l]}$，$a^{[l]}=y=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$。对于这个样本神经网络的表现不佳，这实际上是一只猫，但却只分配到 20%是猫的概率，所以在本例中表现不佳。

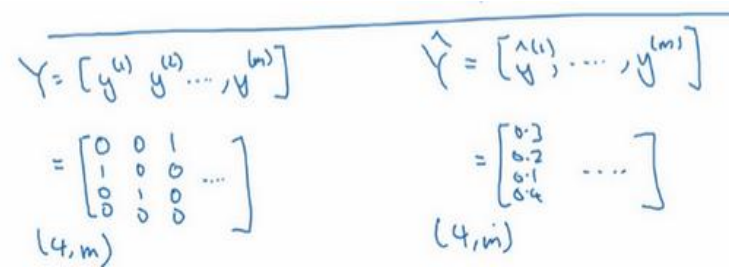


那么你想用什么损失函数来训练这个神经网络？在 **Softmax** 分类中，我们一般用到的损失函数是$L(\hat{y},y)=-\sum_{j=1}^{4}y_j log\,\hat{y}_j$。注意在这个样本中$y_1=y_3=y_4=0$，只有$y_2=1$，如果你看这个求和，最后只剩下$-y_2 log\,\hat{y}_2$，因为当你按照下标$j$全部加起来，所有的项都为 0，除了$j=2$时，又因为$y_2=1$，所以它就等于$-log\,\hat{y}_2$。$L(\hat{y},y)=-\sum_{j=1}^{4}y_j\log\hat{y}_j=-y_2\,log\,\hat{y}_2=-log\,\hat{y}_2$

这就意味着，如果学习算法试图将它变小，因为梯度下降法是用来减少训练集的损失的，要使它变小的唯一方式就是使$-\log\hat{y}_2$变小，就需要使$\hat{y}_2$尽可能大，因为这些是概率，所以不可能比 1 大，因为在这个例子中x是猫的图片，就需要这项输出的概率尽可能地大（$y=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$中第二个元素）。

这是单个训练样本的损失，整个训练集的损失J又如何呢？也就是整个训练集损失的总和，把你的训练算法对所有训练样本的预测都加起来，因此要做的就是用梯度下降法，使这里的损失最小化。

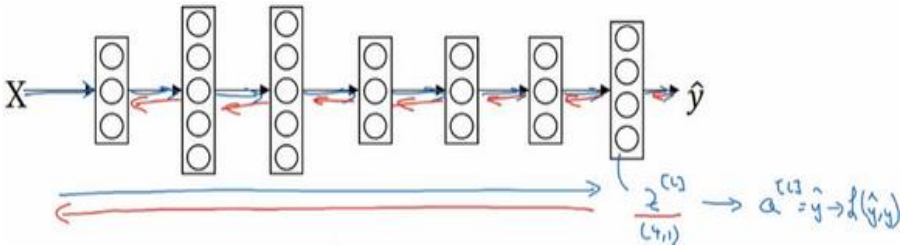
$$J(w^{[1]},b^{[1]},\dots\dots)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$$



最后还有一个细节，因为$C=4$，y是一个 4×1 向量，$\hat{y}$也是一个 4×1 向量，如果实现向量化，矩阵大写Y就是$[y^{(1)}y^{(2)}\dots\dots y^{(m)}]$，例如矩阵$Y=\begin{bmatrix}0&0&1&\dots\\1&0&0&\dots\\0&1&0&\dots\\0&0&0&\dots\end{bmatrix}$，那么矩阵$Y$最终就是一个$4\times m$维矩阵。类似的，$\hat{Y}=[\hat{y}^{(1)}\,\hat{y}^{(2)}\dots\dots\,\hat{y}^{(m)}]$，这个其实就是$\hat{y}^{(1)}$（$a^{[l](1)}=y^{(1)}=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$），或第一个训练样本的输出，那么$\hat{Y}=\begin{bmatrix}0.3&\dots\\0.2&\dots\\0.1&\dots\\0.4&\dots\end{bmatrix}$，$\hat{Y}$本身也是一个$4\times m$维矩阵。

最后我们来看一下，在有 **Softmax** 输出层时如何实现梯度下降法，这个输出层会计算$z^{[l]}$，它是$C\times 1$维的，在这个例子中是 4×1，然后用 **Softmax** 激活函数来得到$a^{[l]}$，然后由此计算出损失。我们已经讲了如何实现神经网络前向传播的步骤，来得到这些输出，并计算损失，那么反向传播步骤或者梯度下降法又如何呢？其实初始化反向传播所需要的关键步骤或者说关键方程是这个表达式$dz^{[l]}=\hat{y}-y$，可以用$\hat{y}$这个 4×1 向量减去y这个 4×1 向量，4 个分类时，在一般情况下就是$C\times 1$，这符合我们对dz的一般定义，这是对$z^{[l]}$损失函数的偏导数（$dz^{[l]}=\frac{\partial J}{\partial z^{[l]}}$），如果精通微积分就可以自己推导，或者说如果你精通微积分，可以试着自己推导，但如果需要从零开始使用这个公式，它也一样有用。

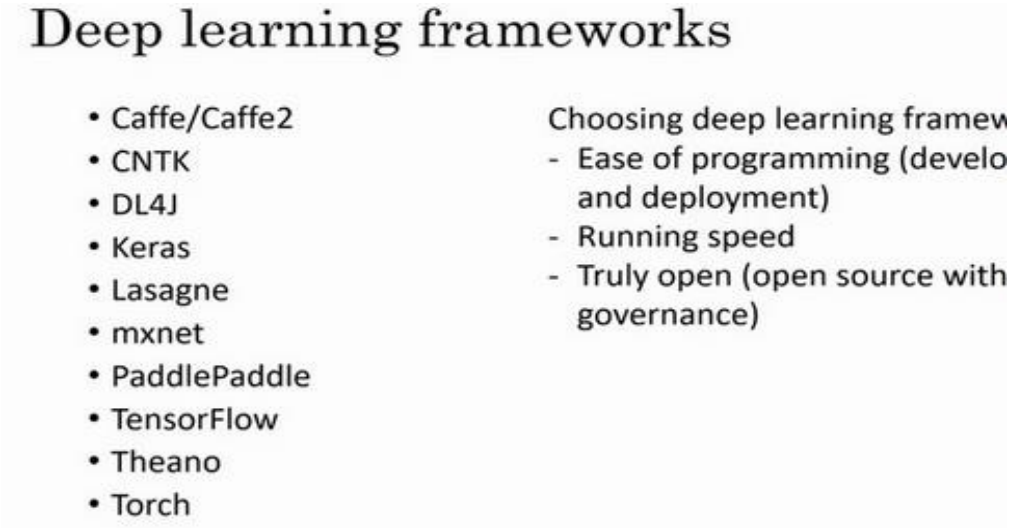
有了这个，就可以计算$dz^{[l]}$，然后开始反向传播的过程，计算整个神经网络中所需要的所有导数。



我们将开始使用一种深度学习编程框架，对于这些编程框架，通常只需要专注于把前向传播做对，只要你将它指明为编程框架，前向传播，它自己会弄明白怎样反向传播，会实现反向传播，所以这个表达式值得牢记（$dz^{[l]}=\hat{y}-y$），如果需要从头开始，实现 **Softmax** 回归或者 **Softmax** 分类，但其实在这周的初级练习中不会用到它，因为编程框架会帮你搞定导数计算。

3.10 深度学习框架

现在有很多好的深度学习软件框架，可以实现这些模型。类比一下，我们知道如何做矩阵乘法，还知道如何编程实现两个矩阵相乘，但是在很大的应用时，很可能不想用自己的矩阵乘法函数，而是想要访问一个数值线性代数库，它会更高效，但如果明白两个矩阵相乘是怎么回事还是挺有用的，那就让我们来看下有哪些框架。



一个重要的标准就是便于编程，这既包括神经网络的开发和迭代，还包括为产品进行配置，为了成千上百万，甚至上亿用户的实际使用，取决于想要做什么。

第二个重要的标准是运行速度，特别是训练大数据集时，一些框架能让你更高效地运行和训练神经网络。

还有一个标准人们不常提到，那就是这个框架是否真的开放，要是一个框架真的开放，它不仅需要开源，而且需要良好的管理。不幸的是，在软件行业中，一些公司有开源软件的历史，但是公司保持着对软件的全权控制，当几年时间过去，人们开始使用他们的软件时，一些公司开始逐渐关闭曾经开放的资源，或将功能转移到他们专营的云服务中。因此我会注意的一件事就是你能否相信这个框架能长时间保持开源，而不是在一家公司的控制之下，它未来有可能出于某种原因选择停止开源，即便现在这个软件是以开源的形式发布的。

程序框架就讲到这里，通过提供比数值线性代数库更高程度的抽象化，这里的每一个程序框架都能让你在开发深度机器学习应用时更加高效。

3.11 TensorFlow

先提一个启发性的问题，假设有一个损失函数J需要最小化，在本例中，我将使用这个高度简化的损失函数，$J(w) = w^2 - 10w + 25$，很容易发现使损失函数最小的w值是 5，但假设我们不知道这点，只有这个函数，我们来看一下怎样用 **TensorFlow** 将其最小化，因为一个非常类似的程序结构可以用来训练神经网络。其中可以有一些复杂的损失函数$J(w, b)$取决于神经网络的所有参数，然后就能用 **TensorFlow** 自动找到使损失函数最小的w和b的值。但让我们先从左边这个更简单的例子入手。

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

In [2]: w=tf.Variable(0,dtype=tf.float32)
        cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)
        init=tf.global_variables_initializer()
        session=tf.Session()
        session.run(init)
        print(session.run(w))

        0.0
```

我在我的 **Jupyter notebook** 中运行 **Python**，

```
import numpy as np
#导入 TensorFlow
import tensorflow as tf
#TensorFlow 中，用 tf.Variable()来定义参数 w
w = tf.Variable(0, dtype = tf.float32)
#然后我们定义损失函数：
cost = tf.add(tf.add(w**2, tf.multiply(- 10.,w)),25)
#然后我们再写优化器，让我们用 0.01 的学习率，目标是最小化损失。
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
#最后下面的几行是惯用表达式：
init = tf.global_variables_initializer()
session = tf.Sessions()  # 这样就开启了一个 TensorFlow session。
session.run(init)        # 来初始化全局变量。
#然后让 TensorFlow 评估一个变量，我们要用到：
session.run(w)
#上面的这一行将 w 初始化为 0，并定义损失函数，我们定义 train 为学习算法，它用梯度下降法优化器使损失函数最小化，但实际上我们还没有运行学习算法，所以 session.run(w)评估了 w，让我：
print(session.run(w))
```

#现在让我们输入：

session.run(train)，它所做的就是运行一步梯度下降法。

#接下来在运行了一步梯度下降法后，让我们评估一下 w 的值，再 print：

print(session.run(w))

#在一步梯度下降法之后，w 现在是 0.1。

```
In [3]: session.run(train)
        print(session.run(w))
```

```
0.1
```

现在我们运行梯度下降 1000 次迭代：

```
In [4]: for i in range(1000):
            session.run(train)
        print(session.run(w))
```

```
4.99999
```

这是运行了梯度下降的 1000 次迭代，最后w变成了 4.99999，记不记得我们说$(w-5)^2$最小化，因此w的最优值是 5，这个结果已经很接近了。

这里用到的一些函数，要注意w是要优化的参数，因此将它称为变量，注意我们需要做的就是定义一个损失函数，使用这些 add 和 multiply 之类的函数。**TensorFlow** 知道如何对 add 和 mutiply，还有其它函数求导，这就是为什么你只需基本实现前向传播，它能弄明白如何做反向传播和梯度计算，因为它已经内置在 add，multiply 和平方函数中。

要是觉得这种写法不好看的话，**TensorFlow** 其实还重载了一般的加减运算等，因此也可以把$cost$写成更好看的形式，把之前的 cost 标成注释，重新运行，得到了同样的结果。

```
In [5]: w=tf.Variable(0,dtype=tf.float32)
        #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        cost=w**2-10*w+25
        train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

        init=tf.global_variables_initializer()
        session=tf.Session()
        session.run(init)
        print(session.run(w))
```

```
0.0
```

```
In [6]: session.run(train)
        print(session.run(w))
```

```
0.1
```

```
In [7]: for i in range(1000):
            session.run(train)
        print(session.run(w))
```

```
4.99999
```

TensorFlow 还有一个特点，那就是这个例子将w的一个固定函数最小化了。如果你想要最小化的函数是训练集函数又如何呢？不管你有什么训练数据x，当你训练神经网络时，训练数据x会改变，那么如何把训练数据加入 **TensorFlow** 程序呢？

我会定义x，把它想做扮演训练数据的角色，事实上训练数据有x和y，但这个例子中只有x，把x定义为：`x = tf.placeholder(tf.float32,[3,1])`，让它成为[3,1]数组，我要做的就是，因为$cost$这个二次方程的三项前有固定的系数，它是$w^2+10w+25$，我们可以把这些数字 1，-10 和 25 变成数据，我要做的就是把$cost$替换成：

`cost = x[0][0]*w**2 +x[1][0]*w + x[2][0]`，现在x变成了控制这个二次函数系数的数据，这个 **placeholder** 函数告诉 **TensorFlow**，你稍后会为x提供数值。

让我们再定义一个数组，`coefficient = np.array([[1.],[-10.],[25.]])`，这就是我们要接入x的数据。最后我们需要用某种方式把这个系数数组接入变量x，做到这一点的句法是，在训练这一步中，要提供给x的数值，我在这里设置：

`feed_dict = {x:coefficients}`

好了，希望没有语法错误，我们重新运行它，希望得到和之前一样的结果。

```
In [11]: coefficients=np.array([[1.],[-10.],[25.]])

         w=tf.Variable(0,dtype=tf.float32)
         x=tf.placeholder(tf.float32,[3,1])
         #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
         #cost=w**2-10*w+25
         cost=x[0][0]*w**2+x[1][0]*w+x[2][0]
         train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

         init=tf.global_variables_initializer()
         session=tf.Session()
         session.run(init)
         print(session.run(w))

         0.0

In [12]: session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         0.1

In [13]: for i in range(1000):
             session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         4.99999
```

现在如果你想改变这个二次函数的系数，假设把：

`coefficient = np.array([[1.],[-10.],[25.]])`

改为：

`coefficient = np.array([[1.],[-20.],[100.]])`

现在这个函数就变成了$(w-10)^2$，如果我重新运行，希望我得到的使$(w-10)^2$最小化的w值为 10，让我们看一下，很好，在梯度下降 1000 次迭代之后，我们得到接近 10 的w。

In [14]:
```
coefficients=np.array([[1.],[-20.],[100.]])

w=tf.Variable(0,dtype=tf.float32)
x=tf.placeholder(tf.float32,[3,1])
#cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
#cost=w**2-10*w+25
cost=x[0][0]*w**2+x[1][0]*w+x[2][0]
train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init=tf.global_variables_initializer()
session=tf.Session()
session.run(init)
print(session.run(w))
```

```
0.0
```

In [15]:
```
session.run(train,feed_dict={x:coefficients})
print(session.run(w))
```

```
0.2
```

In [16]:
```
for i in range(1000):
    session.run(train,feed_dict={x:coefficients})
print(session.run(w))
```

```
9.99998
```

TensorFlow 中的 **placeholder** 是一个之后会赋值的变量，这种方式便于把训练数据加入损失方程，当运行训练迭代，用 `feed_dict` 来让 `x=coefficients`。如果在做 **mini-batch** 梯度下降，在每次迭代时，需要插入不同的 **mini-batch**，那么每次迭代就用 `feed_dict` 来喂入训练集的不同子集，把不同的 **mini-batch** 喂入损失函数需要数据的地方。

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
session = tf.Session()
session.run(init)
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

还有最后一点我想提一下，这三行（蓝色大括号部分）在 **TensorFlow** 里是符合表达习惯的，有些程序员会用这种形式来替代，作用基本上是一样的。