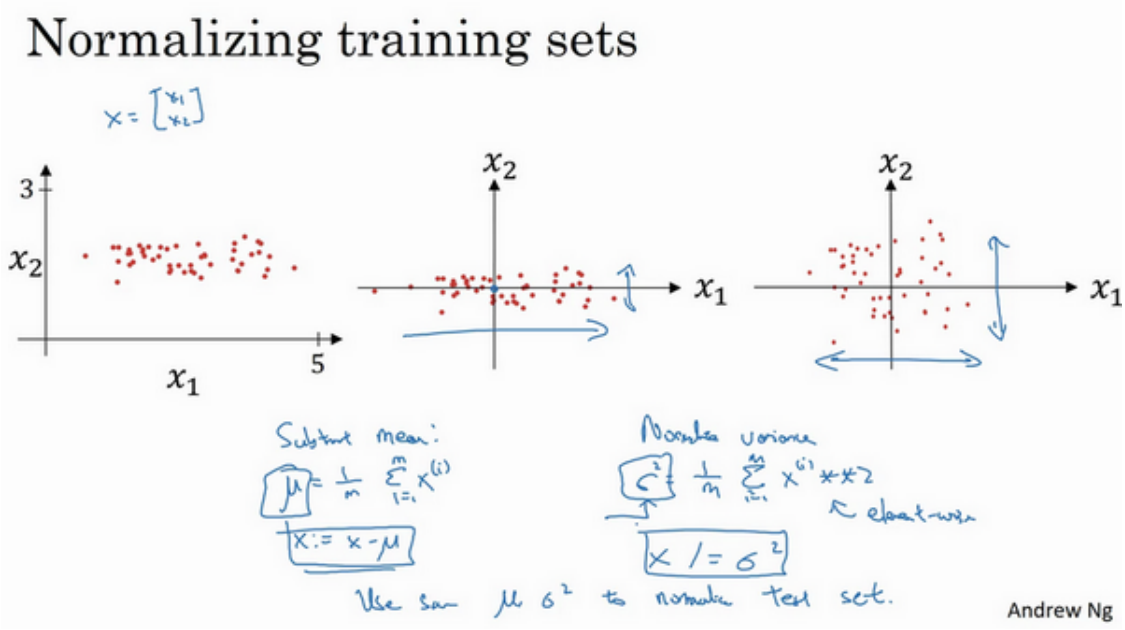
1.9 归一化输入（Normalizing inputs）

训练神经网络，其中一个加速训练的方法就是归一化输入。假设一个训练集有两个特征，输入特征为 2 维，归一化需要两个步骤：

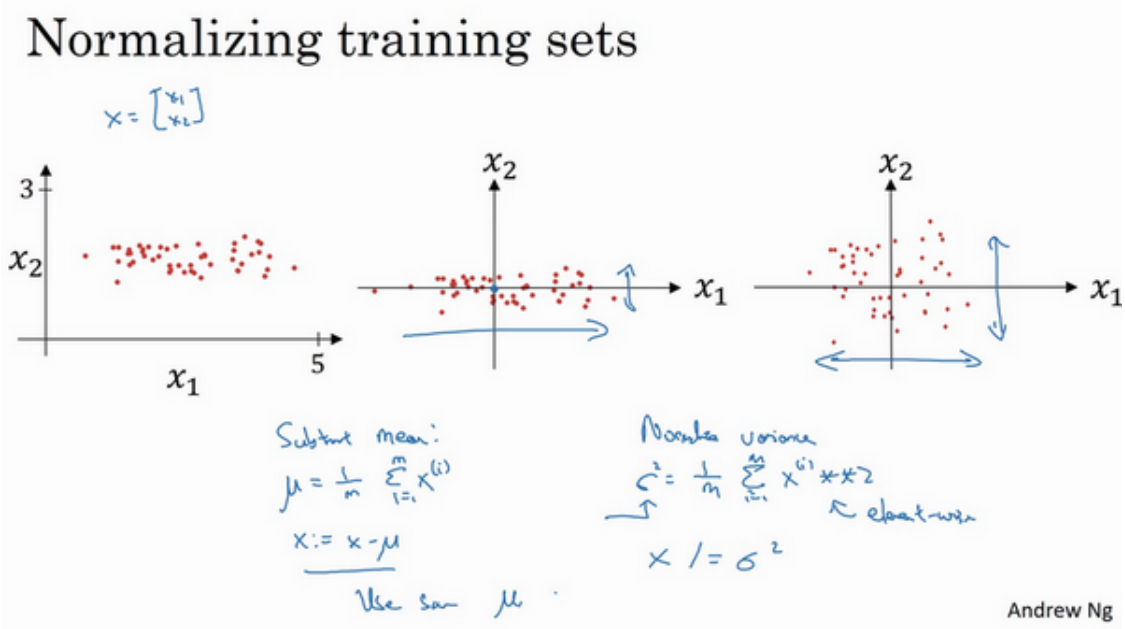
1.零均值

2.归一化方差；

我们希望无论是训练集和测试集都是通过相同的μ和σ^2定义的数据转换，这两个是由训练集得出来的。



第一步是零均值化，$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$，它是一个向量，$x$等于每个训练数据 x减去μ，意思是移动训练集，直到它完成零均值化。



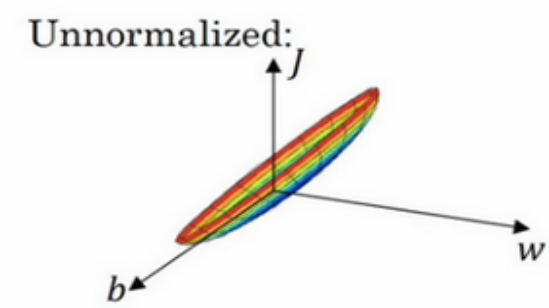
第二步是归一化方差，注意特征x_1的方差比特征x_2的方差要大得多，我们要做的是给σ赋值，$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2$，这是节点$y$ 的平方，σ^2是一个向量，它的每个特征都有方差，注意，我们已经完成零值均化，$(x^{(i)})^2$元素y^2就是方差，我们把所有数据除以向量σ^2，最后变成上图形式。

x_1和x_2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 μ 和 σ^2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，不论μ的值是什么，也不论σ^2的值是什么，这两个公式中都会用到它们。所以你要用同样的方法调整测试集，而不是在训练集和测试集上分别预估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，其中μ和σ^2是由训练集数据计算得来的。

我们为什么要这么做呢？为什么我们想要归一化输入特征，回想一下右上角所定义的代价函数。

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

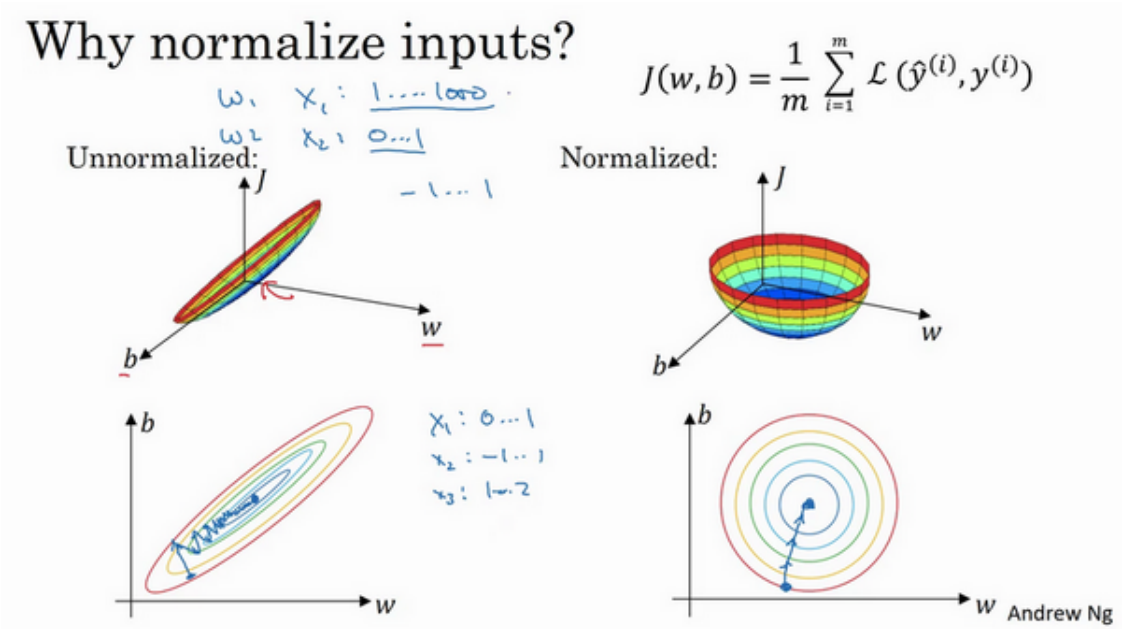
如果你使用非归一化的输入特征，代价函数会像这样：



这是一个非常细长狭窄的代价函数，你要找的最小值应该在这里。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，代价函数就有点像狭长的碗一样，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。

当然，实际上w是一个高维向量，因此用二维绘制w并不能正确地传达并直观理解，但总地直观理解是代价函数会更圆一些，而且更容易优化，前提是特征都在相似范围内，而不是从 1 到 1000，0 到 1 的范围，而是在-1 到 1 范围内或相似偏差，这使得代价函数J优化起来更简单快速。



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。

所以如果输入特征处于不同范围内，可能有些特征值从 0 到 1，有些从 1 到 1000，那么归一化特征值就非常重要了。如果特征值处于相似范围内，那么归一化就不是很重要了。执行这类归一化并不会产生什么危害，我通常会做归一化处理，虽然我不确定它能否提高训练或算法速度。

这就是归一化特征输入，下节课我们将继续讨论提升神经网络训练速度的方法。

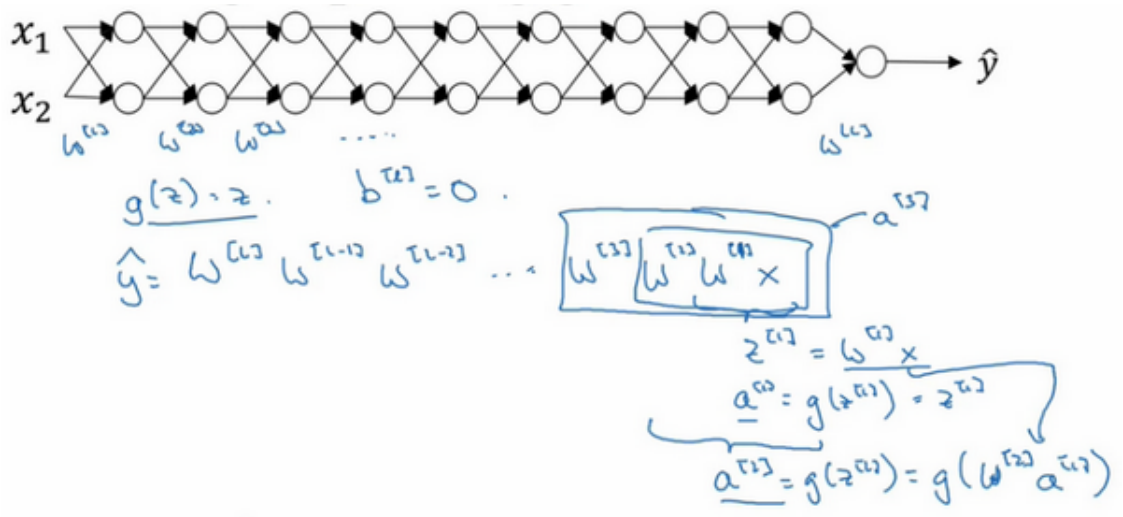
1.10 梯度消失/梯度爆炸（Vanishing / Exploding gradients）

训练神经网络，尤其是深度神经所面临的一个问题就是梯度消失或梯度爆炸，也就是你训练神经网络的时候，导数或坡度有时会变得非常大，或者非常小，甚至于以指数方式变小，这加大了训练的难度。

这节课，你将会了解梯度消失或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。 假设你正在训练这样一个极深的神经网络，为了节约幻灯片上的空间，我画的神经网络每层只有两个隐藏单元，但它可能含有更多，但这个神经网络会有参数$W^{[1]}$，$W^{[2]}$，$W^{[3]}$等等，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，我们忽略b，假设$b^{[l]}$=0，如果那样的话，输出：

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

如果你想考验我的数学水平，$W^{[1]}x=z^{[1]}$，因为$b=0$，所以我想$z^{[1]}=W^{[1]}x$，$a^{[1]}=g(z^{[1]})$，因为我们使用了一个线性激活函数，它等于$z^{[1]}$，所以第一项$W^{[1]}x=a^{[1]}$，通过推理，你会得出$W^{[2]}W^{[1]}x=a^{[2]}$，因为$a^{[2]}=g(z^{[2]})$，还等于$g(W^{[2]}a^{[1]})$，可以用$W^{[1]}x$替换$a^{[1]}$，所以这一项就等于$a^{[2]}$，这个就是$a^{[3]}(W^{[3]}W^{[2]}W^{[1]}x)$。



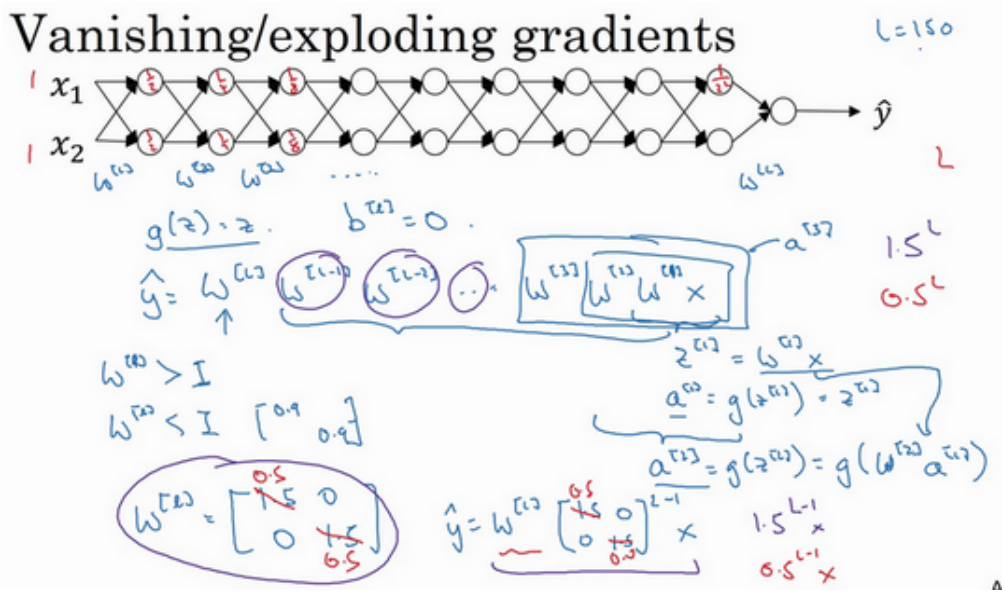
所有这些矩阵数据传递的协议将给出$\hat{y}$而不是y的值。

假设每个权重矩阵$W^{[l]}=\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}$，从技术上来讲，最后一项有不同维度，可能它就是余下的权重矩阵，$y=W^{[1]}\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}^{(L-1)}x$，因为我们假设所有矩阵都等于它，它是 1.5 倍的单位矩阵，最后的计算结果就是$\hat{y}$，$\hat{y}$也就是等于$1.5^{(L-1)}x$。如果对于一个深度神经网络来说L值较大，那么$\hat{y}$的值也会非常大，实际上它呈指数级增长的，它增长的比率是1.5^{L}，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$\frac{1}{2^L}$，所以作为自定义函数，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。

我希望你得到的直观理解是，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，可能是$\begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix}$。



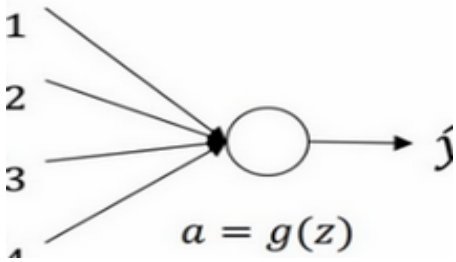
在深度神经网络中，激活函数将以指数级递减，虽然我只是讨论了激活函数以与L相关的指数级数增长或下降，它也适用于与层数L相关的导数或梯度函数，也是呈指数级增长或呈指数递减。

对于当前的神经网络，假设$L = 150$，最近 **Microsoft** 对 152 层神经网络的研究取得了很大进展，在这样一个深度神经网络中，如果激活函数或梯度函数以与L相关的指数增长或递减，它们的值将会变得极大或极小，从而导致训练难度上升，尤其是梯度指数小于L时，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。

总结一下，我们讲了深度神经网络是如何产生梯度消失或爆炸问题的，实际上，在很长一段时间内，它曾是训练深度神经网络的阻力，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多帮助。

1.11 神经网络的权重初始化（Weight Initialization for Deep Networks）

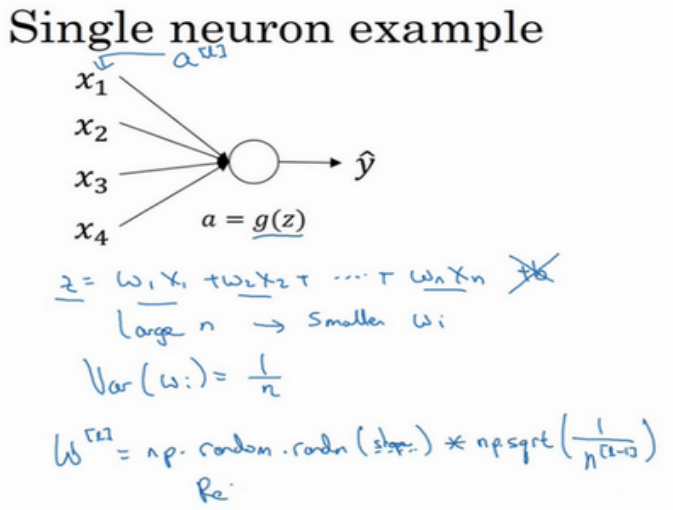
上节课，我们学习了深度神经网络如何产生梯度消失和梯度爆炸问题，最终针对该问题，我们想出了一个不完整的解决方案，虽然不能彻底解决问题，却很有用，有助于我们为神经网络更谨慎地选择随机初始化参数，为了更好地理解它，我们先举一个神经单元初始化地例子，然后再演变到整个深度网络。



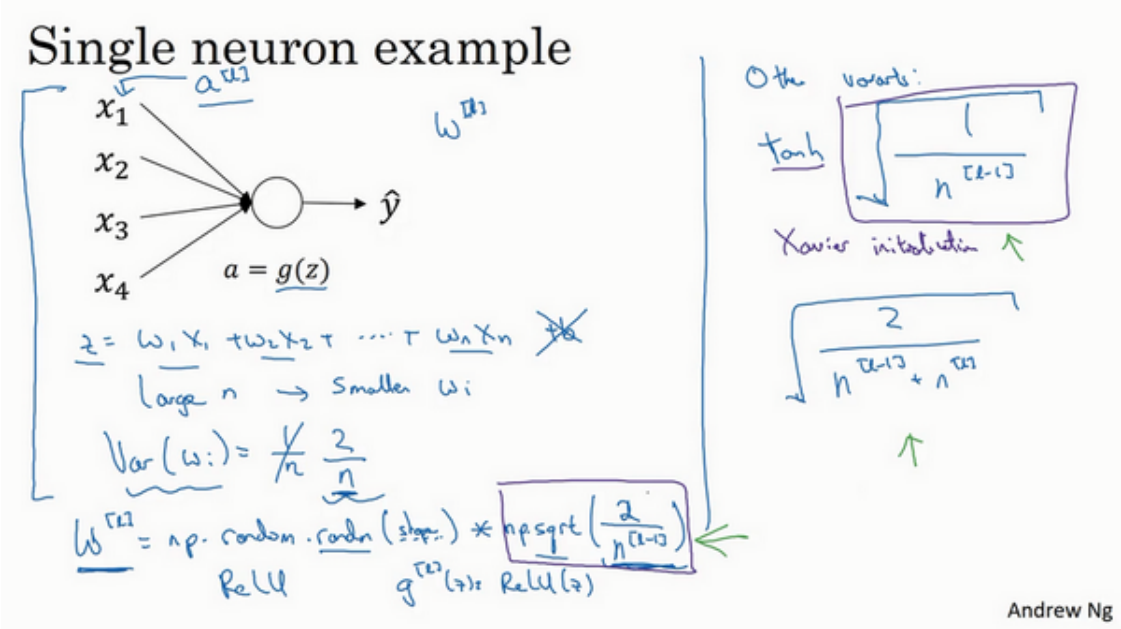
我们来看看只有一个神经元的情况，然后才是深度网络。

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$a^{[l]}$，暂时我们用x表示。

$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$，$b = 0$，暂时忽略$b$，为了预防$z$值过大或过小，你可以看到$n$越大，你希望$w_i$越小，因为$z$是$w_ix_i$的和，如果你把很多此类项相加，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$w^{[l]} = np.random.randn(\text{shape}) * \text{np.sqrt}(\frac{1}{n^{[l-1]}})$，$n^{[l-1]}$就是我喂给第$l$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，这就没解决问题（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，一篇由 **Herd** 等人撰写的论文曾介绍过。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。**Yoshua Bengio** 和他的同事还提出另一种方法，你可能在一些论文中看到过，它们使用的是公式$\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$。其它理论已对此证明，但如果你想用 **Relu** 激活函数，也就是最常用的激活函数，我会用这个公式$\text{np.sqrt}(\frac{2}{n^{[l-1]}})$，如果使用 **tanh** 函数，可以用公式$\sqrt{\frac{1}{n^{[l-1]}}}$，有些作者也会使用这个函数。

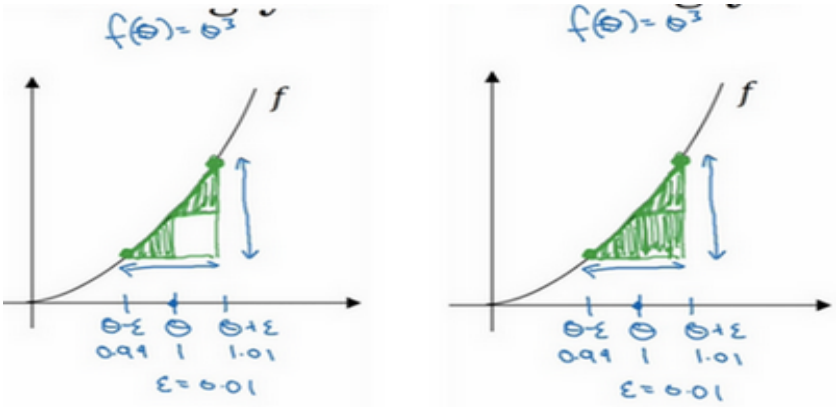
实际上，我认为所有这些公式只是给你一个起点，它们给出初始化权重矩阵的方差的默认值，如果你想添加方差，方差参数则是另一个你需要调整的超级参数，可以给公式

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。

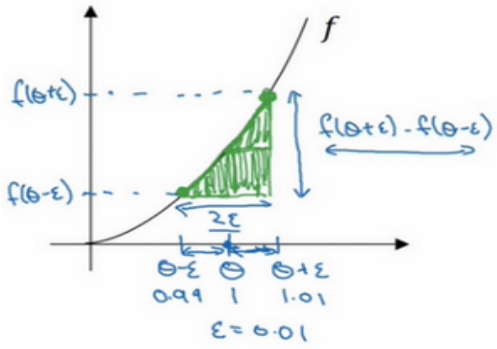
希望你现在对梯度消失或爆炸问题以及如何为权重初始化合理值已经有了一个直观认识，希望你设置的权重矩阵既不会增长过快，也不会太快下降到 0，从而训练出一个权重或梯度不会增长或消失过快的深度网络。我们在训练深度网络时，这也是一个加快训练速度的技巧。

1.12 梯度的数值逼近（Numerical approximation of gradients）

在实施 **backprop** 时，有一个测试叫做梯度检验，它的作用是确保 **backprop** 正确实施。因为有时候，你虽然写下了这些方程式，却不能 100%确定，执行 **backprop** 的所有细节都是正确的。为了逐渐实现梯度检验，我们首先说说如何计算梯度的数值逼近，下节课，我们将讨论如何在 **backprop** 中执行梯度检验，以确保 **backprop** 正确实施。

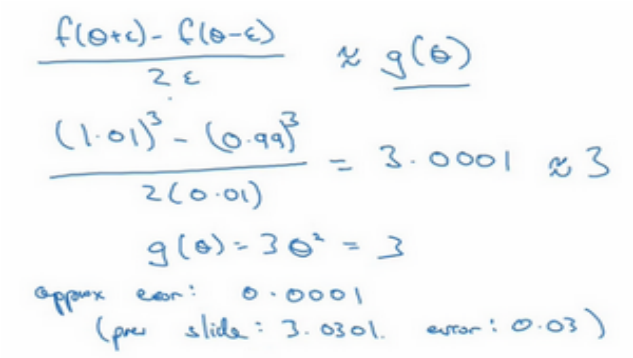


我们先画出函数f，标记为$f(\theta)$，$f(\theta)=\theta^3$，先看一下θ的值，假设$\theta=1$，不增大θ的值，而是在θ 右侧，设置一个$\theta+\varepsilon$，在θ左侧，设置$\theta-\varepsilon$。因此$\theta=1$，$\theta+\varepsilon=1.01, \theta-\varepsilon=0.99$,，跟以前一样，$\varepsilon$的值为 0.01，看下这个小三角形，计算高和宽的比值，就是更准确的梯度预估，选择f函数在$\theta-\varepsilon$上的这个点，用这个较大三角形的高比上宽，技术上的原因我就不详细解释了，较大三角形的高宽比值更接近于θ的导数，把右上角的三角形下移，好像有了两个三角形，右上角有一个，左下角有一个，我们通过这个绿色大三角形同时考虑了这两个小三角形。所以我们得到的不是一个单边公差而是一个双边公差。



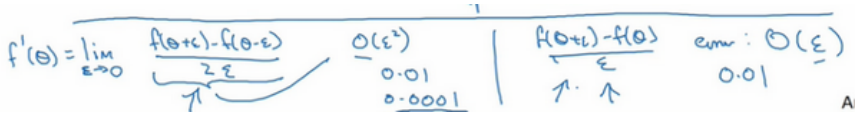
我们写一下数据算式，图中绿色三角形上边的点的值是$f(\theta+\varepsilon)$，下边的点是$f(\theta-\varepsilon)$，这个三角形的高度是$f(\theta+\varepsilon)-f(\theta-\varepsilon)$，这两个宽度都是 ε，所以三角形的宽度是$2\varepsilon$，高宽比值为$\frac{f(\theta+\varepsilon)-(\theta-\varepsilon)}{2\varepsilon}$，它的期望值接近$g(\theta)$，$f(\theta)=\theta^3$传入参数值：

$\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} = \frac{(1.01)^3-(0.99)^3}{2\times0.01}$，大家可以用计算器算算结果，结果应该是 3.0001，而前面一张幻灯片上面是，当$\theta = 1$时，$g(\theta) = 3\theta^2 = 3$，所以这两个$g(\theta)$值非常接近，逼近误差为 0.0001，前一张幻灯片，我们只考虑了单边公差，即从θ到$\theta + \varepsilon$之间的误差，$g(\theta)$的值为 3.0301，逼近误差是 0.03，不是 0.0001，所以使用双边误差的方法更逼近导数，其结果接近于 3，现在我们更加确信，$g(\theta)$可能是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，我认为这种方法还是非常值得使用的，因为它的结果更准确。



这是一些你可能比较熟悉的微积分的理论，如果你不太明白我讲的这些理论也没关系，导数的官方定义是针对值很小的ε，导数的官方定义是$f'(\theta) = \frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$，如果你上过微积分课，应该学过无穷尽的定义，我就不在这里讲了。

对于一个非零的ε，它的逼近误差可以写成$O(\varepsilon^2)$，ε 值非常小，如果$\varepsilon = 0.01$，$\varepsilon^2 = 0.0001$，大写符号O的含义是指逼近误差其实是一些常量乘以ε^2，但它的确是很准确的逼近误差，所以大写O的常量有时是 1。然而，如果我们用另外一个公式逼近误差就是$O(\varepsilon)$，当ε小于 1 时，实际上ε比ε^2大很多，所以这个公式近似值远没有左边公式的准确，所以在执行梯度检验时，我们使用双边误差，即$\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$，而不使用单边公差，因为它不够准确。



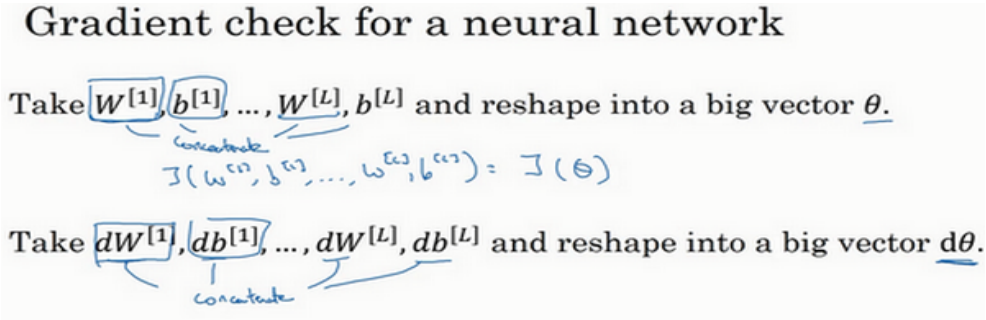
如果你不理解上面两条结论，所有公式都在这儿，不用担心，如果你对微积分和数值逼近有所了解，这些信息已经足够多了，重点是要记住，双边误差公式的结果更准确，下节课我们做梯度检验时就会用到这个方法。我们讲了如何使用双边误差来判断别人给你的函数$g(\theta)$，是否正确实现了函数f的偏导，现在我们可以使用这个方法来检验反向传播是否得以正确实施，如果不正确，它可能有 **bug** 需要你来解决。

1.13 梯度检验（Gradient checking）

梯度检验帮我们节省了很多时间，也多次帮我发现 **backprop** 实施过程中的 bug，接下来，我们看看如何利用它来调试或检验 **backprop** 的实施是否正确。

假设你的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。

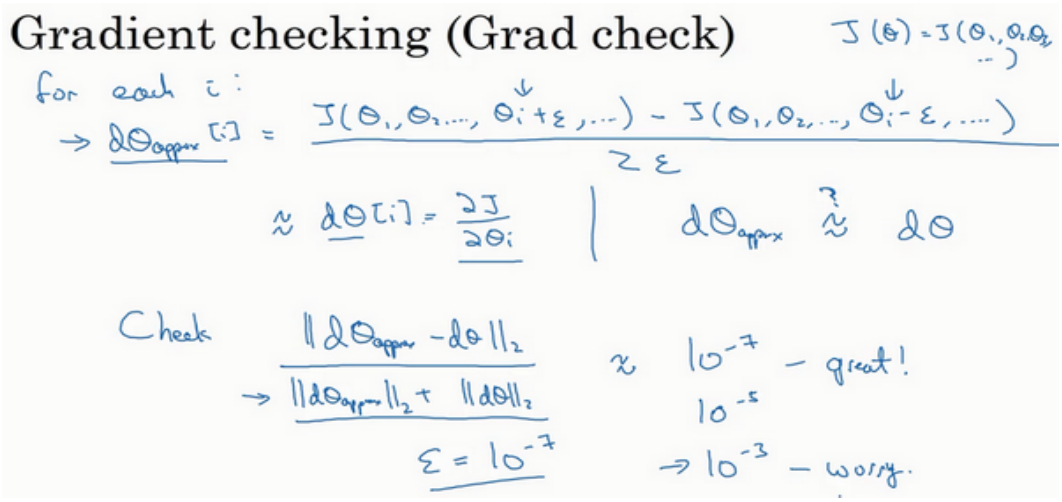
同样的，把$dW^{[1]}$转换成矩阵，$db^{[1]}$已经是一个向量了，直到把$dW^{[l]}$转换成矩阵，这样所有的dW都已经是矩阵，注意$dW^{[1]}$与$W^{[1]}$具有相同维度，$db^{[1]}$与$b^{[1]}$具有相同维度。经过相同的转换和连接运算操作之后，你可以把所有导数转换成一个大向量$d\theta$，它与θ具有相同维度，现在的问题是$d\theta$和代价函数J的梯度或坡度有什么关系？



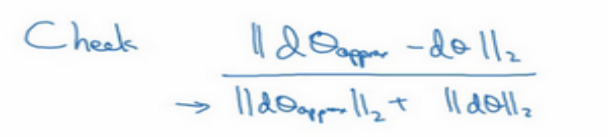
这就是实施梯度检验的过程，英语里通常简称为“**grad check**”，首先，我们要清楚J是超参数θ的一个函数，你也可以将J函数展开为$J(\theta_1, \theta_2, \theta_3, \dots\dots)$，不论超级参数向量$\theta$的维度是多少，为了实施梯度检验，你要做的就是循环执行，从而对每个i也就是对每个θ组成元素计算$d\theta_{\text{approx}}[i]$的值，我使用双边误差，也就是

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

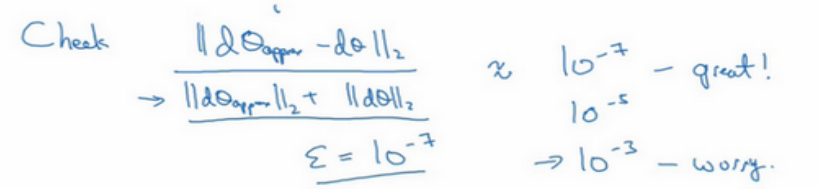
只对θ_i增加ε，其它项保持不变，因为我们使用的是双边误差，对另一边做同样的操作，只不过是减去ε，θ其它项全都保持不变。



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$\theta$具有相同维度，你要做的就是验证这些向量是否彼此接近。



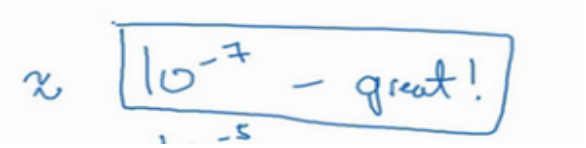
具体来说，如何定义两个向量是否真的接近彼此？我一般做下列运算，计算这两个向量的距离，$d\theta_{\text{approx}}[i]-d\theta[i]$的欧几里得范数，注意这里（$||d\theta_{\text{approx}}-d\theta||_2$）没有平方，它是误差平方之和，然后求平方根，得到欧式距离，然后用向量长度归一化，使用向量长度的欧几里得范数。分母只是用于预防这些向量太小或太大，分母使得这个方程式变成比率，我们实际执行这个方程式，ε可能为10^{-7}，使用这个取值范围内的ε，如果你发现计算方程式得到的值为10^{-7}或更小，这就很好，这就意味着导数逼近很有可能是正确的，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}，我就会担心是否存在 **bug**，计算结果应该比10^{-3}小很多，如果比10^{-3}大很多，我就会很担心，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}），那么，你的实施可能是正确的。

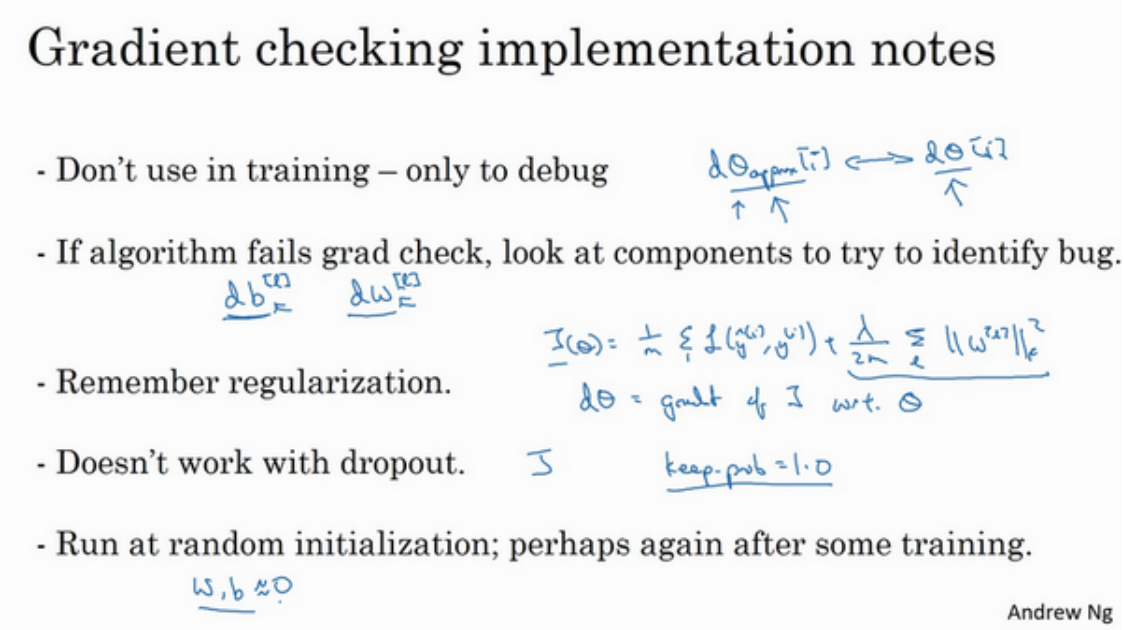


在实施神经网络时，我经常需要执行 **foreprop** 和 **backprop**，然后我可能发现这个梯度检验有一个相对较大的值，我会怀疑存在 **bug**，然后开始调试，调试，调试，调试一段时间后，我得到一个很小的梯度检验值，现在我可以很自信的说，神经网络实施是正确的。

现在你已经了解了梯度检验的工作原理，它帮助我在神经网络实施中发现了很多 **bug**，希望它对你也有所帮助。

1.14 梯度检验应用的注意事项（Gradient Checking Implementation Notes）

这节课，分享一些关于如何在神经网络实施梯度检验的实用技巧和注意事项。



首先，不要在训练中使用梯度检验，它只用于调试。我的意思是，计算所有i值的$d\theta_{\text{approx}}[i]$是一个非常漫长的计算过程，为了实施梯度下降，你必须使用W和b **backprop** 来计算$d\theta$，并使用 **backprop** 来计算导数，只要调试的时候，你才会计算它，来确认数值是否接近$d\theta$。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它，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，检查每一项，并试着找出 bug，也就是说，如果$d\theta_{\text{approx}}[i]$与$d\theta[i]$的值相差很大，我们要做的就是查找不同的$i$值，看看是哪个导致$d\theta_{\text{approx}}[i]$与$d\theta[i]$的值相差这么多。举个例子，如果你发现，相对某些层或某层的θ或$d\theta$的值相差很大，但是$\text{dw}^{[l]}$的各项非常接近，注意θ的各项与b和w的各项都是一一对应的，这时，你可能会发现，在计算参数b的导数db的过程中存在 **bug**。反过来也是一样，如果你发现它们的值相差很大，$d\theta_{\text{approx}}[i]$的值与$d\theta[i]$的值相差很大，你会发现所有这些项目都来自于$dw$或某层的$dw$，可能帮你定位 **bug** 的位置，虽然未必能够帮你准确定位 **bug** 的位置，但它可以帮助你估测需要在哪些地方追踪 **bug**。

第三点，在实施梯度检验时，如果使用正则化，请注意正则项。如果代价函数$J(\theta) =$

$\frac{1}{m}\sum L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum ||W^{[l]}||^2$，这就是代价函数$J$的定义，$d\theta$等于与$\theta$相关的$J$函数的梯度，包括这个正则项，记住一定要包括这个正则项。

第四点，梯度检验不能与 **dropout** 同时使用，因为每次迭代过程中，**dropout** 会随机消除隐藏层单元的不同子集，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，这些节点都有可能被消除，所以很难计算代价函数J。你只是对成本函数做抽样，用**dropout**，每次随机消除不同的子集，所以很难用梯度检验来双重检验**dropout**的计算，所以我一般不同时使用梯度检验和 **dropout**。如果你想这样做，可以把 **dropout** 中的 **keepprob** 设置为 1.0，然后打开 **dropout**，并寄希望于 **dropout** 的实施是正确的，你还可以做点别的，比如修改节点丢失模式确定梯度检验是正确的。实际上，我一般不这么做，我建议关闭 **dropout**，用梯度检验进行双重检查，在没有 **dropout** 的情况下，你的算法至少是正确的，然后打开 **dropout**。

最后一点，也是比较微妙的一点，现实中几乎不会出现这种情况。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……，但是在运行梯度下降时，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。你需要做一件事，我不经常这么做，就是在随机初始化过程中，运行梯度检验，然后再训练网络，w和b会有一段时间远离 0，如果随机初始化值比较小，反复训练网络之后，再重新运行梯度检验。

这就是梯度检验，恭喜大家，这是本周最后一课了。回顾这一周，我们讲了如何配置训练集，验证集和测试集，如何分析偏差和方差，如何处理高偏差或高方差以及高偏差和高方差并存的问题，如何在神经网络中应用不同形式的正则化，如$L2$正则化和 **dropout**，还有加快神经网络训练速度的技巧，最后是梯度检验。