# Initialization & Regularization & Gradient Checking

# 1、作业简介

本文主要有以下三个部分：

1. 初始化参数：

1.1 使用0来初始化参数。

1.2 使用随机数来初始化参数。

1.3 使用抑梯度异常初始化参数（参见视频中的梯度消失和梯度爆炸）。

2. 正则化模型：

2.1 使用二范数对二分类模型正则化，尝试避免过拟合。

2.2 使用随机删除节点的方法精简模型，同样是为了尝试避免过拟合。

3. 梯度校验：

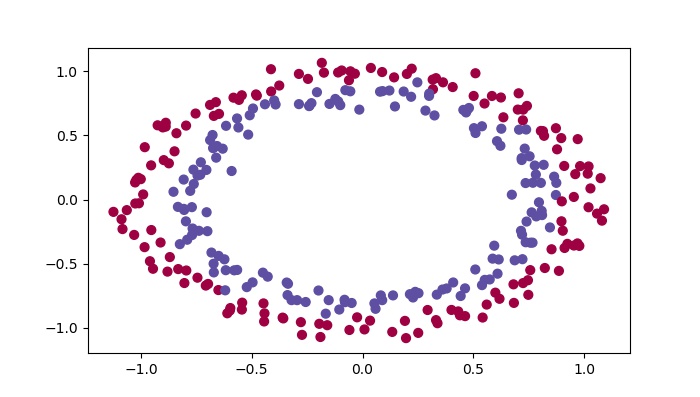
3.1 对模型使用梯度校验，检测它是否在梯度下降的过程中出现误差过大的情况。

## 2、初始化（第一部分）

在我们模型训练的第一步中，我们首先需要给我们的模型参数一个初始值。

一个好的初始值有助于我们整个训练过程，一方面，可以加快梯度下降的收敛速度；另一方面，使得误差更有可能减小到一个更小的值。

在初始化之前，来看看数据集是怎样的，读取并绘制数据，可得到如下结果：



**2.1 神经网络模型**

我们将要建立一个分类器把蓝点和红点分开，在之前我们已经实现过一个3层的神经网络，我们将对它进行初始化。

将尝试下面三种初始化方法:

①初始化为0（Zeros initialization）：在输入参数中全部初始化为0。

②初始化为随机数（Random initialization）：把输入参数设置为随机值，

权重初始化为大的随机值。

③抑梯度异常初始化（He initialization）：参见梯度消失和梯度爆炸的那一个视频。

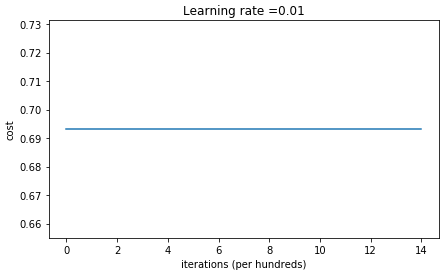
**2.2 零初始化**

在神经网络中有两种类型的参数需要初始化：

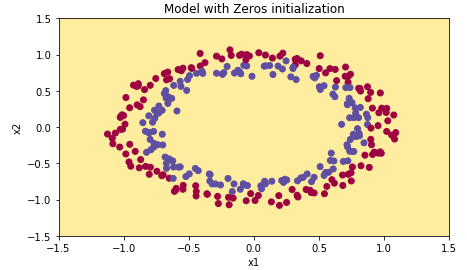
-权重矩阵

-偏置矢量

使用初始化为0的参数来训练模型，由结果可知学习率一直没有变化，也就是说这个模型根本没有学习。



经过预测，得到训练集与测试集的准确度均为0.5，系统性能较差，且成本并没有真正降低，该算法的性能并不比随机猜测更好。为什么？可以看看预测和决策边界的细节。



模型对每个例子都预测为0，分类失败。通常，零初始化都会导致神经网络无法打破对称性。这意味着每一层中的每一个神经元都将学习相同的东西，你也可以训练一个神经网络，每一层的，而这个神经网络的功能并不比logistic回归这样的线性分类器强大。

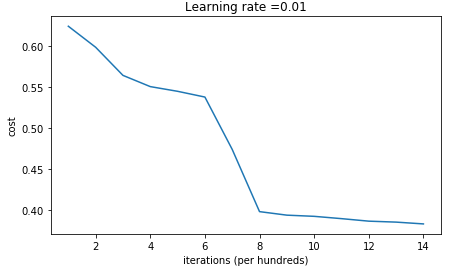
应该记住的是：

-权重应随机初始化以打破对称性。

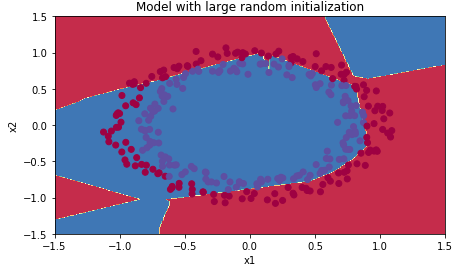
-但是，可以将偏差初始化为零。只要是随机初始化的，对称性仍然是破坏的。

**2.3 随机初始化**

为了打破对称性（break symmetry），我们可以随机地把参数赋值。在随机初始化之后，每个神经元可以开始学习其输入的不同功能，我们还会设置比较大的参数值，看看会发生什么。



经过预测，得到训练集的准确度均为0.83，测试集的准确度为0.86。



结论：

-我们可以看到开始的代价很高，这是因为由于具有较大的随机权重，最后一个激活(sigmoid)输出的结果非常接近于0或1，而当它出现错误时，它会导致非常高的损失。实际上，当时，损失是无穷大的。

-初始化参数如果不好的话会导致梯度消失、爆炸，这也会减慢优化算法。

-如果我们对这个网络进行更长时间的训练，将看到更好的结果，但是使用过大的随机数初始化会减慢优化的速度。

-随着迭代次数的增加，最终的分离结果还能有一定的提升。但是如果初始值选择更大的话，性能也会随之降低

  总而言之，将权重初始化为非常大的时候其实效果并不好，下面我们试试小一点的参数值。

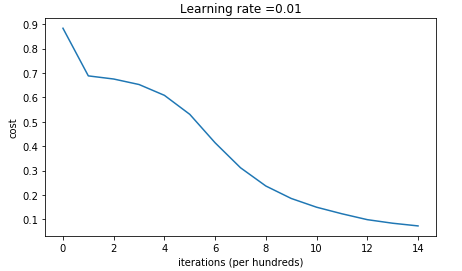
**2.4 抑梯度异常初始化**

He初始化与Xavire初始化十分类似。

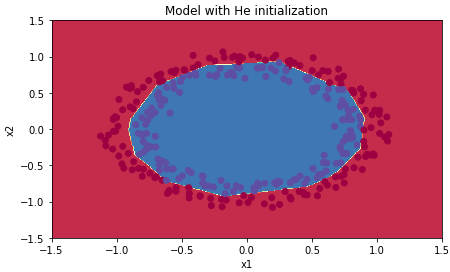
Xavire初始化中，系数为, sqrt(1, layers\_dims[l-1])。

He初始化中，系数为, sqrt(2, layers\_dims[l-1])。

这样就基本把参数W初始化到了1附近。

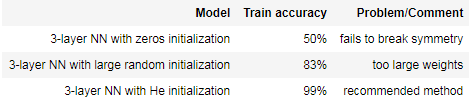


经过预测，得到训练集的准确度均为0.993333333333，测试集的准确度为0.96，可以看到误差越来越小。



从上图可以看出，使用He初始时，可以得到一个相对较好的分类结果,初始化的模型将蓝色和红色的点在少量的迭代中很好地分离出来。

在上面的学习中，我们尝试了三种不同的初始化方法，在相同的超参数和迭代次数的情况下，比较结果如下：



总结一下：

-不同的初始化方法可能导致性能最终不同。

-随机初始化有助于打破对称，使得不同隐藏层的单元可以学习到不同的参数。

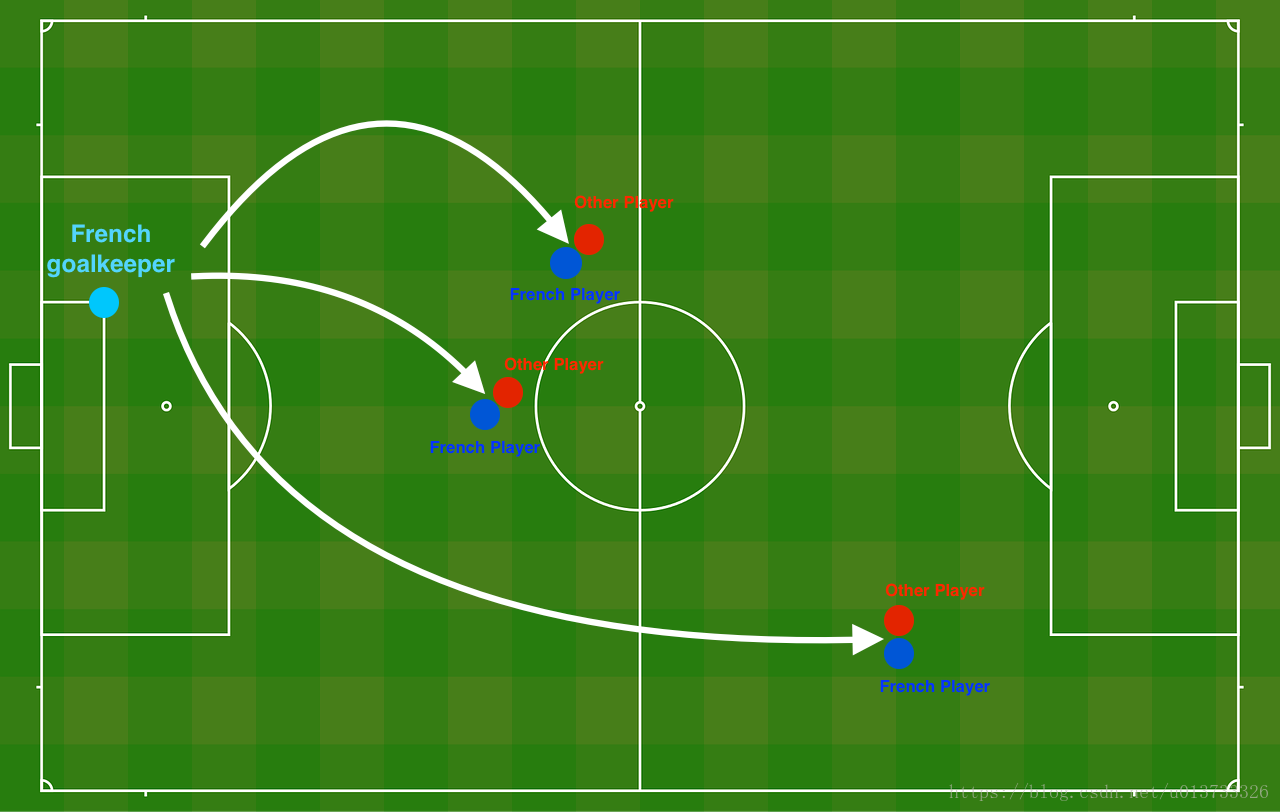
-初始化时，初始值不宜过大。

-He初始化搭配ReLU激活函数常常可以得到不错的效果。

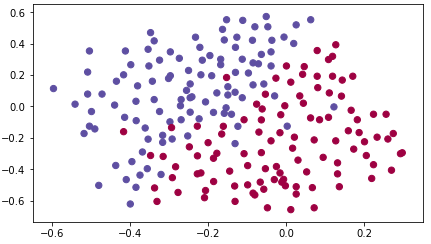
## 3、正则化（第二部分）

在深度学习中，如果数据集没有足够大的话，可能会导致一些过拟合的问题。过拟合导致的结果就是在训练集上有着很高的精确度，但是在遇到新的样本时，精确度下降会很严重。为了避免过拟合的问题，接下来我们要讲解的方式就是正则化。

问题描述：假设你现在是一个AI专家，你需要设计一个模型，可以用于推荐在足球场中守门员将球发至哪个位置可以让本队的球员抢到球的可能性更大。说白了，实际上就是一个二分类，一半是己方抢到球，一半就是对方抢到球，我们来看一下这个图：（你刚刚被法国足球公司聘为人工智能专家，他们希望你推荐法国守门员踢球的位置，这样法国队的球员就可以用他们的头击球了。）



首先读取并绘制数据集（法国过去10场比赛的二维数据集）：



* 上图中，每一个点对应一个足球落下的位置。对于蓝色的点，表示我方足球队员抢到球；对于红色的点，则表示对方球员抢到球。我们的目标是建立一个模型，来找到适合我方球员能抢到球的位置。

我们要做以下三件事，来对比出不同的模型的优劣：

-不使用正则化

-使用正则化

-使用L2正则化

-使用随机节点删除

* 每个点对应于足球场上的一个位置，在法国守门员从足球场左侧射门后，足球运动员用头击球。

-如果圆点是蓝色的，则表示法国选手设法用头击球

-如果圆点是红色的，则表示对方球员用头击球

你的目标：使用一个深度学习模型来找到守门员应该踢球的位置。

数据集分析：这个数据集有点嘈杂，但它看起来像一条对角线，将左上半部分（蓝色）和右下半部分（红色）分开，效果很好。

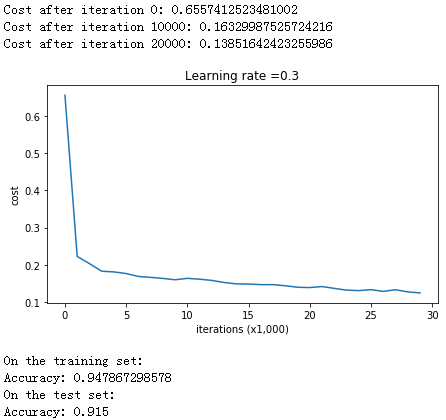
您将首先尝试非正则化模型。然后你将学习如何正则化它，并决定你将选择哪种模式来解决法国足球公司的问题。

**3.1 非正则化模型**

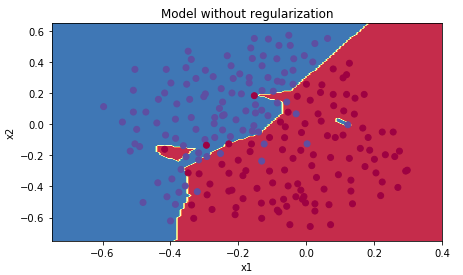
先来看一下我们的模型：

正则化模式：将lambd输入设置为非零值。我们使用“lambd”而不是“lambda”，因为“lambda”是Python中的保留关键字。

随机删除节点：将keep\_prob设置为小于1的值。



经过预测，得到训练集的准确度为94%，测试集的准确度为91.5%。



从图中可以看出，在无正则化时，分割曲线有了明显的过拟合特性。

**3.2 L2正则化**

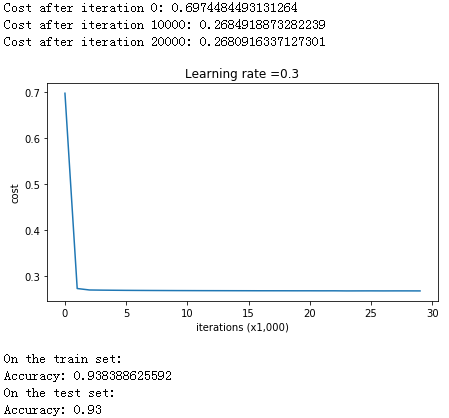
避免过度拟合的标准方法称为L2正则化，它包括适当修改你的成本函数，我们从原来的成本函数(1)到现在的函数(2)：

 (1)

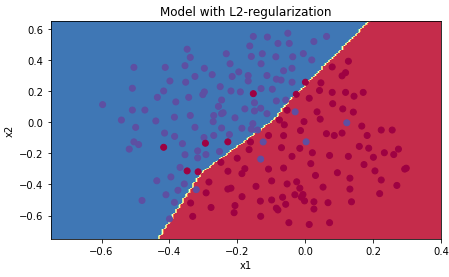
 (2)

计算的代码为：np.sum(np.square(Wl))。

需要注意的是在前向传播中我们对 ，和这三个项执行此操作，将这三个项相加并乘以 。实现反向传播所需的更改以考虑正则化，这些更改只涉及dW1、dW2和dW3，对于每一项必须添加正则项的梯度。



经过预测，得到训练集的准确度为93.84%，测试集的准确度为93%。



结论：

-λ的值是一个超参数，可以使用dev集进行优化。

-L2正则化使决策边界更平滑。如果λ太大，可能会“过度平滑”，从而导致模型具有高偏差。

L2正则化到底在做什么？L2正则化依赖于小权重模型比大权重模型简单的假设。因此，通过削减成本函数中权重的平方值，可以将所有权重值逐渐改变到到较小的值。它因代价太大而不能有大的权重，所以会有更平滑的模型，其中输入变化时输出变化更慢，但是需要花费更多的时间。

L2正则化对以下内容有影响：

-成本计算：在代价函数中增加了一个正则化项。

-反向传播函数：在权重矩阵方面，梯度计算时也要依据正则化来做出相应的计算，梯度中增加了额外的项。

-权重最终变小（“权重衰减”）：权重被逐渐改变到较小的值。

**3.3 dropout（随机节点删除）**

最后，我们使用Dropout来进行正则化，Dropout的原理就是每次迭代过程中随机将其中的一些节点失效。当我们关闭一些节点时，我们实际上修改了我们的模型。背后的想法是，在每次迭代时，我们都会训练一个只使用一部分神经元的不同模型。随着迭代次数的增加，我们的模型的节点会对其他特定节点的激活变得不那么敏感，因为其他节点可能在任何时候会失效。

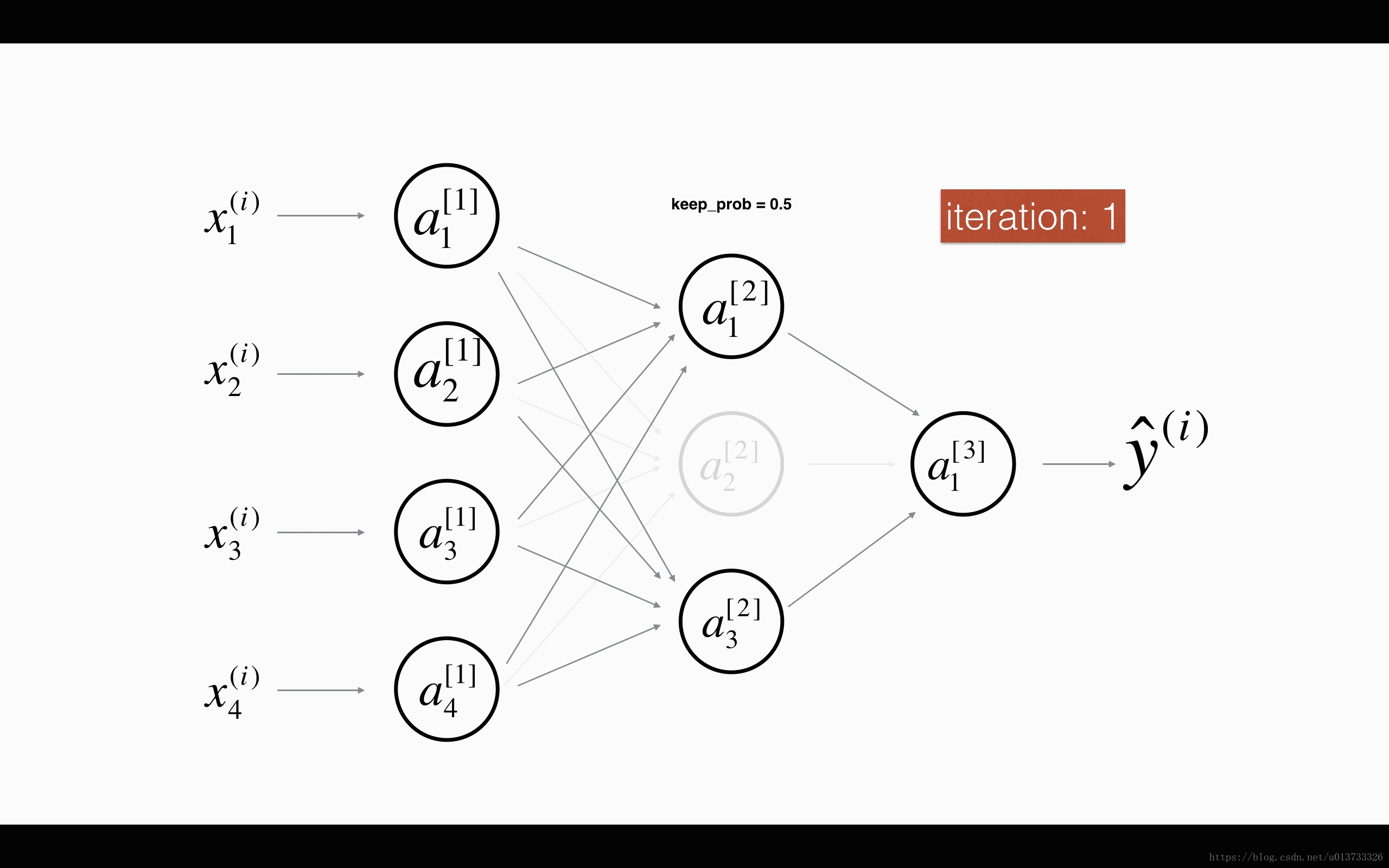


图1 ： 第二层启用随机节点删除。在每一次迭代中，关闭（设置为零）一层的每个神经元，概率为1 - keep\_prob，我们在这里保持概率为keep\_prob（这里为50％）。丢弃的节点都不参与迭代时的前向和后向传播。

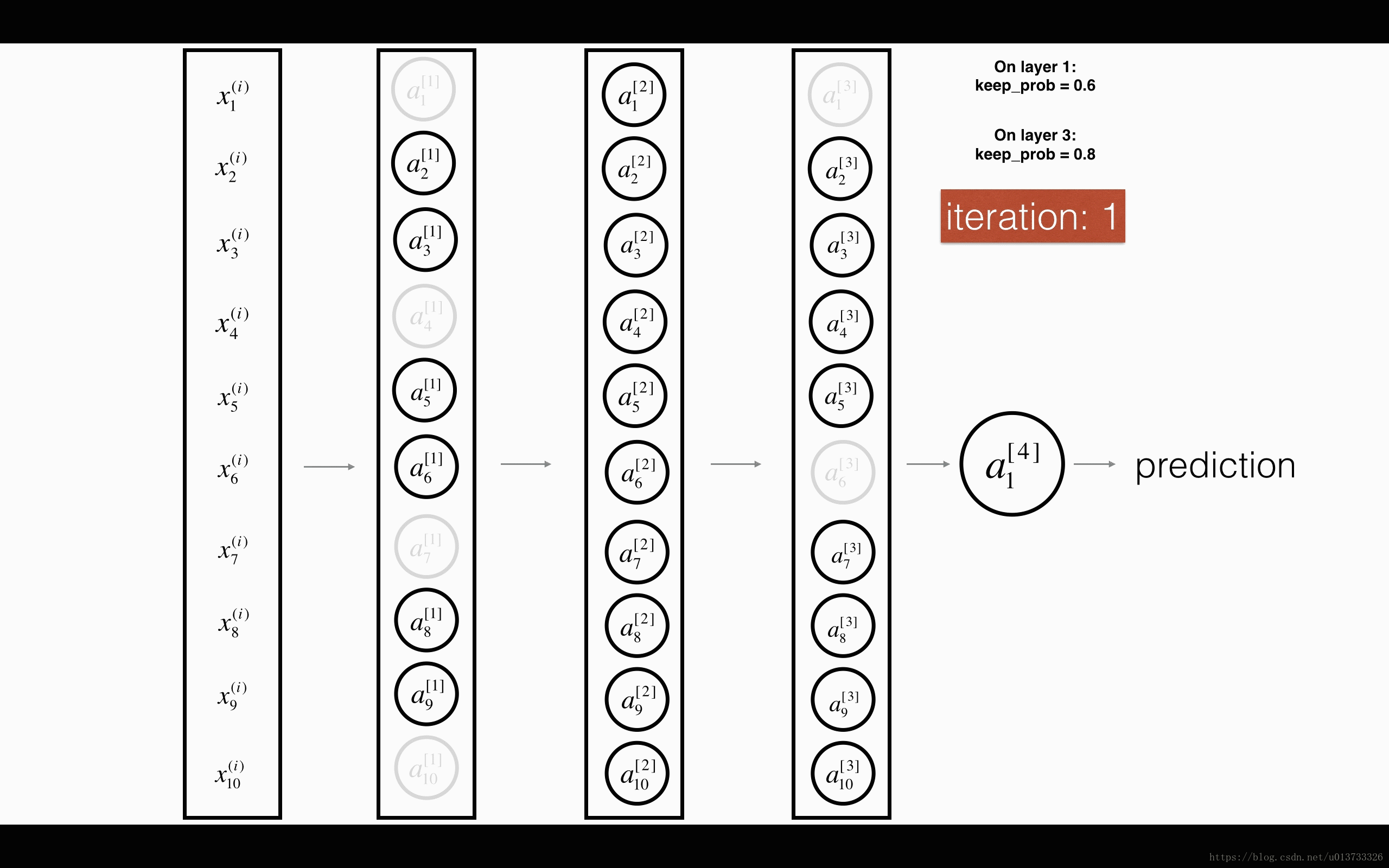


图2 ：在第一层和第三层启用随机删除。第一层平均40%节点被删除， 第三层平均删除了20%的节点 。

* 前向传播

下面我们将关闭第一层和第三层的一些节点，我们需要做以下四步：

Step1: 在视频中，吴恩达老师讲解了使用np.random.rand() 来初始化和具有相同维度的，在这里，我们将使用向量化实现，我们先来实现一个和有相同维度的随机矩阵

Step2: 如果低于 (keep\_prob)的值我们就把它设置为1，如果高于(keep\_prob)的值我们就设置为0。注意0和1分别等价于False和True。{通过对中的值进行适当的阈值化，将中的每个条目设置为0（概率为1-keep\_prob）或1（概率为keep\_prob）。提示：要将矩阵X的所有项设置为0（如果项大于0.5）或1（如果项小于0.5），您需要执行：X=（X<0.5）。注意0和1分别等价于False和True。}

Step3: 把更新为。 (我们已经关闭了一些节点)。我们可以使用 作为掩码。我们做矩阵相乘的时候，关闭的那些节点（值为0）就会不参与计算，因为0乘以任何值都为0。

Step4: 使用除以 keep\_prob。这样做的话我们通过缩放就在计算成本的时候仍然具有相同的期望值，这叫做反向dropout。

* 反向传播

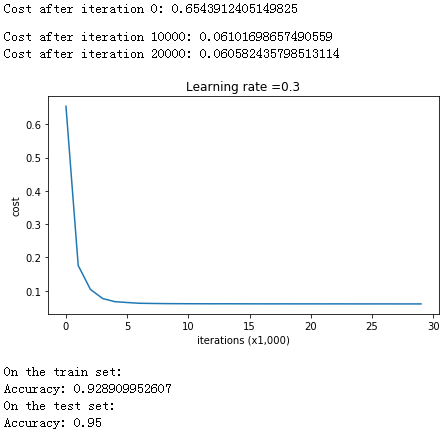
改变了前向传播的算法，我们也需要改变后向传播的算法，使用存储在缓存中的掩码和将舍弃的节点位置信息添加到第一个和第二个隐藏层。

带dropout的反向传播实际上相当容易，我们需要做以下两步：

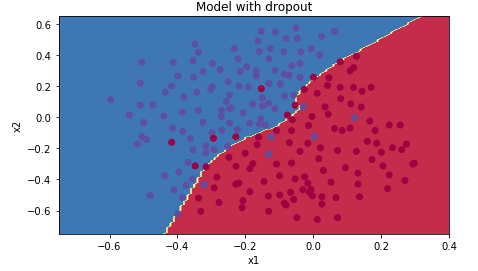
Step1: 在正向传播过程中，通过对应用一个掩码，来关闭一些神经元。在反向传播中，通过把相同的掩码应用到，来关闭相同的神经元。

Step2: 在正向传播过程中，用 keep\_prob除以。因此，在反向传播中，你必须再次用 keep\_prob除以（微积分的解释是，如果被keep\_prob缩放，那么它的导数也被相同的keep\_prob缩放）。

使用dropout运行模型（保持keep\_prob=0.86）。这意味着在每一次迭代中，你以24%的概率关闭第1层和第2层的每个神经元。



经过预测，得到训练集的准确度为92.89%，测试集的准确度提升至95%。我们可以看到，正则化会把训练集的准确度降低，但是测试集的准确度提高了，模型并不会在训练集上过度拟合，并且在测试集上做得很好。



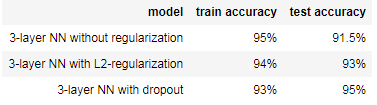
注：使用dropout的一个常见错误是在训练和测试中都使用它，你应该只在训练中使用dropout（随机消除节点）。

像tensorflow、paddle、keras或caffe这样的深度学习框架附带了一个dropout层实现。

关于dropout你应该记住的是：

* dropout是一种正则化技术。
* 只在训练期间使用dropout，在测试期间不要使用dropout（随机消除节点）。
* 在正向和反向传播过程中应用dropout。
* 在训练期间，用keep\_prob除以每个dropout层，以保持激活的预期值相同。例如，如果keep\_prob为0.5，那么我们将平均关闭一半节点，因此输出将按0.5的比例缩放，因为只有剩下的一半对解决方案有贡献。除以0.5等于乘以2，因此现在输出具有相同的预期值。即使keep-prob不是0.5，你也可以检查它是否有效。

对比三种方法，结果如下：



请注意，正则化会损害训练集的性能！这是因为它限制了网络过度拟合训练集的能力。但由于它最终能提供更好的测试精度，所以它有助于我们的系统。

结论：

-正则化将有助于减少过度拟合。

-正则化会使你的权重降低。

-L2正则化和dropout是两种非常有效的正则化技术。

**4、梯度检验**

您是一个团队的一员，该团队致力于在全球范围内提供移动支付服务，并要求您构建一个深入的学习模型来检测欺诈行为——无论何时有人进行支付，您都要查看支付是否可能是欺诈性的，例如用户的帐户是否已被黑客接管。

但是反向传播在实现上是相当具有挑战性的，有时也会有错误。因为这是一个任务关键型应用程序，所以公司的CEO希望真正确定反向传播的实现是正确的。你的首席执行官说，“给我一个证据，证明你的反向传播确实有效！”为了保证这一点，您将使用“梯度检验”。

也就是说，假设你现在是一个全球移动支付团队中的一员，现在需要建立一个深度学习模型去判断用户账户在进行付款的时候是否是被黑客入侵的。但是，在我们执行反向传播的计算过程中，反向传播函数的计算过程是比较复杂的。为了验证我们得到的反向传播函数是否正确，现在你需要编写一些代码来验证反向传播函数的正确性。

**4.1 梯度检验是如何工作的？**

反向传播计算梯度，θ表示模型中的参数，使用前向传播和损失函数计算 J，因为前向传播相对容易实现，所以可以确信自己得到了正确的结果，因此几乎100％确定正确计算了J的代价。因此，我们可以使用计算J的代码来验证计算的代码。

让我们回头看一下导数（或梯度）的定义：

 (3)

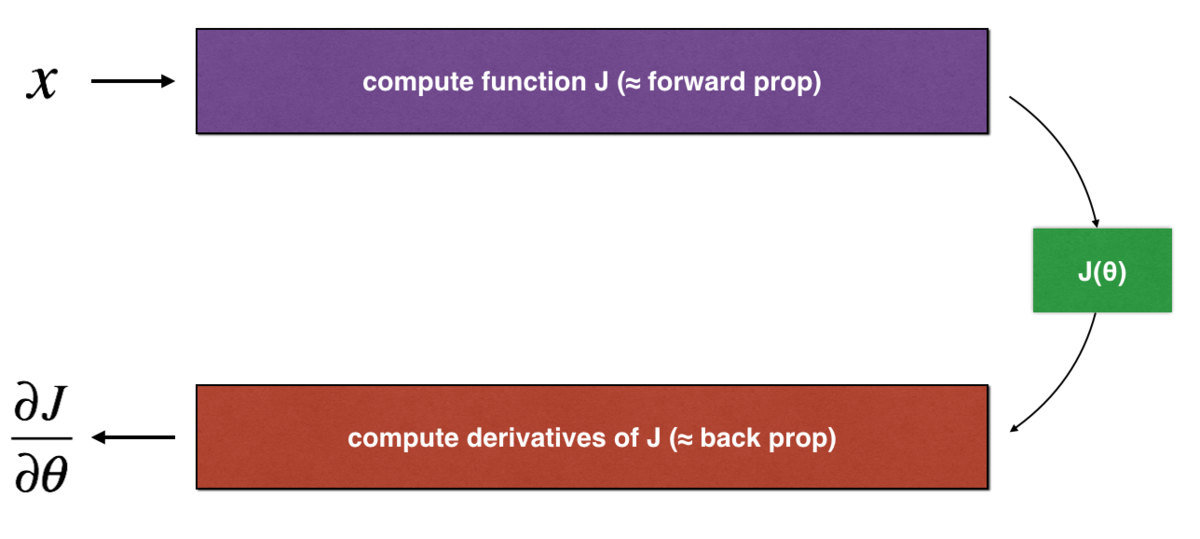
-是要确保计算正确的值。

-你可以计算 J(θ+ε)和J(θ−ε)（在θ是一个实数的情况下），因为你确信你对J的实现是正确的。

使用等式（3）和一个小的ε值来说服你的CEO你的计算的代码是正确的！

**4.2 一维梯度检验**

我们先来看一下一维线性模型的梯度检验计算过程：



梯度检查的步骤如下：



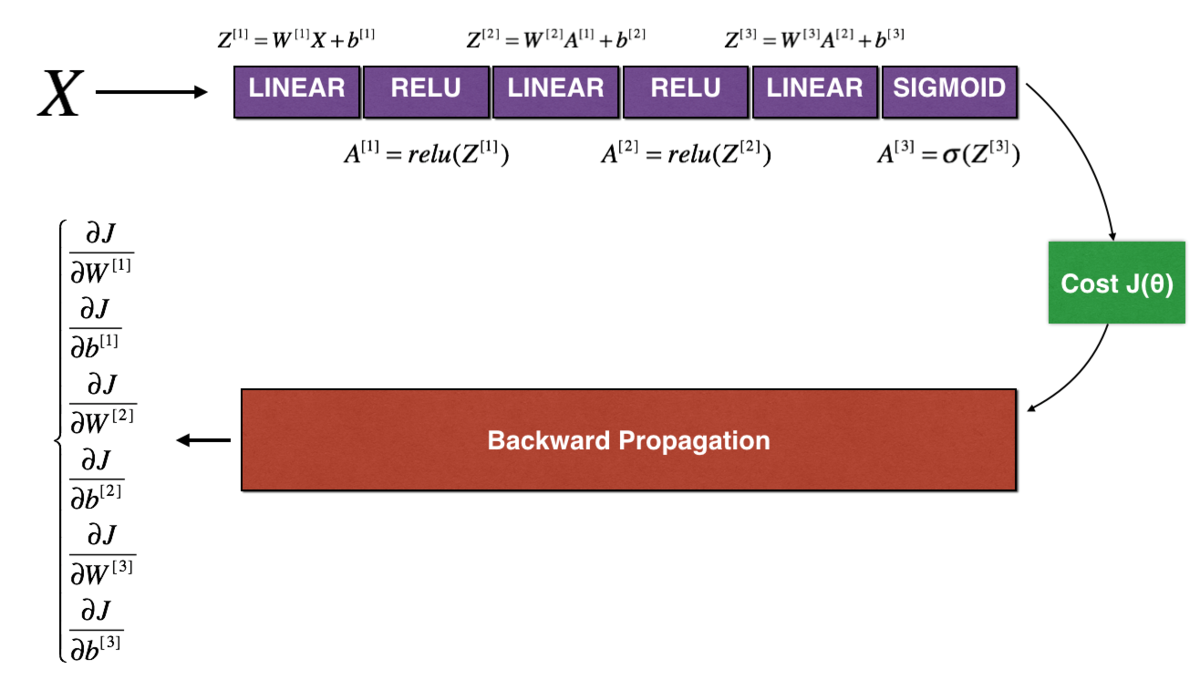
接下来，计算梯度的反向传播值，最后计算误差：

 (4)

当difference小于时，我们通常认为我们计算的结果是正确的。

**4.3 N维梯度检验**

高维参数是怎样计算的呢？我们看一下下图：



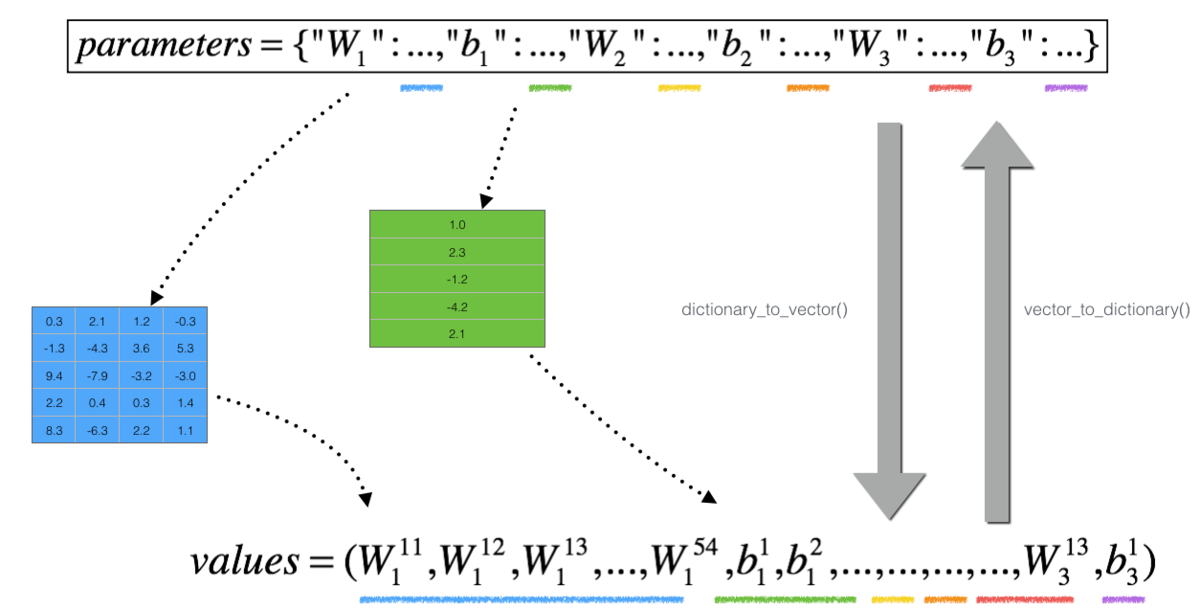
深度神经网络：LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

您在欺诈检测测试集上获得了一些结果，但您并不是100%确定您的模型。没有人是完美的！让我们实现渐变检查来验证渐变是否正确。

如果想比较“gradapprox”与反向传播计算的梯度。该公式仍然是：



然而，θ不再是标量，这是一个名为“parameters”的字典。 我们为你实现了一个函数“dictionary\_to\_vector（）”，它将“parameters”字典转换为一个称为“values”的向量，该向量是通过将所有参数（W1，b1，W2，b2，W3，b3）重塑为向量并将它们连接起来而获得。反函数是“vector\_to\_dictionary”，它返回“parameters”字典。



我们还使用gradients\_to\_vector（）将“gradients”字典转换为向量“grad”。

下面是帮助您实现梯度检验的伪代码。

For i in num\_parameters:

-计算 J\_plus[i]:

1.把设置为 np.copy(parameters\_values)

2.把设置为

3.使用 forward\_propagation\_n(x, y, vector\_to\_dictionary())来计算

-计算 J\_minus[i]: 使用相同的方法计算

-计算

-计算梯度

-计算误差：

测试gradient\_check\_n（）时得到的结果为“梯度检查：梯度超出阈值!

difference = 1.1885552035482147e-07”，发现我们给你的反向传播代码似乎有错误！很好，你已经实现了梯度检查。返回到反向传播并尝试查找/更正错误（提示：检查dW2和db1）。当你认为你已经修复了梯度检查时，重新运行它。请记住，如果修改代码，则需要重新执行定义反向传播的单元格。

注：梯度检验很慢！用来近似梯度计算成本很高。因此，在训练过程中，我们不会每次迭代都运行梯度检验，只需几次检查梯度是否正确。

至少我们已经介绍过梯度检验不适用于dropout。你通常会在没有dropout的时候运行梯度检验算法，以确保你的反向是正确的，然后添加dropout。

恭喜您可以确信您的欺诈检测深度学习模型工作正常！你甚至可以用这个来说服你的CEO。

你应该记住的是：

-梯度检验验证了反向传播梯度（反向传播过程中计算的导数）和梯度的数值近似（使用正向传播计算）之间的距离。

-梯度检验很慢，所以我们不会在每次训练迭代中都运行它。通常运行它只是为了确保代码正确，然后关闭它并在实际的学习过程中使用backprop。