

## 1.4 正则化 (Regularization)

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，这是非常可靠的方法，但你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。**

如果你怀疑神经网络过度拟合了数据，即存在高方差问题，那么最先想到的方法可能是正则化，另一个解决高方差的方法就是准备更多数据，这也是非常可靠的办法，但你可能无法时时准备足够多的训练数据，或者，获取更多数据的成本很高，但正则化有助于避免过度拟合，或者减少网络误差，下面我们就来讲讲正则化的作用原理。

我们用逻辑回归来实现这些设想，**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  $\lambda$ ，也就是正则化参数，一会儿再详细讲。

$\frac{\lambda}{2m}$ 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 $w^T w$ ，也就是向量参数 $w$  的欧几里德范数 (2 范数) 的平方，此方法称为L2正则化。因为这里用了欧几里德法线，被称为向量参数 $w$ 的L2范数。

$$\begin{aligned} & w \in \mathbb{R}^{n_x}, b \in \mathbb{R} \\ & \min_{w,b} J(w,b) \\ & J(w,b) = \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2 \\ & \text{L}_2 \text{ regularization} \quad \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \end{aligned}$$

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，只是我习惯省略不写，因为 $w$ 通常是一个高维参数矢量，已经可以表达高偏差问题， $w$ 可能包含有很多参数，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，而不是 $b$ ，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。

$$J(w, b) = \frac{1}{n} \sum_{i=1}^n L(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2n} \|w\|_2^2$$

$L2$ 正则化是最常见的正则化类型，你们可能听说过 $L1$ 正则化， $L1$ 正则化，加的不是 $L2$ 范数，而是正则项 $\frac{\lambda}{m}$ 乘以 $\sum_{j=1}^{n_x} |w_j|$ ， $\sum_{j=1}^{n_x} |w_j|$ 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。

$$L_1 \text{ regularization} \quad \frac{\lambda}{2m} \sum_{j=1}^n |w_j| = \frac{\lambda}{2m} \|w\|_1 \quad w \text{ will be sparse}$$

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， $\lambda$ 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 $\lambda$ 是另外一个需要调整的超级参数，顺便说一下，为了方便写代码，在 Python 编程语言中， $\lambda$ 是一个保留字段，编写代码时，我们删掉 $\lambda$ ，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？

# Logistic regression

$\min_{w,b} J(w,b)$

$\underline{w} \in \mathbb{R}^{n_x}, \underline{b} \in \mathbb{R}$

$\lambda = \text{regularization parameter}$   
 $\text{lambda}$

$J(w,b) = \frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \|\underline{w}\|_2^2$

$\cancel{+ \frac{\lambda}{2m} b^2}$   
 $\text{omit}$

$L_2 \text{ regularization}$

$\underline{\|\underline{w}\|_2^2} = \sum_{j=1}^{n_x} w_j^2 = \underline{w^T w} \leftarrow$

$L_1 \text{ regularization}$

$\frac{\lambda}{2m} \sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m} \|\underline{w}\|_1$

$w \text{ will be sparse}$

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



$$dW^{(2)} = (\text{from backprop}) + \frac{\lambda}{n} W^{(2)}$$

$$\rightarrow W^{(2)} := W^{(2)} - \alpha dW^{(2)}$$

"Weight decay"

$$\frac{\partial J}{\partial W^{(2)}} = dW^{(2)}$$

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  $\alpha$  乘以 **backprop** 再加上  $\frac{\lambda}{n} W^{[l]}$ 。

$$W^{(2)} := W^{(2)} - \alpha \left[ (\text{from backprop}) + \frac{\lambda}{n} W^{(2)} \right]$$

$$= W^{(2)} - \frac{\alpha \lambda}{n} W^{(2)} - \alpha (\text{from backprop})$$

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  $(1 - \alpha \frac{\lambda}{n})$  倍的权重，矩阵  $W$  减去  $\alpha \frac{\lambda}{n}$  倍的它，也就是用这个系数  $(1 - \alpha \frac{\lambda}{n})$  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  $\alpha$  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”。

$$W^{(2)} := W^{(2)} - \alpha \left[ (\text{from backprop}) + \frac{\lambda}{n} W^{(2)} \right]$$

$$= W^{(2)} - \frac{\alpha \lambda}{n} W^{(2)} - \alpha (\text{from backprop})$$

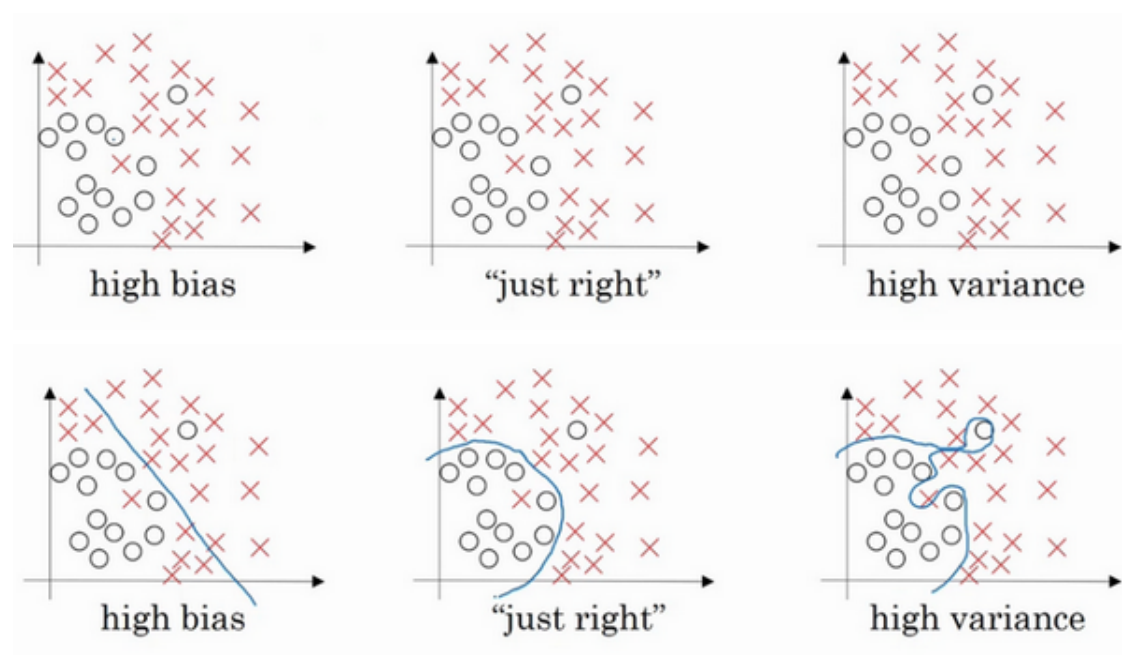
$$= \underbrace{\left(1 - \frac{\alpha \lambda}{n}\right)}_{\leq 1} \underline{W^{(2)}} - \alpha (\text{from backprop})$$

我不打算这么叫它，之所以叫它“权重衰减”是因为这两项相等，权重指标乘以了一个小于 1 的系数。

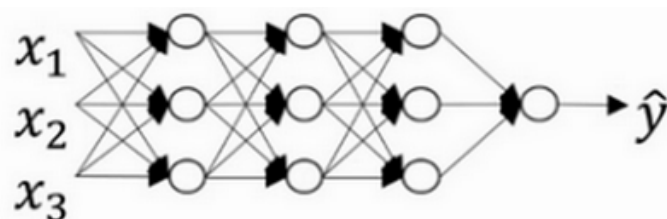
以上就是在神经网络中应用 L2 正则化的过程，有人会问我，为什么正则化可以预防过拟合，我们放在下节课讲，同时直观感受一下正则化是如何预防过拟合的。

## 1.5 为什么正则化有利于预防过拟合呢？（Why regularization reduces overfitting?）

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。



左图是高偏差，右图是高方差，中间是 **Just Right**，这几张图我们在前面课程中看到过。



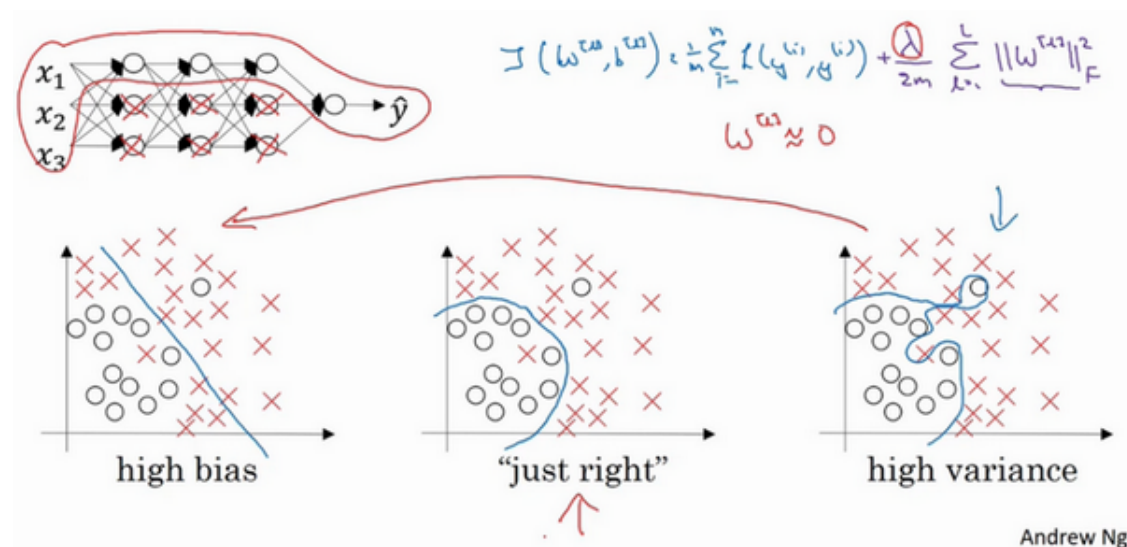
现在我们来看下这个庞大的深度拟合神经网络。我知道这张图不够大，深度也不够，但你可以想象这是一个过拟合的神经网络。这是我们的代价函数 $J$ ，含有参数 $W$ ， $b$ 。我们添加正则项，它可以避免数据权值矩阵过大，这就是弗罗贝尼乌斯范数，为什么压缩 $L2$ 范数，或者弗罗贝尼乌斯范数或者参数可以减少过拟合？

直观上理解就是如果正则化 $\lambda$ 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左图的高偏差状态。

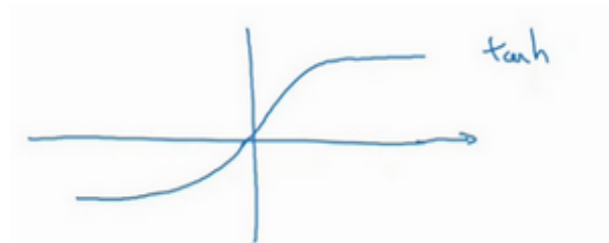
但是 $\lambda$ 会存在一个中间值，于是会有一个接近“**Just Right**”的中间状态。



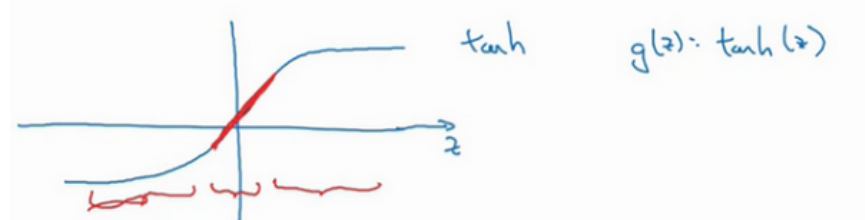
直观理解就是 $\lambda$ 增加到足够大， $W$ 会接近于 0，实际上是不会发生这种情况的，我们尝试消除或至少减少许多隐藏单元的影响，最终这个网络会变得更简单，这个神经网络越来越接近逻辑回归，我们直觉上认为大量隐藏单元被完全消除了，其实不然，实际上是该神经网络的所有隐藏单元依然存在，但是它们的影响变得更小了。神经网络变得更简单了，貌似这样更不容易发生过拟合，因此我不确定这个直觉经验是否有用，不过在编程中执行正则化时，你实际看到一些方差减少的结果。



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。



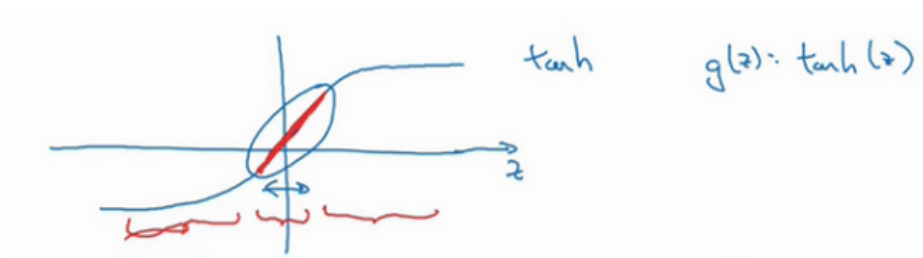
现在你应该摒弃这个直觉，如果正则化参数 $\lambda$ 很大，激活函数的参数会相对较小，因为代价函数中的参数变大了，

$$z^{[n]} = W^{[n]} a^{[n-1]} + b^{[n]}$$

如果 $W$ 很小，相对来说， $z$ 也会很小。

$$\lambda \uparrow \quad W^{[n]} \downarrow \quad z^{[n]} = W^{[n]} a^{[n-1]} + b^{[n]}$$

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，如同我们在幻灯片中看到的过度拟合高方差的情况。

$$\lambda \uparrow \quad W^{[n]} \downarrow \quad z^{[n]} = W^{[n]} a^{[n-1]} + b^{[n]}$$

Every layer is linear.

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。

$$J(\dots) = \underbrace{\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}_{\text{loss}} + \underbrace{\frac{\lambda}{2} \sum_l \|W^{[l]}\|_F^2}_{\text{regularization}}$$

如果你使用的是梯度下降函数，在调试梯度下降时，其中一步就是把代价函数 $J$ 设计成这样一个函数，在调试梯度下降时，它代表梯度下降的调幅数量。可以看到，[代价函数对于](#)

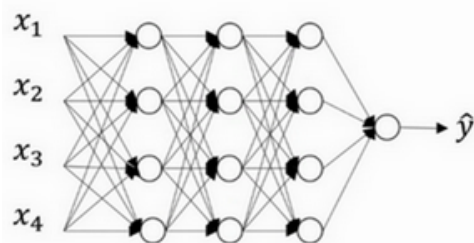
梯度下降的每个调幅都单调递减。如果你实施的是正则化函数，请牢记， $J$ 已经有一个全新的定义。如果你用的是原函数 $J$ ，也就是这第一个项正则化项，你可能看不到单调递减现象，为了调试梯度下降，请务必使用新定义的 $J$ 函数，它包含第二个正则化项，否则函数 $J$ 可能不会在所有调幅范围内都单调递减。

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。

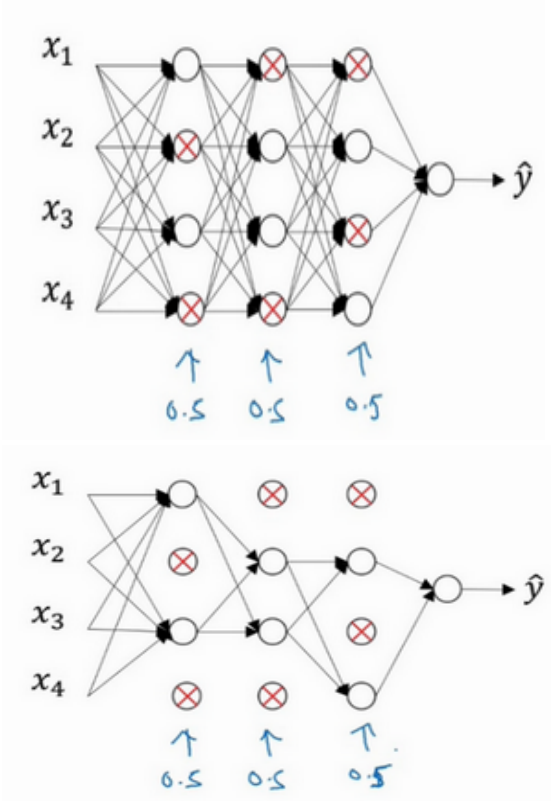


## 1.6 dropout 正则化 (Dropout Regularization)

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。

Illustrate with layer  $l=3$ . keep-prob=0.8 0.2  
$$d3 = \text{np.random.rand}(a3.\text{shape}[0], a3.\text{shape}[1]) < \text{keep-prob}$$
  
a:

如何实施 **dropout** 呢？方法有几种，接下来我要讲的是最常用的方法，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])
```

然后看它是否小于某数，我们称之为 **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，它的作用就是生成随机矩阵，如果对  $a^{[3]}$  进行因子分解，效果也是一样的。 $d^{[3]}$  是一个矩阵，每个样本和每个隐藏单元，其中  $d^{[3]}$  中的对应值为 1 的概率都是 0.8，对应为 0 的概率是 0.2，随机数字小于 0.8。它等于 1 的概率是 0.8，等于 0 的概率是 0.2。

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  $d^{[3]}$ ， $a3 = \text{np.multiply}(a3, d3)$ ，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。

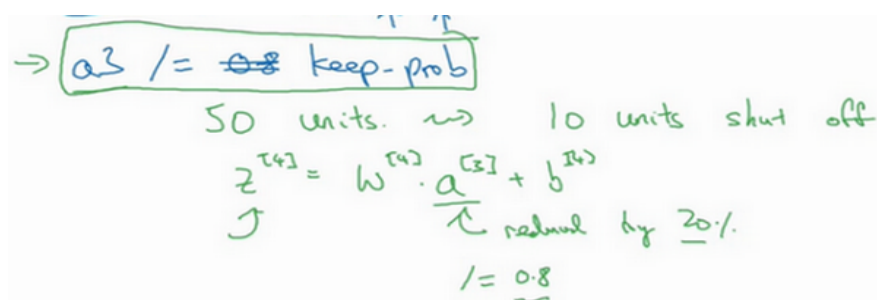
Illustrate with layer  $l=3$ . keep-prob=0.8 0.2  
$$d3 = \text{np.random.rand}(a3.\text{shape}[0], a3.\text{shape}[1]) < \text{keep-prob}$$
  
$$a3 = \text{np.multiply}(a3, d3) \quad \# a3 *= d3.$$

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，大家可以用 **python** 尝试一下。

最后，我们向外扩展 $a^{[3]}$ ，用它除以 0.8，或者除以 **keep-prob** 参数。

$$a^{[3]} /= \text{keep-prob}$$

下面我解释一下为什么要这么做，为方便起见，我们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（ $50 \times 20\% = 10$ ）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 $a^{[3]}$ 中有 20%的元素被归零，**为了不影响 $z^{[4]}$ 的期望值**，我们需要用 $w^{[4]}a^{[3]}/0.8$ ，它将会修正或弥补我们所需的那 20%， $a^{[3]}$ 的期望值不会变，划线部分就是所谓的 **dropout** 方法。



$\rightarrow a^{[3]} /= \text{keep-prob}$

50 units.  $\rightarrow$  10 units shut off

$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$

$\uparrow$   $\uparrow$  reduced by 20%  $\uparrow$

$\quad \quad \quad /= 0.8$

它的功能是，不论 **keep-prop** 的值是多少 0.8, 0.9 甚至是 1，如果 **keep-prop** 设置为 1，那么就不存在 **dropout**，因为它会保留所有节点。**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，因为它的扩展问题变少，我们将在下节课讨论。

现在你使用的是 $d$ 向量，你会发现，不同的训练样本，清除不同的隐藏单元也不同。实际上，如果你通过相同训练集多次传递数据，每次训练数据的梯度不同，则随机对不同隐藏单元归零，有时却并非如此。比如，需要将相同隐藏单元归零，第一次迭代梯度下降时，把一些隐藏单元归零，第二次迭代梯度下降时，也就是第二次遍历训练集时，对不同类型的隐藏层单元归零。向量 $d$ 或 $d^{[3]}$ 用来决定第三层中哪些单元归零，无论用 **foreprop** 还是 **backprop**，这里我们只介绍了 **foreprob**。

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：

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

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

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

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

以此类推直到最后一层，预测值为 $\hat{y}$ 。

Handwritten notes showing the forward pass of a neural network without dropout:

$$a^{[0]} = X$$

No drop out.

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

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

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

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

↓

$$\hat{y}$$

显然在测试阶段，我们并未使用 **dropout**，自然也就不需要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。理论上，你只需要多次运行预测处理过程，每一次，不同的隐藏单元会被随机归零，预测处理遍历它们，但计算效率低，得出的结果也几乎相同，与这个不同程序产生的结果极为相似。

**Inverted dropout** 函数在除以 **keep-prob** 时可以记住上一步的操作，目的是确保即使在测试阶段不执行 **dropout** 来调整数值范围，激活函数的预期结果也不会发生变化，所以没必要在测试阶段额外添加尺度参数，这与训练阶段不同。

$$l = \text{keep} - \text{prob}$$

这就是 **dropout**，大家可以通过本周的编程练习来执行这个函数，亲身实践一下。

为什么 **dropout** 会起作用呢？下节课我们将更加直观地了解 **dropout** 的具体功能。

## 1.7 理解 dropout (Understanding Dropout)

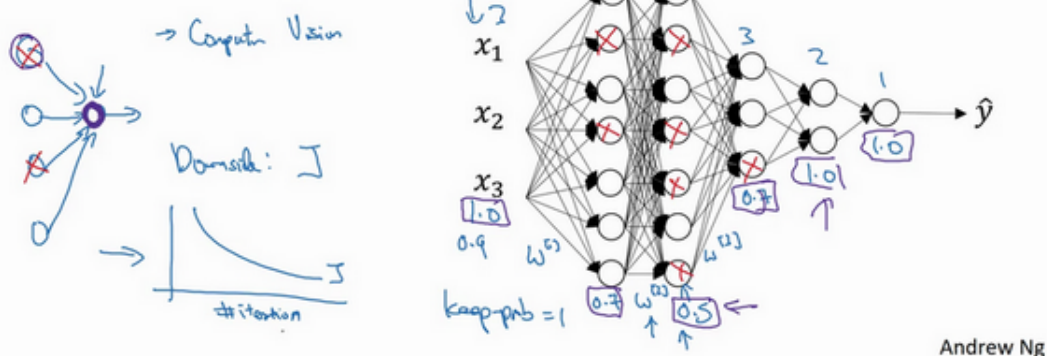
**Dropout** 可以随机删除网络中的神经单元，他为什么可以通过正则化发挥如此大的作用呢？

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。

### Why does drop-out work?

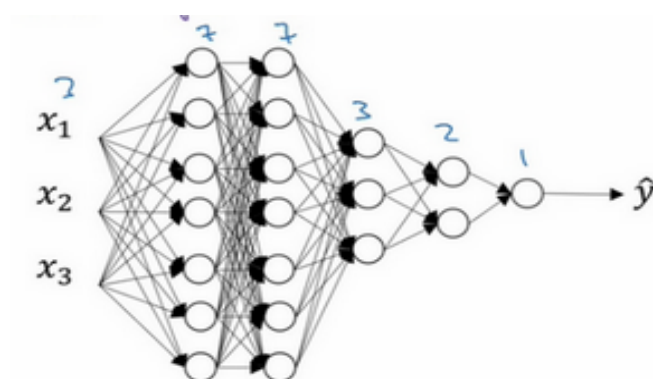
Intuition: Can't rely on any one feature, so have to spread out weights.  $\leadsto$  Shrink weights.  $L2$



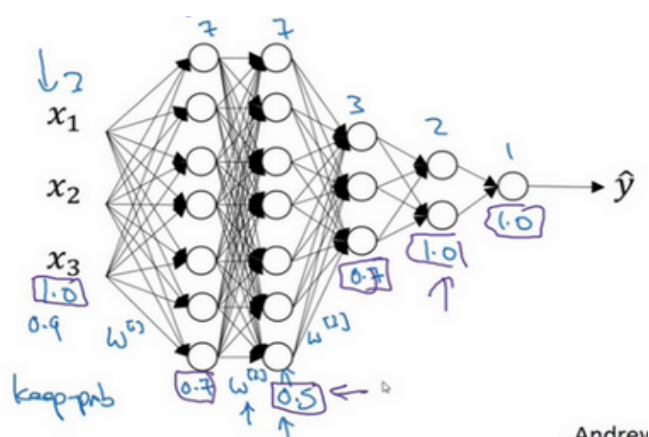
第二个直观认识是，我们从单个神经元入手，如图，这个单元的工作就是输入并生成一些有意义的输出。通过 **dropout**，该单元的输入几乎被消除，有时这两个单元会被删除，有时会删除其它单元，就是说，我用紫色圈起来的这个单元，它不能依靠任何特征，因为特征都有可能被随机清除，或者说该单元的输入也都有可能被随机清除。我不愿意把所有赌注都放在一个节点上，不愿意给任何一个输入加上太多权重，因为它可能会被删除，因此该单元将通过这种方式积极地传播开，并为单元的四个输入增加一点权重，通过传播所有权重，**dropout** 将产生收缩权重的平方范数的效果，和我们之前讲过的 $L2$ 正则化类似，实施 **dropout** 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  $7 \times 3$ ，第二个权重矩阵  $W^{[2]}$  是  $7 \times 7$ ，第三个权重矩阵  $W^{[3]}$  是  $3 \times 7$ ，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  $7 \times 7$ ，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。



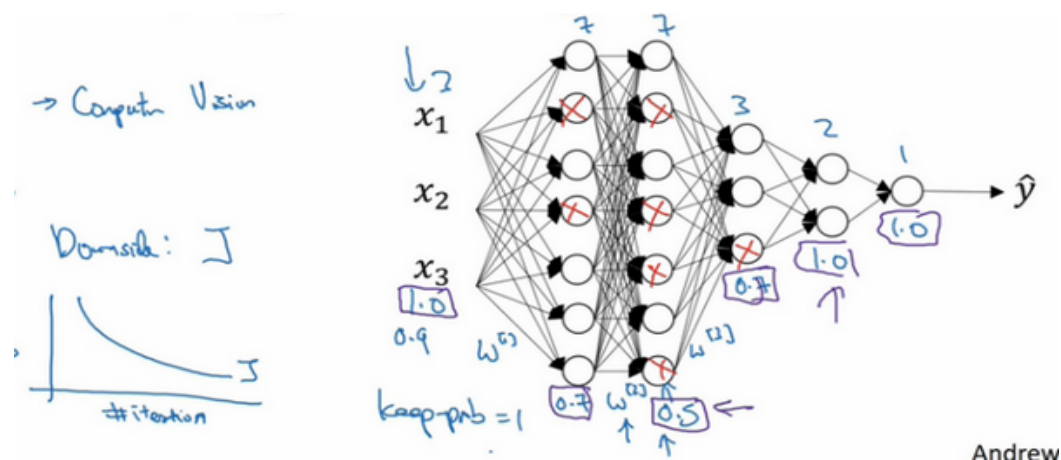
注意 **keep-prob** 的值是 1，意味着保留所有单元，并且不在这一层使用 **dropout**，对于有可能出现过拟合，且含有诸多参数的层，我们可以把 **keep-prob** 设置成比较小的值，以便应用更强大的 **dropout**，有点像在处理 $L2$ 正则化的正则化参数 $\lambda$ ，我们尝试对某些层施行更多正则化，从技术上讲，我们也可以对输入层应用 **dropout**，我们有机会删除一个或多个输



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**。

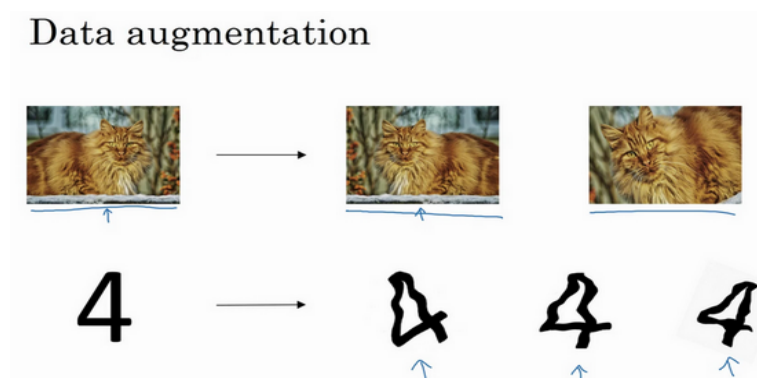
结束前分享两个实施过程中的技巧，实施 **dropout**，在计算机视觉领域有很多成功的第一次。计算视觉中的输入量非常大，输入太多像素，以至于没有足够的数据，所以 **dropout** 在计算机视觉中应用得比较频繁，有些计算机视觉研究人员非常喜欢用它，几乎成了默认的选择，但要牢记一点，**dropout** 是一种正则化方法，它有助于预防过拟合，因此除非算法过拟合，不然我是不会使用 **dropout** 的，所以它在其它领域应用得比较少，主要存在于计算机视觉领域，因为我们通常没有足够的数据，所以一直存在过拟合，这就是有些计算机视觉研究人员如此钟情于 **dropout** 函数的原因。直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（Other regularization methods）

除了 $L2$ 正则化和随机失活（**dropout**）正则化，还有几种方法可以减少神经网络中的过拟合：



### 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，而且有时候我们无法扩增数据，但我们可以通过添加这类图片来增加训练集。例如，水平翻转图片，并把它添加到训练集。所以现在训练集中有原图，还有翻转后的这张图片，所以通过水平翻转图片，训练集则可以增大一倍，因为训练集有冗余，这虽然不如我们额外收集一组新图片那么好，但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，你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。

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。

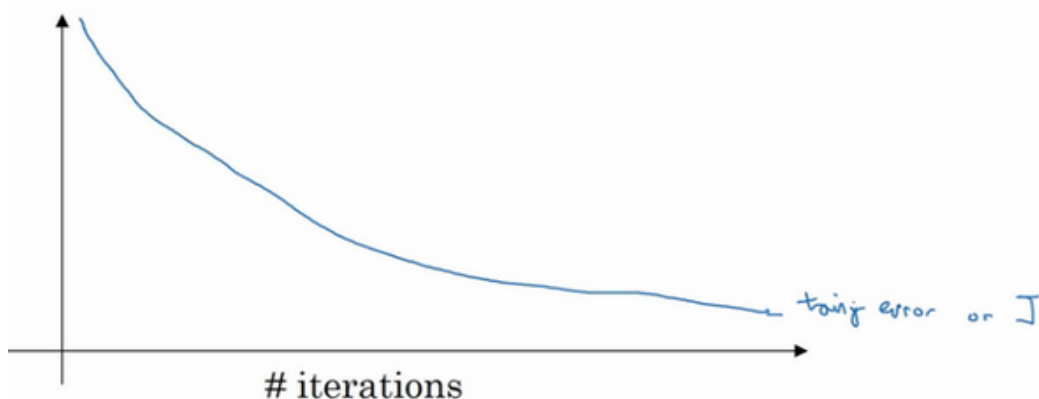


像这样人工合成数据的话，我们要通过算法验证，图片中的猫经过水平翻转之后依然是猫。大家注意，我并没有垂直翻转，因为我们不想上下颠倒图片，也可以随机选取放大后的部分图片，猫可能还在上面。

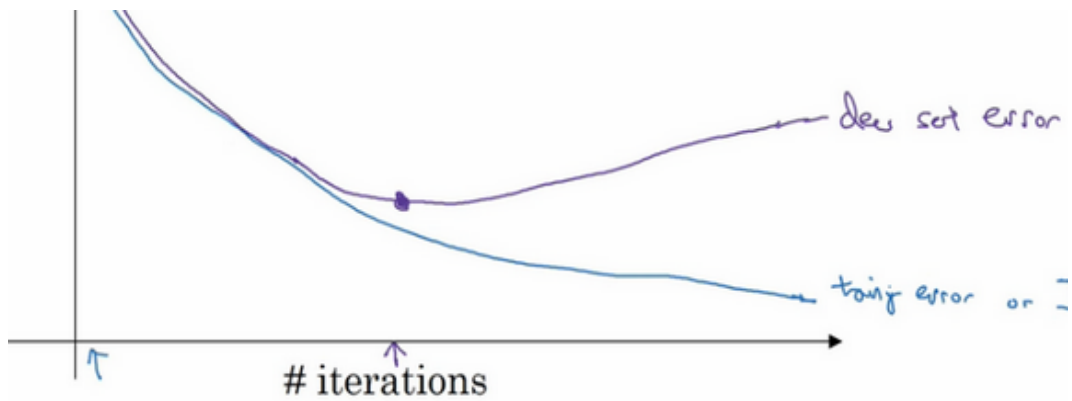
对于光学字符识别，我们还可以通过添加数字，随意旋转或扭曲数字来扩增数据，把这些数字添加到训练集，它们仍然是数字。为了方便说明，我对字符做了强变形处理，所以数字 4 看起来是波形的，其实不用对数字 4 做这么夸张的扭曲，只要轻微的变形就好，我做成这样是为了让大家看的更清楚。实际操作的时候，我们通常对字符做更轻微的变形处理。因为这几个 4 看起来有点扭曲。所以，数据扩增可作为正则化方法使用，实际功能上也与正则化相似。

## 二.early stopping

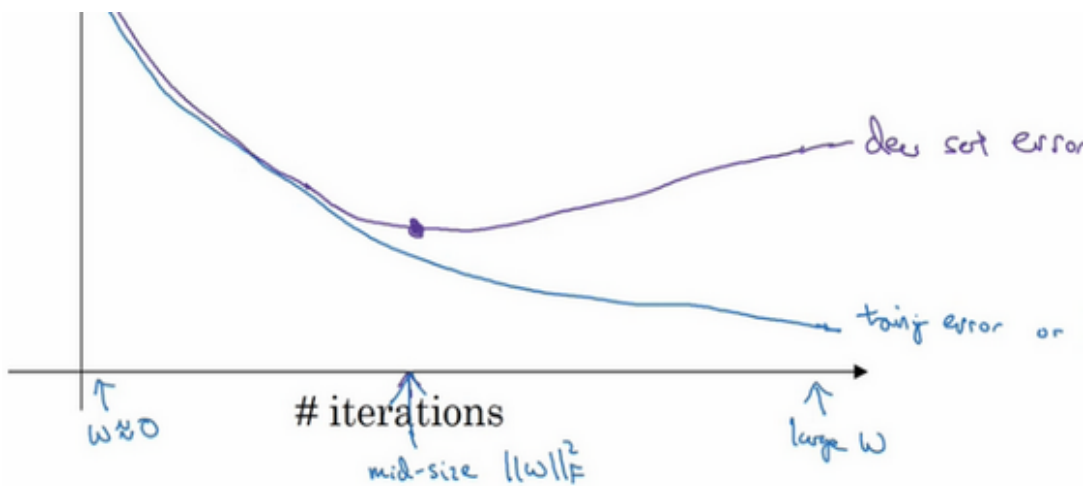
还有另外一种常用的方法叫作 **early stopping**，运行梯度下降时，我们可以绘制训练误差，或只绘制代价函数 $J$ 的优化过程，在训练集上用 0-1 记录分类误差次数。呈单调下降趋势，如图。



因为在训练过程中，我们希望训练误差，代价函数 $J$ 都在下降，通过 **early stopping**，我们不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，你会说，神经网络已经在这个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？



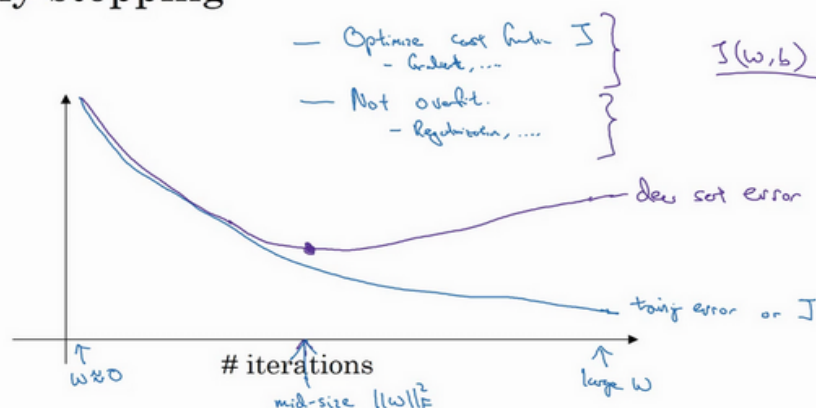
当你还未在神经网络上运行太多迭代过程的时候，参数 $w$ 接近 0，因为随机初始化 $w$ 值时，它的值可能都是较小的随机值，所以在你长期训练神经网络之前 $w$ 依然很小，在迭代过程和训练过程中 $w$ 的值会变得越来越大大，比如在这儿，神经网络中参数 $w$ 的值已经非常大了，所以 **early stopping** 要做就是在中间点停止迭代过程，我们得到一个 $w$ 值中等大小的弗罗贝尼乌斯范数，与 $L2$ 正则化相似，选择参数 $w$ 范数较小的神经网络，但愿你的神经网络过度拟合不严重。



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**early stopping**，但是它也有一个缺点，我们来了解一下。

我认为机器学习过程包括几个步骤，其中一步是选择一个算法来优化代价函数 $J$ ，我们有很多工具来解决这个问题，如梯度下降，后面我会介绍其它算法，例如 **Momentum**，**RMSprop** 和 **Adam** 等等，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。

## Early stopping



在机器学习中，超级参数激增，选出可行的算法也变得越来越复杂。我发现，如果我们用一组工具优化代价函数 $J$ ，机器学习就会变得更简单，在重点优化代价函数 $J$ 时，你只需要留意 $w$ 和 $b$ ， $J(w, b)$ 的值越小越好，你只需要想办法减小这个值，其它的不用关注。然后，预防过拟合还有其他任务，换句话说就是减少方差，这一步我们用另外一套工具来实现，这个原理有时被称为“正交化”。思路就是在一个时间做一个任务，后面课上我会具体介绍正交化，如果你还不了解这个概念，不用担心。

但对我来说 **early stopping** 的主要缺点就是你不能独立地处理这两个问题，因为提早停止梯度下降，也就是停止了优化代价函数 $J$ ，因为现在你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，同时你又希望不出现过拟合，你没有采取不同的方式来解决这两个问题，而是用一种方法同时解决两个问题，这样做的结果是我要考虑的东西变得更复杂。

如果不用 **early stopping**，另一种方法就是 $L2$ 正则化，训练神经网络的时间就可能很长。我发现，这导致超级参数搜索空间更容易分解，也更容易搜索，但是缺点在于，你必须尝试很多正则化参数 $\lambda$ 的值，这也导致搜索大量 $\lambda$ 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 $\lambda$ 的很多值。

如果你还不能完全理解这个概念，没关系，下节课我们会详细讲解正交化，这样会更好理解。

虽然 $L2$ 正则化有缺点，可还是有很多人愿意用它。吴恩达老师个人更倾向于使用 $L2$ 正则化，尝试许多不同的 $\lambda$ 值，假设你可以负担大量计算的代价。而使用 **early stopping** 也能得到相似结果，还不用尝试这么多 $\lambda$ 值。

这节课我们讲了如何使用数据扩增，以及如何使用 **early stopping** 降低神经网络中的方差或预防过拟合。