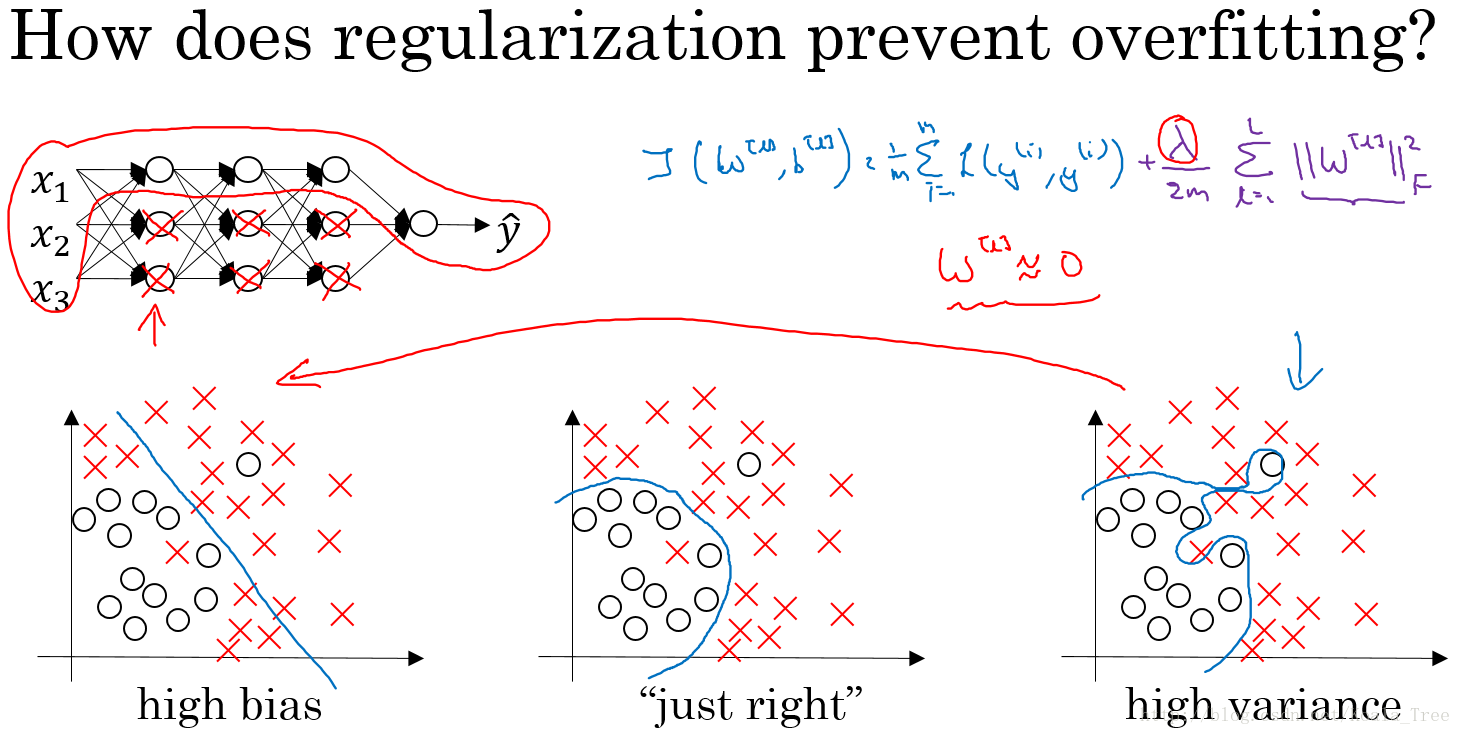
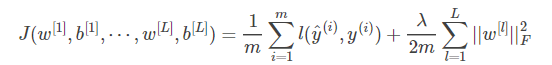
1.5为什么正则化可以减小过拟合

假设下图的神经网络结构属于过拟合状态：



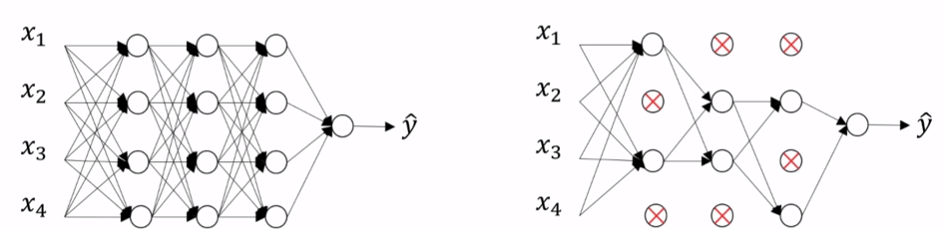
函数：



加入正则化项，直观上理解，正则化因子λ设置的足够大的情况下，为了使代价函数最小化，权重矩阵W就会被设置为接近于0的值。则相当于消除了很多神经元的影响，那么图中的大的神经网络就会变成一个较小的网络。

1.6 Dropout正则化

Dropout会遍历网络的每一层，设置消除神经网络中节点的概率，最后得到一个节点更少、规模更小的网络。



实现Dropout的方法：Inverted dropout（反向随机失活）

定义一个三层的神经网络L=3;

d3表示一个三层的dropout向量； keep-prob=0.8;表示保留某个隐藏单元的概率;

d3 = np.random.rand(a3.shape[0],a3.shape[1])<keep-prob;

a3 =np.multpls(a3,d3);

a3=/keep-prob;

依照例子中的keep\_prob = 0.8，那么就有大约20%的神经元被删除了，也就是说a[3]a[3]中有20%的元素被归零了，在下一层的计算中有Z[4]=W[4]⋅a[3]+b[4]Z[4]=W[4]⋅a[3]+b[4]，所以为了不影响Z[4]Z[4]的期望值，所以需要W[4]⋅a[3]W[4]⋅a[3]的部分除以一个keep\_prob。

Inverted dropout通过对“a3 /= keep\_prob”,则保证无论keep\_prob设置为多少，都不会对Z[4]Z[4]的期望值产生影响。在测试阶段不要用dropout，因为那样会使得预测结果变得随机。

1.7理解 Dropout

以单个神经元入手，单个神经元的工作就是接收输入，并产生一些有意义的输出，但是加入了Dropout以后，输入的特征都是有可能会被随机清除的，所以该神经元不会再特别依赖于任何一个输入特征，也就是说不会给任何一个输入设置太大的权重。

所以通过传播过程，dropout将产生和L2范数相同的收缩权重的效果。对于不同的层，设置的keep\_prob也不同，一般来说神经元较少的层，会设keep\_prob =1.0，神经元多的层，则会将keep\_prob设置的较小。

缺点：

dropout的一大缺点就是其使得 Cost function不能再被明确的定义，以为每次迭代都会随机消除一些神经元结点，所以我们无法绘制出每次迭代J(W,b)下降的图。

使用Dropout：

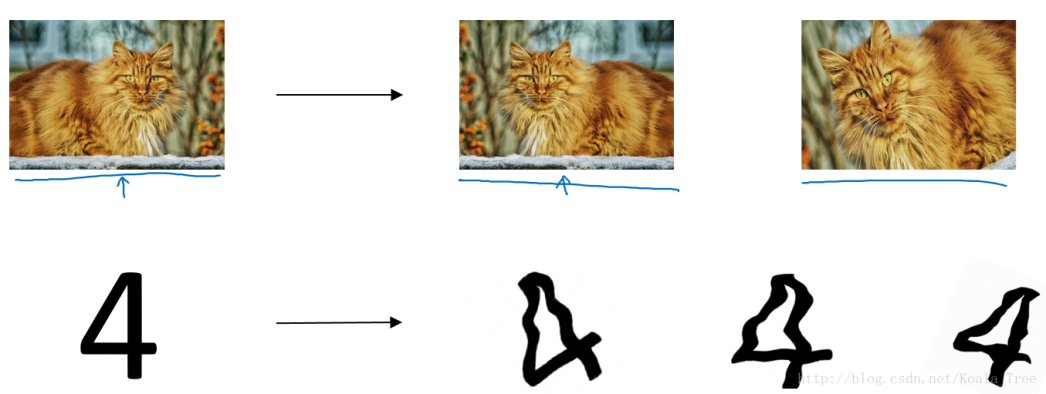
关闭dropout功能，即设置 keep\_prob = 1.0；

运行代码，确保J(W，b)函数单调递减；

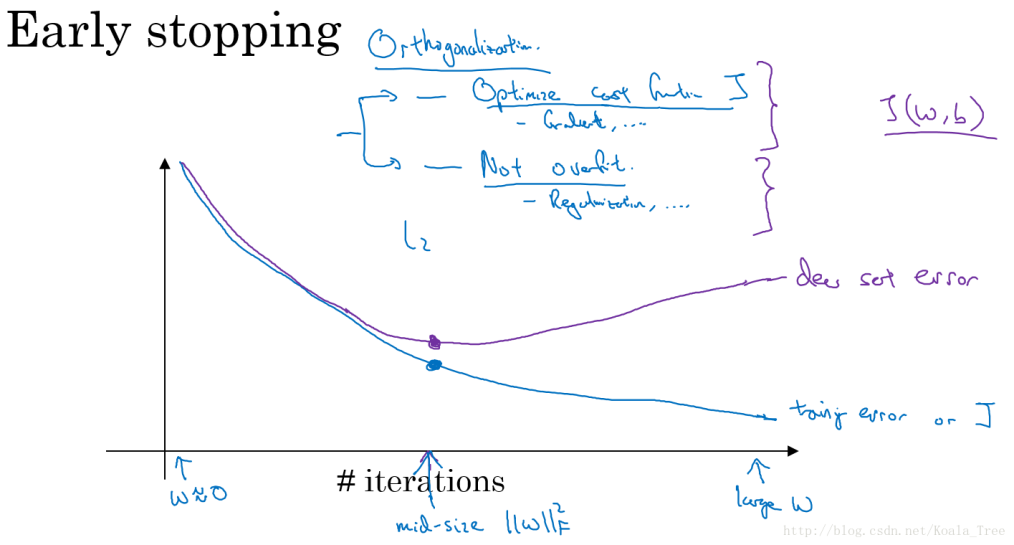
再打开dropout函数。

1.8其他正则化方法

数据扩增（Data augmentation）：通过图片的一些变换，得到更多的训练集和验证集

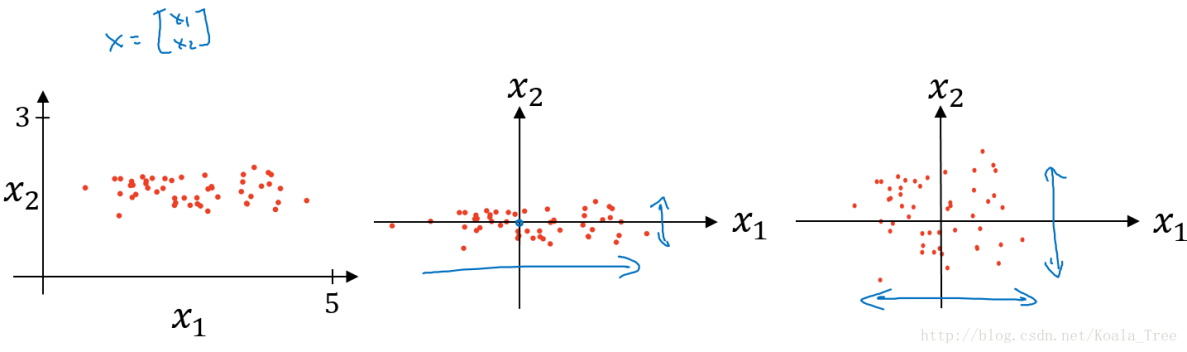


Early stopping：在交叉验证集的误差上升之前的点停止迭代，避免过拟合。这种方法的缺点是无法同时解决bias和variance之间的最优。



1.9 归一化输入

对数据集特征x1,x2归一化的过程：

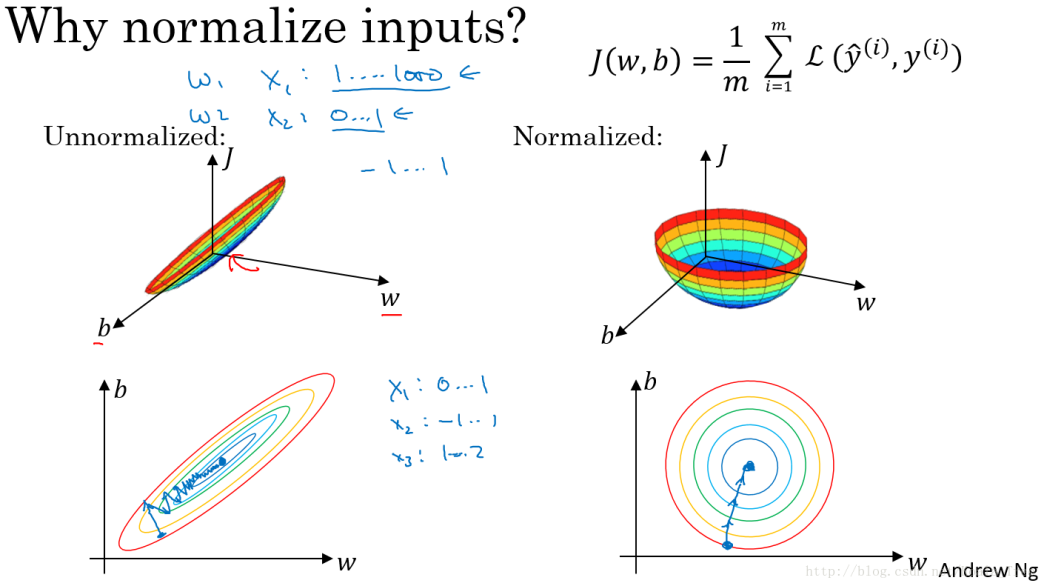


计算每个特征所有样本数据的均值：μ=1m∑i=1mx(i)μ=1m∑i=1mx(i)；

减去均值得到对称的分布：x:=x−μx:=x−μ；

归一化方差：σ2=1m∑i=1mx(i)2σ2=1m∑i=1mx(i)2，x=x/σ2

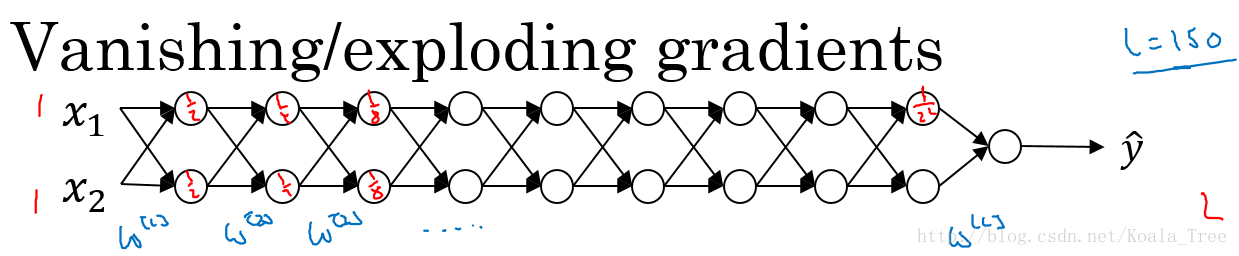
使用归一化的原因：



由图可以看出不使用归一化和使用归一化前后Cost function 的函数形状会有很大的区别。在不使用归一化的代价函数中，如果我们设置一个较小的学习率，那么很可能我们需要很多次迭代才能到达代价函数全局最优解；如果使用了归一化，那么无论从哪个位置开始迭代，我们都能以相对很少的迭代次数找到全局最优解。

1.10 梯度消失和梯度爆炸

如下图所示的神经网络结构，以两个输入为例：



这里我们首先假定g(z)=z，b[l]=0，所以对于目标输出有：

y^=W[L]W[L−1]⋯W[2]W[1]X

y^=W[L]W[L−1]⋯W[2]W[1]X

W[l]W[l]的值大于1的情况：

如：W[l]=[1.5001.5]W[l]=[1.5001.5]，那么最终，y^=W[L][1.5 001.5]L−1Xy^=W[L][1.50 01.5]L−1X，激活函数的值将以指数级递增；

W[l]W[l]的值小于1的情况：

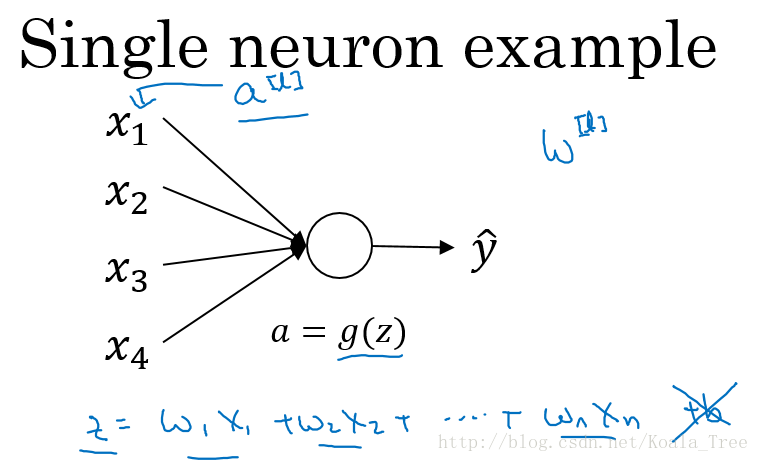
如：W[l]=[0.5000.5]W[l]=[0.5000.5]，那么最终，y^=W[L][0.5 000.5]L−1Xy^=W[L][0.50 00.5]L−1X，激活函数的值将以指数级递减。

上面的情况对于导数也是同样的道理，所以在计算梯度时，根据情况的不同，梯度函数会以指数级递增或者递减，导致训练导数难度上升，梯度下降算法的步长会变得非常非常小，需要训练的时间将会非常长。

在梯度函数上出现的以指数级递增或者递减的情况就分别称为梯度爆炸或者梯度消失。

1.11利用初始化缓解梯度消失和爆炸问题

以一个单个神经元为例子：



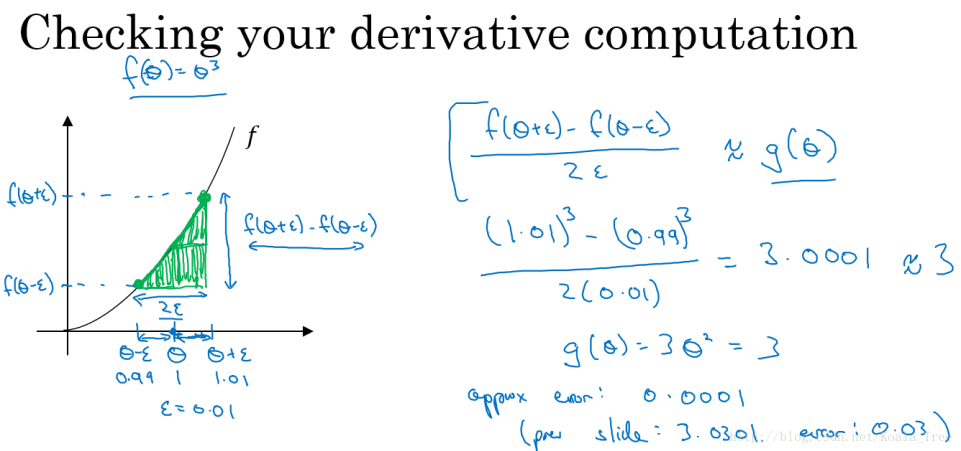
当输入的数量n较大时，我们希望每个wi的值都小一些，这样它们的和得到的z也较小。

这里为了得到较小的wi，设置Var(wi)=1/n .

这么做是因为，如果激活函数的输入x近似设置成均值为0，标准方差1的情况，输出z也会调整到相似的范围内。虽然没有解决梯度消失和爆炸的问题，但其在一定程度上确实减缓了梯度消失和爆炸的速度。

1.12梯度的数值逼近

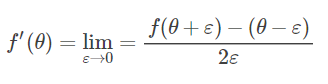
使用双边误差的方法去逼近导数：



由图可以看出，双边误差逼近的误差是0.0001，先比单边逼近的误差0.03，其精度要高了很多。

公式：

双边导数：



误差：O(ε2)

单边导数：

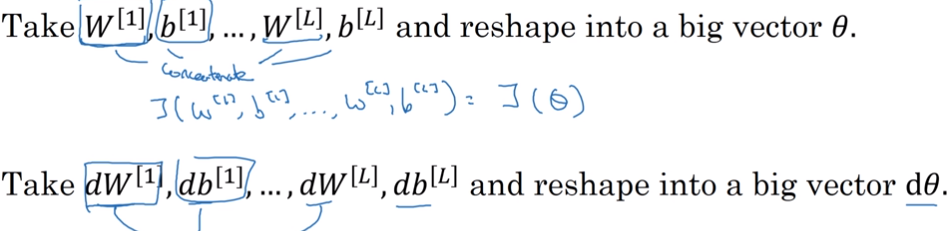


误差：O(ε)

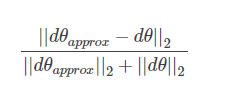
1.13 梯度检验

因为我们的神经网络中含有大量的参数：W[1],b[1],⋯,W[L],b[L]W[1],b[1],⋯,W[L],b[L]，为了做梯度检验，需要将这些参数全部连接起来，reshape成一个大的向量θ。

同时对dW[1],db[1],⋯,dW[L],db[L]dW[1],db[1],⋯,dW[L],db[L]执行同样的操作。



判断dθapprox≈dθ是否接近；



1.14 关于梯度检验实现的注记

①不要在训练过程中使用梯度检验，只在debug的时候使用，使用完毕关闭梯度检验的功能；

②如果算法的梯度检验出现了错误，要检查每一项，找出错误，也就是说要找出哪个dθapprox[i]与dθ的值相差比较大；

③不要忘记了正则化项；

④梯度检验不能与dropout同时使用。因为每次迭代的过程中，dropout会随机消除隐层单元的不同神经元，这时是难以计算dropout在梯度下降上的代价函数J；