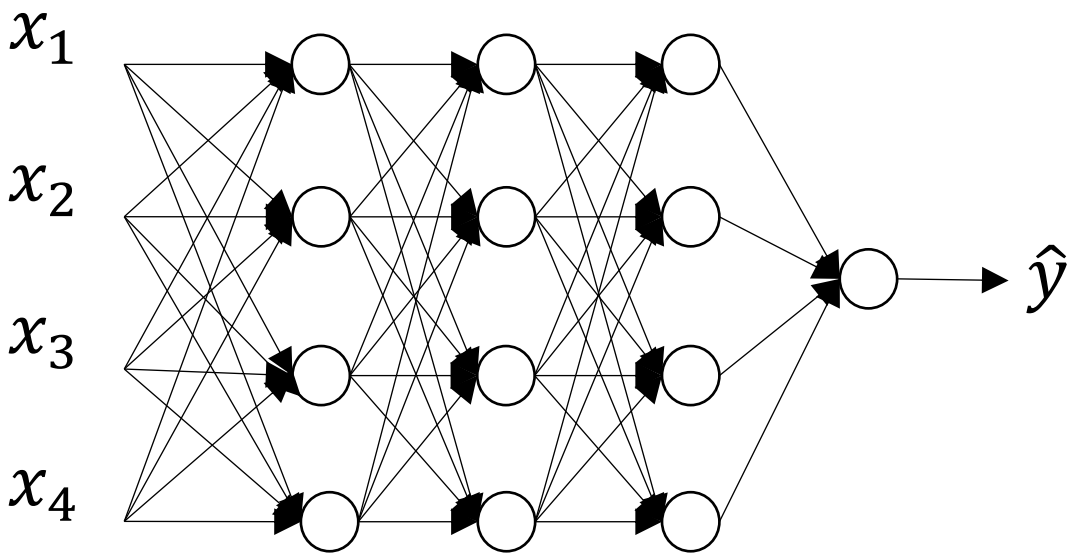
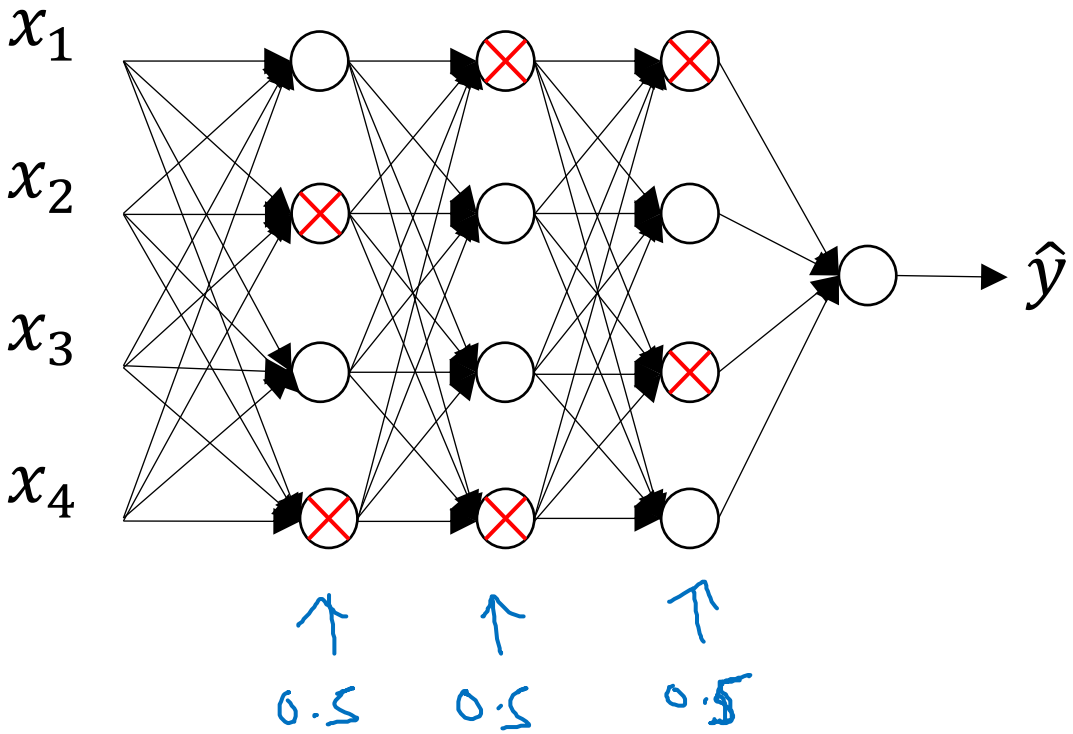
deeplearning.ai

Regularizing your neural network

Dropout regularization

Dropout regularization





Implementing dropout ("Inverted dropout")

Illustrate with layer $l = 3$. keep-prob = 0.8 (0.2)

→ `d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob`

`a3 = np.multiply(a3, d3)` `# a3 *= d3.`

→ `a3 /= keep_prob` ←

50 units. ⇝ 10 units shut off

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

$a^{[3]}$ reduced by 20%.

$/= 0.8$

Test

Making predictions at test time

$a^{[0]} = X$

No drop out.

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

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

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

$a^{[3]} = \ldots$

$\hat{y}$

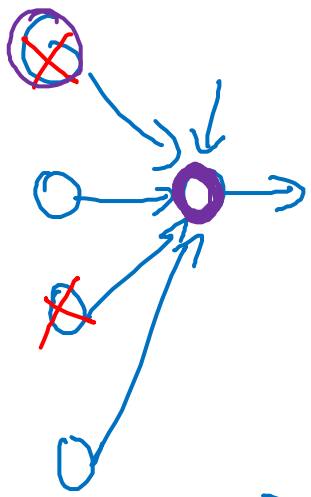
/= keep-prob

Regularizing your neural network

Understanding dropout

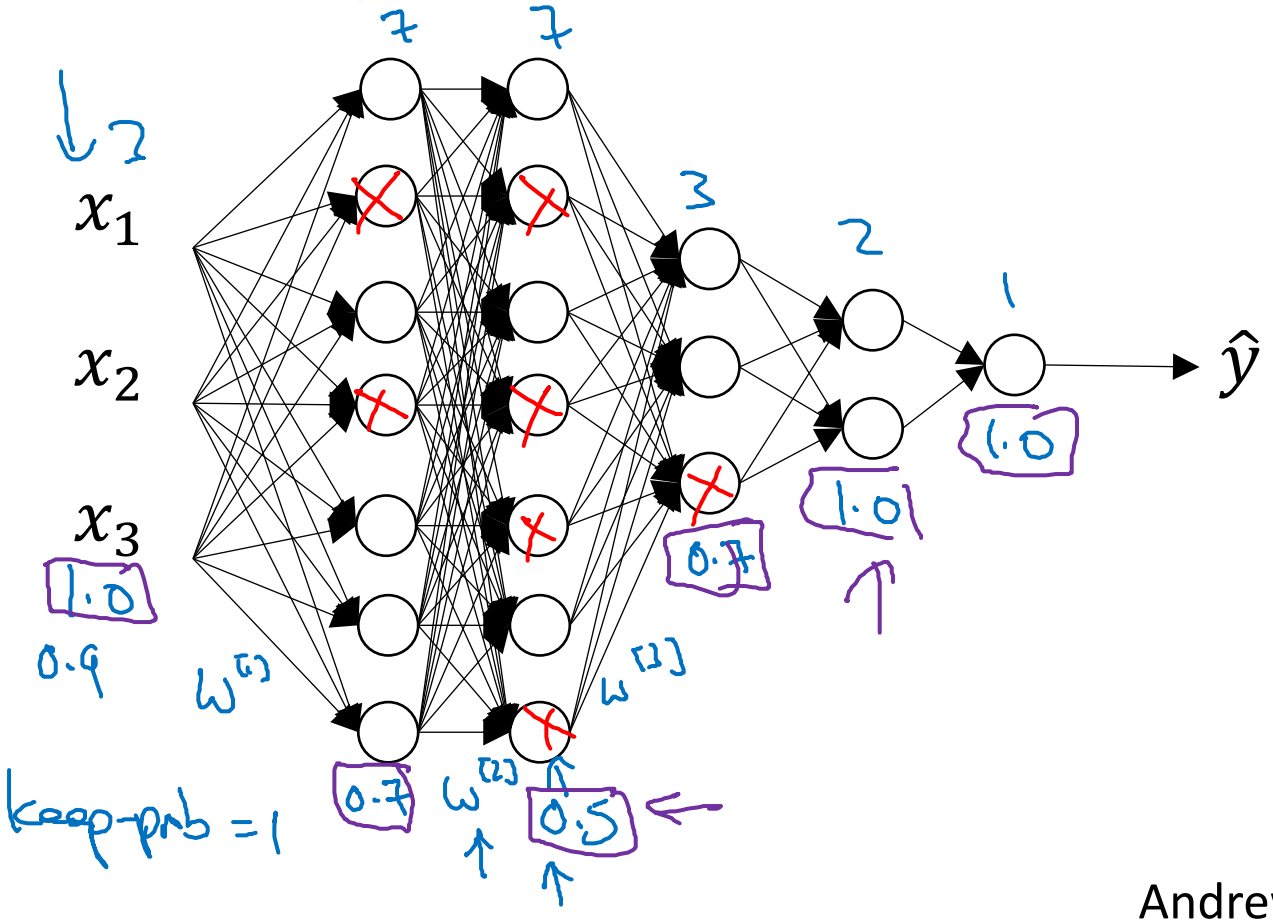
Why does drop-out work?

Intuition: Can't rely on any one feature, so have to spread out weights. ⇝ Shrink weights. b_2



→ Computer Vision

Downside: J



- similar to what we saw with L2 regularization,
- the effect of implementing drop out is that it shrinks
- the weights and does some of those outer regularization that helps prevent over-fitting.
- But it turns out that drop out can formally be
- shown to be an adaptive form without a regularization.
- But L2 penalty on different weights are different,
- depending on the size of the activations being multiplied that way.
- But to summarize, it is possible to show that drop out has
- a similar effect to L2 regularization.
- Only to L2 regularization applied to different ways can be
- a little bit different and even more adaptive to the scale of different inputs.

随机失活（dropout）这种
从网络中随机敲除神经元的做法看起来有些疯狂 但是为什么用于正则化时它的表现这么好呢?
让我们再深究一下 在之前的视频中 我们给出了一个数学上关于梯度 我之前解释dropout会
让神经元随机失活 这就使得好像每一次迭代
都会在一个更小的神经网络中计算 而使用更小的神经网络就好像具有正则化效果 这里再给出第
二个解释 我们从单一神经元的角度来看这个问题 比如这个点 它的任务就是 利用这些输入单元
生成一个有意义的输出 而如果使用了dropout 这些输入会被随机的丢弃 有的时候这两个神经
元会被丢弃 有的时候另一个神经元会被丢弃 因此 这就意味着 我用紫色圈起来的这个 它不能
依赖于任何一个特征 因为每个都可能被随机丢弃 或者说它的每一个输入都可能随机失活 所以
在特定的时候 就不愿把所有的赌注 只放在这一个输入神经元上 对吗? 因为任何一个输入都可
能失活 所以我们也不愿把太多的权重放在某一个上 因此这个神经元将会更积极的使用这种方式
对于每个输入都给一个比较小的权重 而泛化这些权值 将有利于压缩这些权重的平方泛数（平方
和）和L2正则化类似 使用dropout有助于 收缩权值以及防止过拟合 但是 更准确的来说
dropout应该被看作一种自适应形式而不是正则化 L2正则对不同权值的惩罚方式有所不同 这
取决于被激活的乘方大小 总之来说 dropout能起到和L2正则类似的效果 只是针对不同的情况
L2正则可以有少许的变化 所以适用面更广 当你使用dropout时还要注意一个细节 这里是一个
神经网络 有3个输入 这里有7个隐藏神经元 7个 3个 2个 1个 我们必须要确定的一个参数是
留存率（keep prop）它表示一层中一个神经元不失活的概率 因此 可以对每一层设定不同的
留存率 第一层中 W1权值矩阵是3×7 第二层W2是7×7 第三层W3是7×3 以此类推 很明显W2是
最大的权值矩阵 因为它的参数最多 达到了7×7 所以为了让这里不容易发生过拟合 可能对于这
一层 我猜是第二层 你可以设定一个相对低的留存率 0.5 而对于其他你不太担心会发生过拟合
的层 你可以设定一个更高的留存率 比如0.7 如果某一层我们完全不担心会过拟合 你可以把留
存率设定为1.0 为清楚起见 我把这些数字用紫色框起来 可以看到不同层有不同的留存率 需要
注意的是留存率1.0表示 你保留了每一个神经元 在这一层你并没有使用dropout 但是对于那
些容易发生过拟合的层 也就是那些有许多参数的层 为了达到更好的效果 你可以设定一个较小
的留存率 这就好像 你想使用L2正则对某些层进行更严格的正则化时 对参数λ进行起始设定 从
技术上来说 你也可以在输入层上使用dropout 随机的选择一个或几个输入特征进行组合 但是
在实践中 通常不会这样做 最常见的做法是将这一层的留存率设为1.0 当然你也可以设置一个
较高的值 比如0.9 但一般不会让一半的特征输入失活 所以对于输入层 如果你要使用dropout
通常是把它设置为一个接近1的数 总结一下 如果你觉得某一层比其他层更容易发生过拟合 就可
以给这一层设置更低的留存率 这样的缺点是在交叉验证（网格）搜索时 会有更多的超参数
（运行会更费时）另一个选择就是对一些层使用dropout（留存率相同）而另一些不使用 这样
的话 就只有一个超参数了 在最后总结之前 再说几个实际使用时值得注意的 最早对dropout
技术的成功应用 许多是在计算机视觉领域 在这个领域中 它的输入层向量维度非常大 因为要包
含每个像素点的值 几乎不可能有足够的数据 因此dropout在计算机视觉领域使用非常频繁 有
些研究人员总是使用它 几乎已经成为一种默认了 但需要记住dropout是一种正则化技术 目的
是防止过拟合 所以 除非我的算法已经过拟合了 我是不会考虑使用dropout的 所以相对计算
机视觉领域 它在其他应用领域使用会少一些 因为一般没有足够的数据 几乎总是发生过拟合 这
才导致一些计算机视觉专家特别依赖dropout 但这并不表示其他领域也如此 dropout的另一
个缺点是让代价函数J 变得不那么明确 因为每一次迭代 都有一些神经元随机失活 所以当你去
检验梯度下降算法表现的时候 你会发现很难确定代价函数是否已经定义的足够好（随着迭代 值

不断变小） 这是因为你对代价函数J 的定义不明确 或者难以计算 因此就不能用绘图的方法去调试错误了 像这样的图 通常这个时候我会关闭dropout 把留存率设为1 然后再运行代码并确保代价函数J 是单调递减的 最后再打开dropout并期待 使用dropout的时候没有引入别的错误 我想 你需要使用其他方法 而不是类似这种画图的方法去确保你的代码 在使用dropout后梯度下降算法依然有效 到此为止 仍然有一些值得我们去了解的正则化技术 我们将在下一段视频中介绍