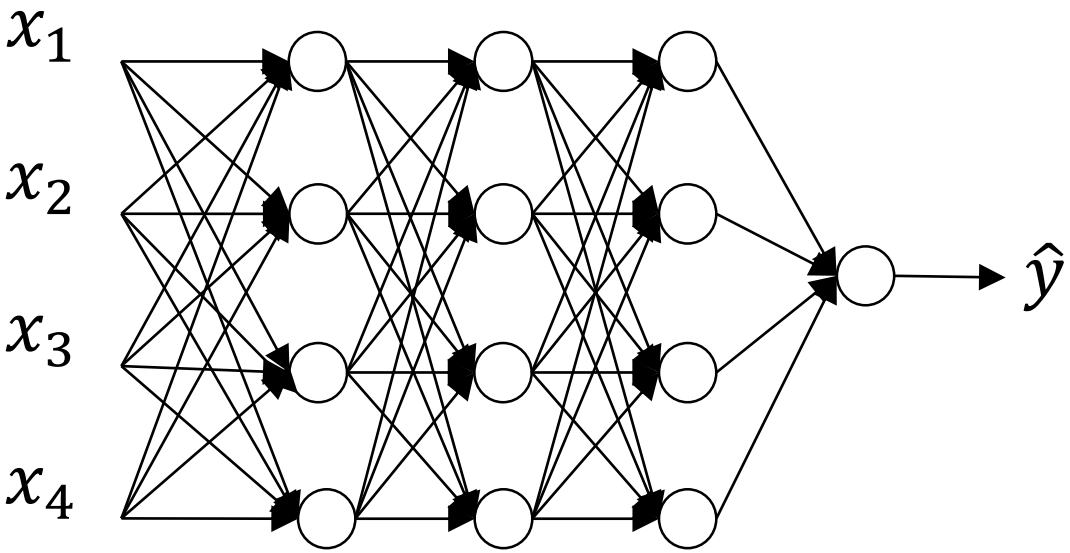
deeplearning.ai

Regularizing your neural network

Dropout regularization

Dropout regularization



0.5 0.5 0.5

So you end up with a much smaller, really diminished network.

Implementing dropout ("Inverted dropout")

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

There will be the probability that a given hidden unit will be kept.

0.2

There's a 0.2 chance of eliminating any hidden unit

$d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\text{-}prob$

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

For every element zero in d3, you end up zeros out the corresponding element of d3.

The invert dropout technique by dividing by the keep.prop, it ensures that the expected value of a3 remains the same.

$a3 /= keep\text{-}prob$

This makes test time easier, because you have less of a scaling problem.

50 units. ⇝ 10 units shut off

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

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

Test

$/= 0.8$

This will correct or just a bump that back up by the roughly 20% that you need. So it's not changed the expected value of a3.

Andrew Ng

Making predictions at test time

$a^{[0]} = X$

No drop out.

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

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

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

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

$\hat{y}$

The effect of that was to ensure that even you don't implement dropout at test time to the scaling, the expect value of these activations don't change. So you don't need to add in an extra funny scaling parameter at test time.

$/= keep\text{-}prob$

And that's because when you are making predictions at the test time, you don't really want your output to be random. If you are implementing dropout at test time, that just add noise to your predictions. In theory, one thing you could do is run a prediction process many times with different hidden units randomly dropped out and have it across them. But that' computationally in efficient and will give you roughly the same result, very, very similar results to this different procedure as well.

Andrew Ng

deeplearning.ai

Regularizing your neural network

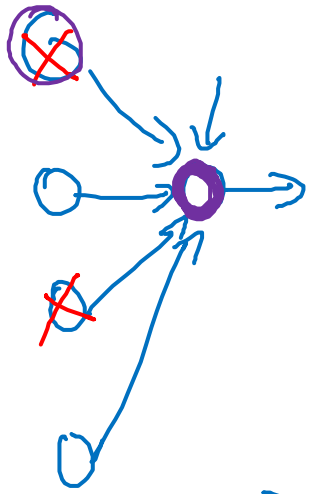
Understanding dropout

Dropout randomly knocks out units in your network. So It's as if on every iteration, you're working with a smaller neural network, and so using a smaller network seems like it should have a regularizing effect.

Why does drop-out work?

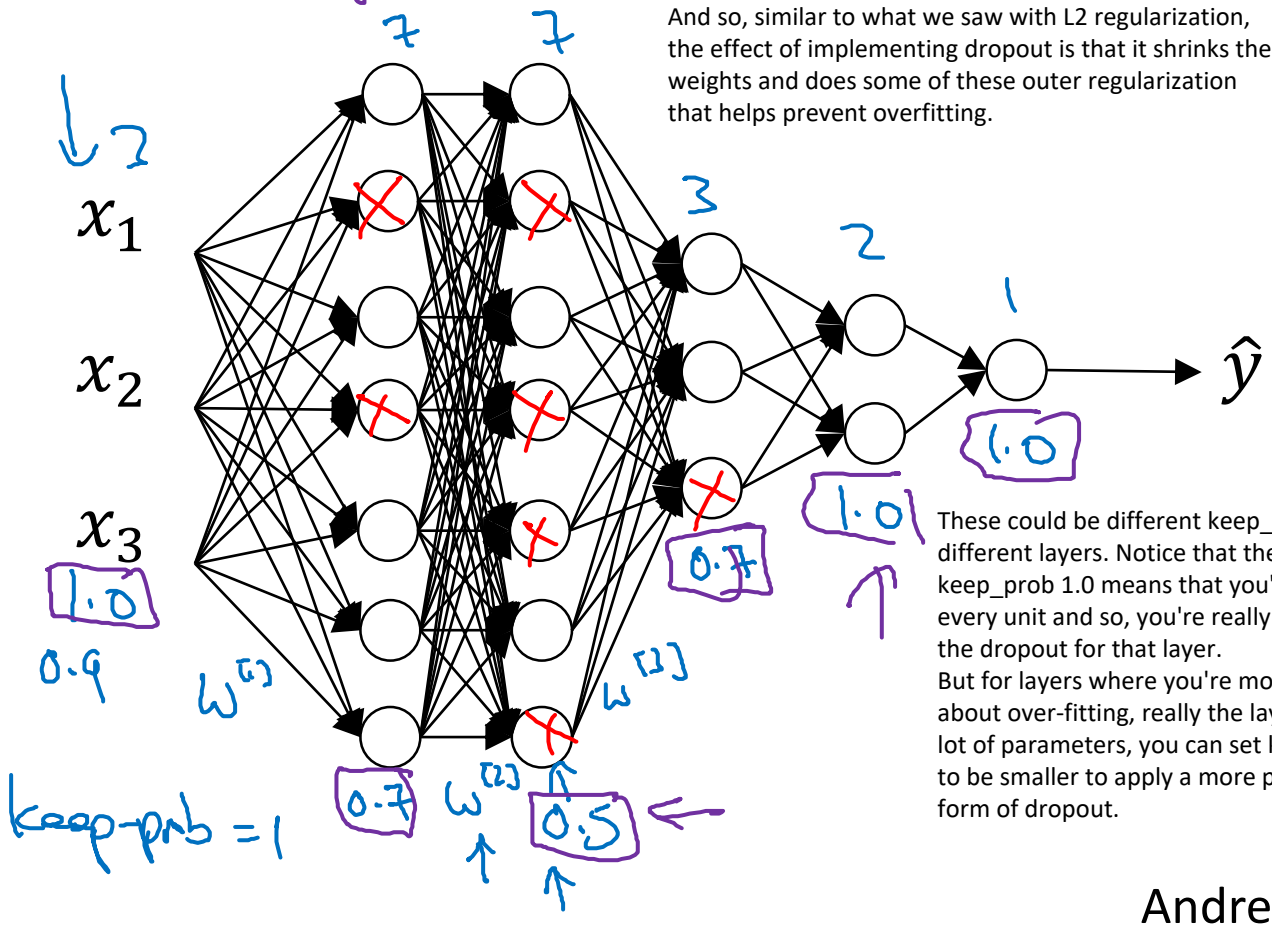
But it turns out that dropout can formally be shown to be in an adaptive form without a regularization. But L2 penality on different weights are different, depending on the size of the activations being multiplied that way. But to summarize, it is possible to show that dropout has a similar effect to 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.

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



→ Computer Vision

Downside: J



This will tend to have an effect of shrinking the squared norm of the weights.
And so, similar to what we saw with L2 regularization, the effect of implementing dropout is that it shrinks the weights and does some of these outer regularization that helps prevent overfitting.

These could be different keep_probs for different layers. Notice that the keep_prob 1.0 means that you're keeping every unit and so, you're really not using the dropout for that layer.
But for layers where you're more worried about over-fitting, really the layers with a lot of parameters, you can set keep_prob to be smaller to apply a more powerful form of dropout.

Andrew Ng

It's kind of like cranking up the regularization parameter lambda for L2 regularization where you try to regularize some layers more than others. And technically, you can also apply dropout to the input layer,where you can have some chance of just maxing out one or more of the input features. Although in practice, usually don't do that often. And so keep_prob of one point zero is quite common for the input layer. You can also use a very high value, maybe zero point nine, but it's much less likely that you want to eliminate half of the input features. So usually keep_prob, if you apply the law, would be a number close to one, if you even apply dropout at all to the input layer.

The downsides is ,this gives you even more hyper parameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and the just have one hyper parameter, which is the keep_prob for the layers for which you do apply dropout.

But really the thing to remember is that dropout is a regularization technique, it helps prevent over-fitting. And so unless my algorithm is over-fitting, I wouldn't actually bother to use dropout. SO it's used somewhat less often than other application areas. There is just with computer vision, you usually just don't have enough data, so you're almost always overfitting, which is why there tends to be some computer vision researchers who swear by dropout. But by intuition, I doesn't always generalize I think to other disciplines.

One big downside of dropout is that the cost function J is no longer well-defined. On every iteration, you are randomly killing off a bunch of nodes, and so, if you are double checking the performance of gradient descent, it's actually harder to double check that. Because the cost function J that you're optimizing is actually less well-defind, or is certainly hard to calculate. So you lose this debugging tool to will a plot, a graph like this. So what I usually do is turn off dropout, you will set keep_prob equals one and run my code and make sure that it is monotonically decreasing J, and then turn on dropout and hope that I didn't introduce bugs into my code during dropout.