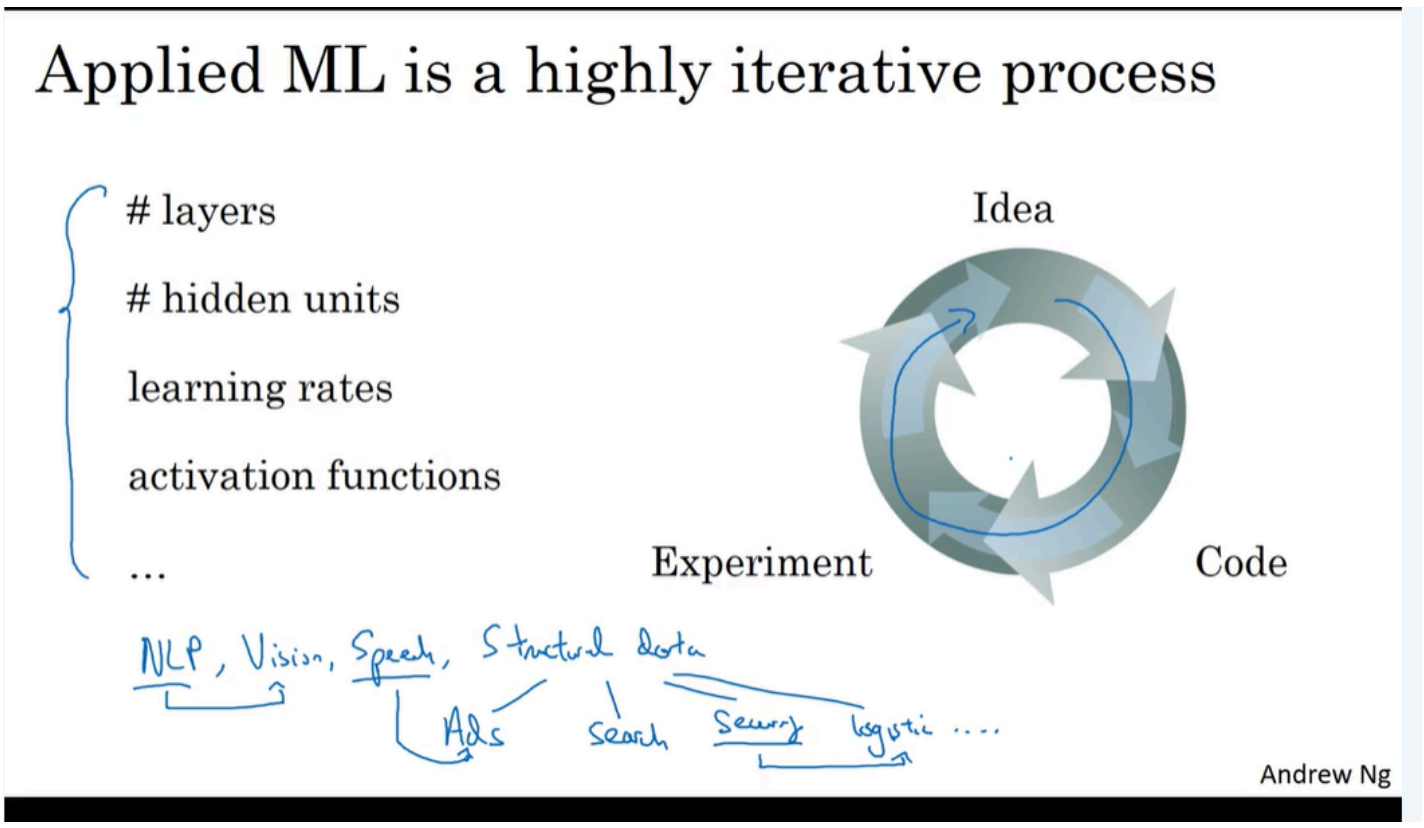
Regularizing your Neural Network

Train / Dev / Test sets



Perhaps now you've learned how to implement a neural network. In this week, you'll learn the practical aspects of how to make your neural network work well. Ranging from things like hyperparameter tuning to how to set up your data, to how to make sure your optimization algorithm runs quickly so that you get your learning algorithm to learn in a reasonable amount of time. In this first week, we'll first talk about how the cellular machine learning problem, then we'll talk about randomization, then we'll talk about some tricks for making sure your neural network implementation is correct. With that, let's get started. Making good choices in how you set up your training, development, and test sets can make a huge difference in helping you quickly find a good high-performance neural network.

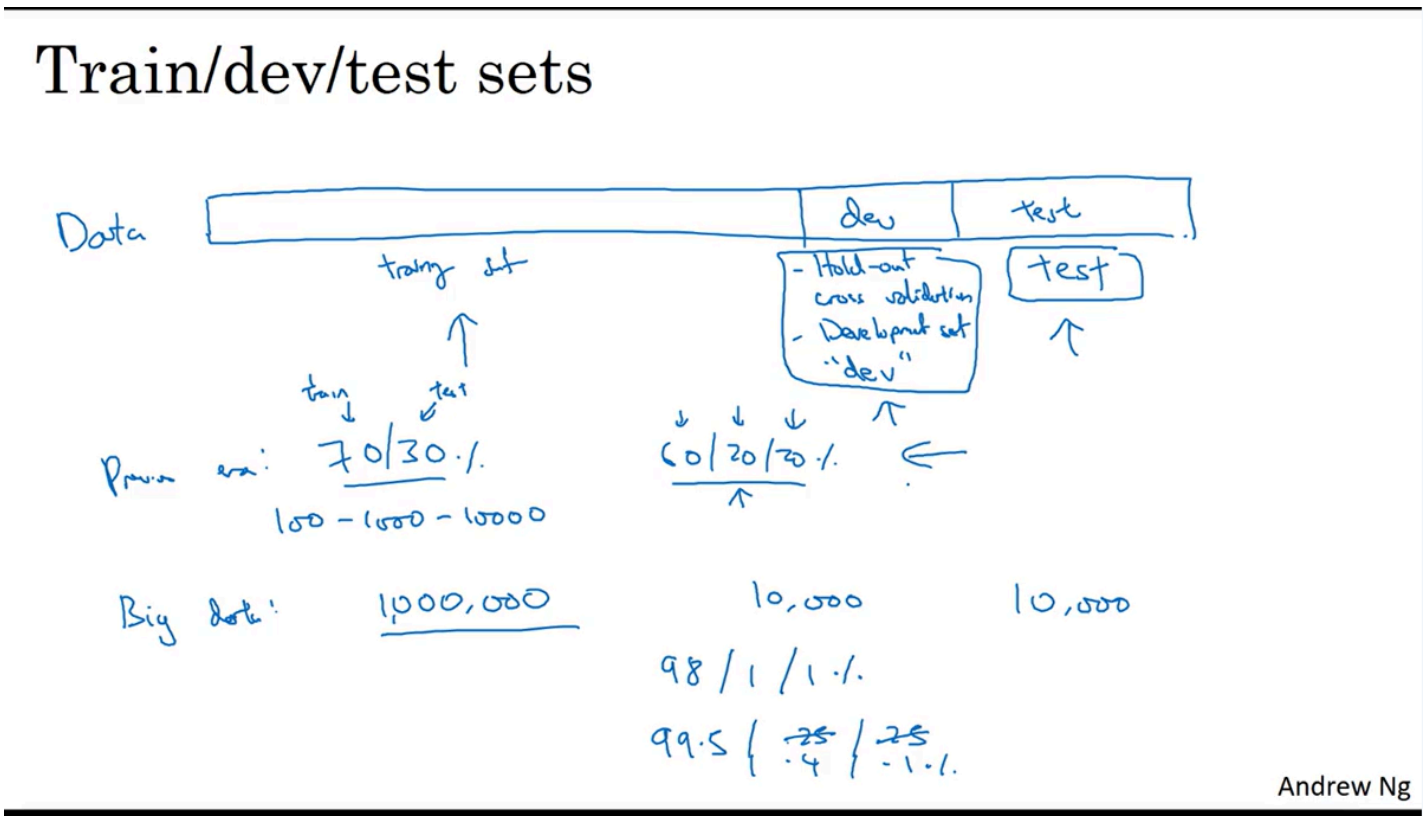
When training a neural network, you have to make a lot of decisions, such as how many layers will your neural network have? And, how many hidden units do you want each layer to have? And, what's the learning rate? And, what are the activation functions you want to use for the different layers? When you're starting on a new application, it's almost impossible to correctly guess the right values for all of these, and for other hyperparameter choices, on your first attempt. So, in practice, applied machine learning is a highly iterative process, in which you often start with an idea, such as you want to build a neural network of a certain number of layers, a certain number of hidden units, maybe on certain data sets, and so on. And then you just have to code it up and try it, by running your code.

You run an experiment and you get back a result that tells you how well this particular network, or this particular configuration works. And based on the outcome, you might then refine your ideas and change your choices and maybe keep iterating, in order to try to find a better and a better, neural network. Today, deep learning has found great success in a lot of areas ranging from natural language processing, to computer vision, to speech recognition, to a lot of applications on also structured data. And structured data includes everything from advertisements to web search, which isn't just Internet search engines. It's also, for example, shopping websites. Already any website that wants to deliver great search results when you enter terms into a search bar. To computer security, to logistics, such as figuring out where to send drivers to pick up and drop off things...to many more.

So what I'm seeing is that sometimes a researcher with a lot of experience in NLP might enter...you know, might try to do something in computer vision. Or maybe a researcher with a lot of experience in speech recognition might, you know, jump in and try to do something on advertising. Or someone from security might want to jump in and do something on logistics. And what I've seen is that intuitions from one domain or from one application area often do not transfer to other application areas. And the best choices may depend on the amount of data you have, the number of input features you have through your computer configuration and whether you're training on GPUs or CPUs. And if so, exactly what configuration of GPUs and CPUs...and many other things. So, for a lot of applications, I think it's almost impossible.

Even very experienced deep learning people find it almost impossible to correctly guess the best choice of hyperparameters the very first time. And so today, applied deep learning is a very iterative process where you just have to go around this cycle many times to hopefully find a good choice of network for your application. So one of the things that

determine how quickly you can make progress is how efficiently you can go around this cycle. And setting up your data sets well, in terms of your train, development and test sets can make you much more efficient at that.

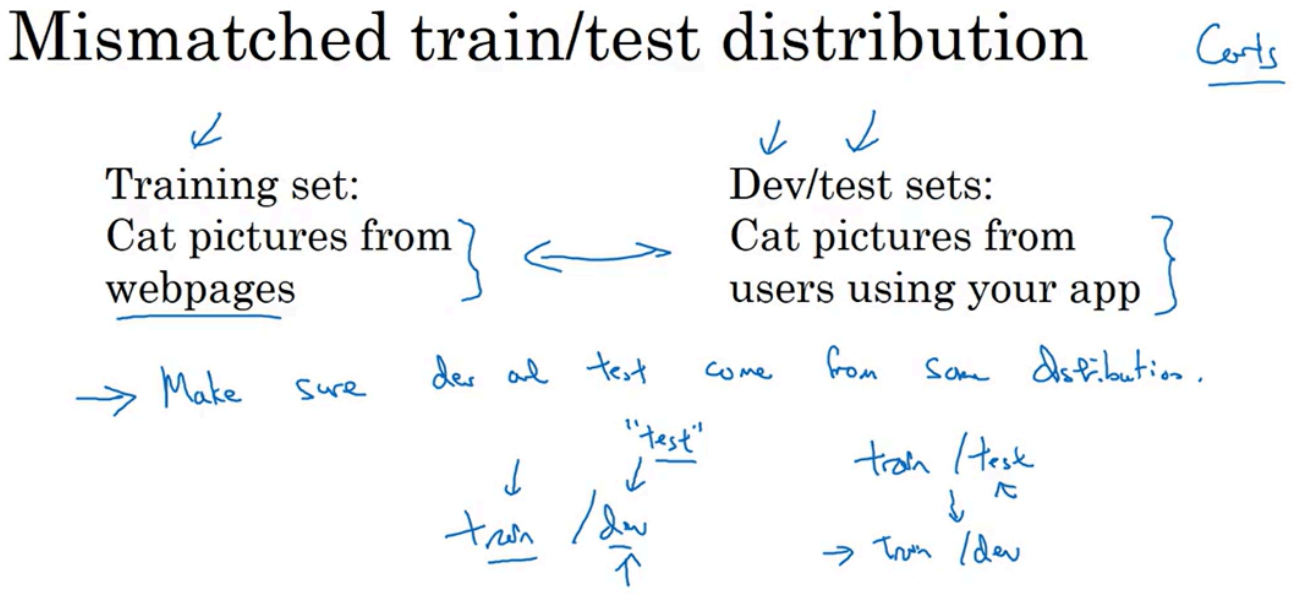


So if this is your training data, let's draw that as a big box. Then traditionally, you might take all the data you have and carve off some portion of it to be your training set, some portion of it to be your hold-out cross validation set, and this is sometimes also called the development set. And for brevity, I'm just going to call this the dev set, but all of these terms mean roughly the same thing.

And then you might carve out some final portion of it to be your test set. And so the workflow is that you keep on training algorithms on your training set. And use your dev set or your hold-out cross validation set to see which of many different models performs best on your dev set. And then after having done this long enough, when you have a final model that you want to evaluate, you can take the best model you have found and evaluate it on your test set in order to get an unbiased estimate of how well your algorithm is doing. So in the previous era of machine learning, it was common practice to take all your data and split it according to maybe a 70/30% in terms of a...people often talk about the 70/30 train test splits. If you don't have an explicit dev set or maybe a 60/20/20% split, in terms of 60% train, 20% dev and 20% test. And several years ago, this was widely considered best practice in machine learning.

If you have here maybe 100 examples in total, maybe 1000 examples in total, maybe after 10,000 examples, these sorts of ratios were perfectly reasonable rules of thumb. But in the modern big data era, where, for example, you might have a million examples in total, then the trend is that your dev and test sets have been becoming a much smaller percentage of the total. Because remember, the goal of the dev set or the development set is that you're going to test different algorithms on it and see which algorithm works better. So the dev set just needs to be big enough for you to evaluate, say, two different algorithm choices or ten different algorithm choices and quickly decide which one is doing better. And you might not need a whole 20% of your data for that. So, for example, if you have a million training examples, you might decide that just having 10,000 examples in your dev set is more than enough to evaluate, you know, which one or two algorithms does better. And in a similar vein, the main goal of your test set is, given your final classifier, to give you a pretty confident estimate of how well it's doing.

And again, if you have a million examples, maybe you might decide that 10,000 examples is more than enough in order to evaluate a single classifier and give you a good estimate of how well it's doing. So, in this example, where you have a million examples, if you need just 10,000 for your dev and 10,000 for your test, your ratio will be more like...this 10,000 is 1% of 1 million, so you'll have 98% train, 1% dev, 1% test. And I've also seen applications where, if you have even more than a million examples, you might end up with, you know, 99.5% train and 0.25% dev, 0.25% test. Or maybe a 0.4% dev, 0.1% test. So just to recap, when setting up your machine learning problem, I'll often set it up into a train, dev and test sets, and if you have a relatively small dataset, these traditional ratios might be okay. But if you have a much larger data set, it's also fine to set your dev and test sets to be much smaller than your 20% or even 10% of your data. We'll give more specific guidelines on the sizes of dev and test sets later in this specialization.



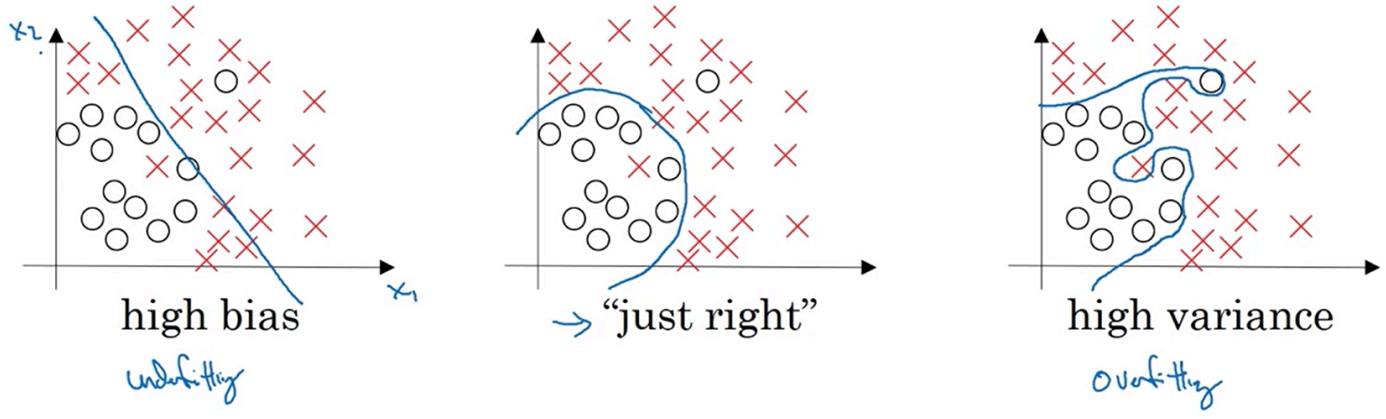
One other trend we're seeing in the era of modern deep learning is that more and more people train on mismatched train and test distributions. Let's say you're building an app that lets users upload a lot of pictures and your goal is to find pictures of cats in order to show your users. Maybe all your users are cat lovers. Maybe your training set comes from cat pictures downloaded off the Internet, but your dev and test sets might comprise cat pictures from users using your app. So maybe your training set has a lot of pictures crawled off the Internet but the dev and test sets are pictures uploaded by users. Turns out a lot of webpages have very high resolution, very professional, very nicely framed pictures of cats. But maybe your users are uploading, you know, blurrier, lower res images just taken with a cell phone camera in a more casual condition.

Finally, it might be okay to not have a test set. Remember, the goal of the test set is to give you a ... unbiased estimate of the performance of your final network, of the network that you selected. But if you don't need that unbiased estimate, then it might be okay to not have a test set. So what you do, if you have only a dev set but not a test set, is you train on the training set and then you try different model architectures. Evaluate them on the dev set, and then use that to iterate and try to get to a good model. Because you've fit your data to the dev set, this no longer gives you an unbiased estimate of performance. But if you don't need one, that might be perfectly fine.

In the machine learning world, when you have just a train and a dev set but no separate test set, most people will call this a training set and they will call the dev set the test set. But what they actually end up doing is using the test set as a hold-out cross validation set. Which maybe isn't completely a great use of terminology, because they're then overfitting to the test set. So when the team tells you that they have only a train and a test set, I would just be cautious and think, do they really have a train dev set? Because they're overfitting to the test set. Culturally, it might be difficult to change some of these team's terminology and get them to call it a trained dev set rather than a trained test set, even though I think calling it a train and development set would be more correct terminology. And this is actually okay practice if you don't need a completely unbiased estimate of the performance of your algorithm.

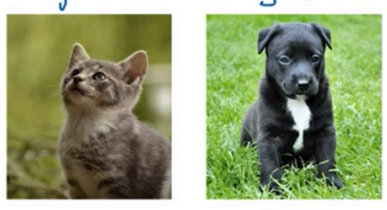
Bias / Variance

Bias and Variance



Bias and Variance

Cat classification



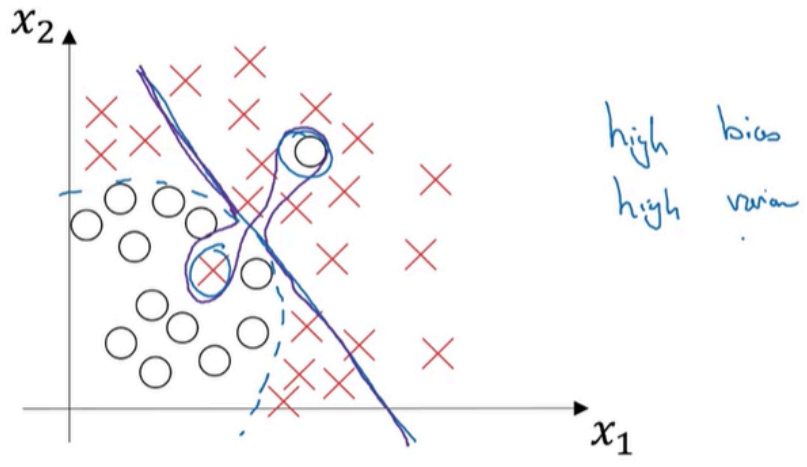
Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

Human: ≈0%

Optimal (Bayes) error: ~~0%~~ 15%

Blurry images

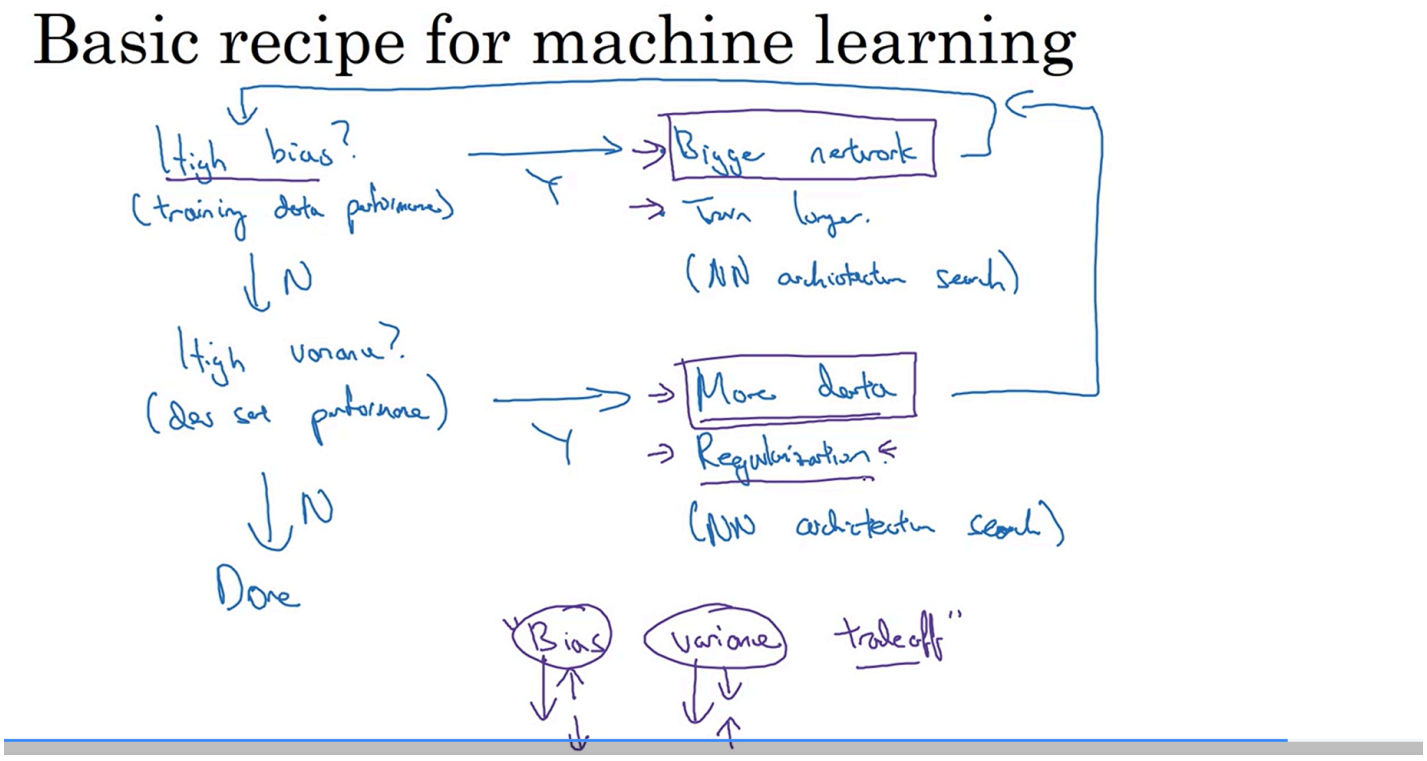
High bias and high variance



the purple line is high bias and high variance

Basic Recipe for Machine Learning

After having trained in an initial model, I will first ask, does your algorithm have high bias? And so, to try and evaluate if there is high bias, you should look at, really, the training set or the training data performance. Right. And so, if it does have high bias, does not even fitting in the training set that well, some things you could try would be to try pick a network, such as more hidden layers or more hidden units, or you could train it longer, you know, maybe run trains longer or try some more advanced optimization algorithms, which we'll talk about later in this course.



Regularization

Please note that in the next video at 5:45, the Frobenius norm formula should be the following:

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(w_{i,j}^{[l]}\right)^2$$

The limit of summation of i should be from 1 to $n^{[l]}$

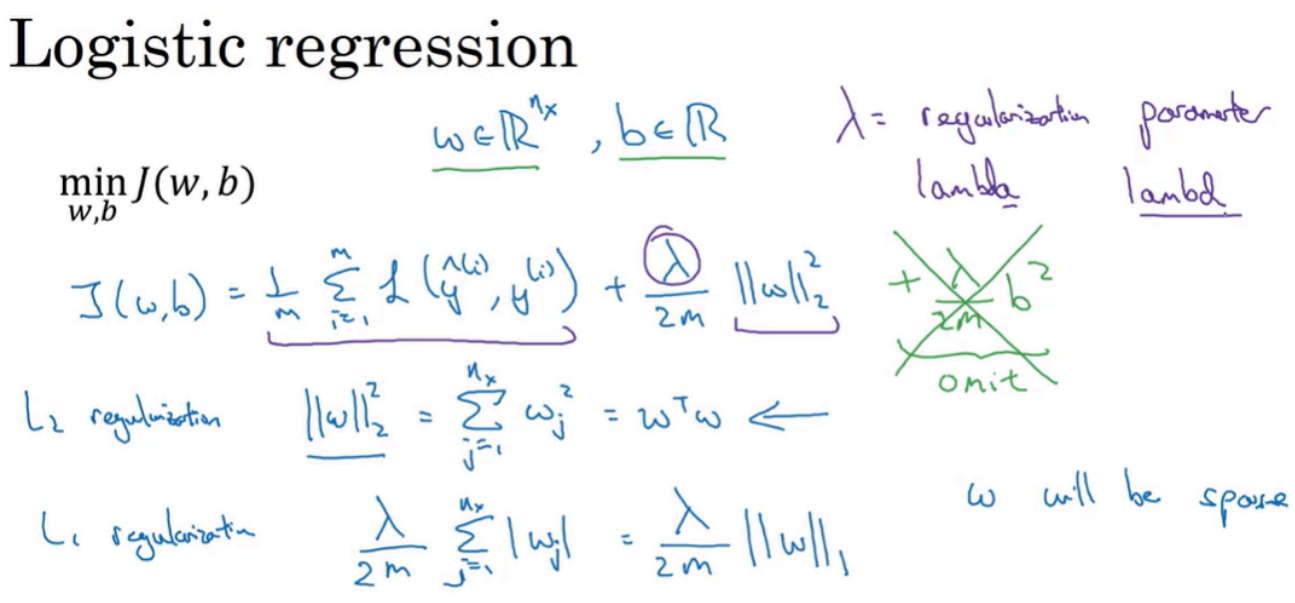
The limit of summation of j should be from 1 to $n^{[l-1]}$.

(it's flipped in the video). The rows "i" of the matrix should be the number of neurons in the current layer $n^{[l]}$,

whereas the columns "j" of the weight matrix should equal the number of neurons in the previous layer $n^{[l-1]}$.

If you suspect your neural network is over fitting your data, that is, you have a high variance problem, one of the first things you should try is probably regularization. The other way to address high variance is to get more training data that's also quite reliable. But you can't always get more training data, or it could be expensive to get more data. But adding regularization will often help to prevent overfitting, or to reduce variance in your network

Recall that for logistic regression, you try to minimize the cost function J, which is defined as this cost function.



Some of your training examples of the losses of the individual predictions in the different examples, where you recall that w and b in the logistic regression, are the parameters. So w is an x-dimensional parameter vector, and b is a real number. And so, to add regularization to logistic regression, what you do is add to it, this thing, lambda, which is called the regularization parameter. I'll say more about that in a second. But lambda / 2m times the norm of w squared. So here, the norm of w squared, is just equal to sum from j equals 1 to nx of wj squared, or this can also be written w, transpose w, it's just a square Euclidean norm of the prime to vector w. And this is called L2 regularization.

Because here, you're using the Euclidean norm, also it's called the L2 norm with the parameter vector w. Now, why do you regularize just the parameter w? Why don't we add something here, you know, about b as well? In practice, you could do this, but I usually just omit this. Because if you look at your parameters, w is usually a pretty high dimensional parameter vector, especially with a high variance problem. Maybe w just has a lot of parameters, so you aren't fitting all the parameters well, whereas b is just a single number. So almost all the parameters are in w rather than b.

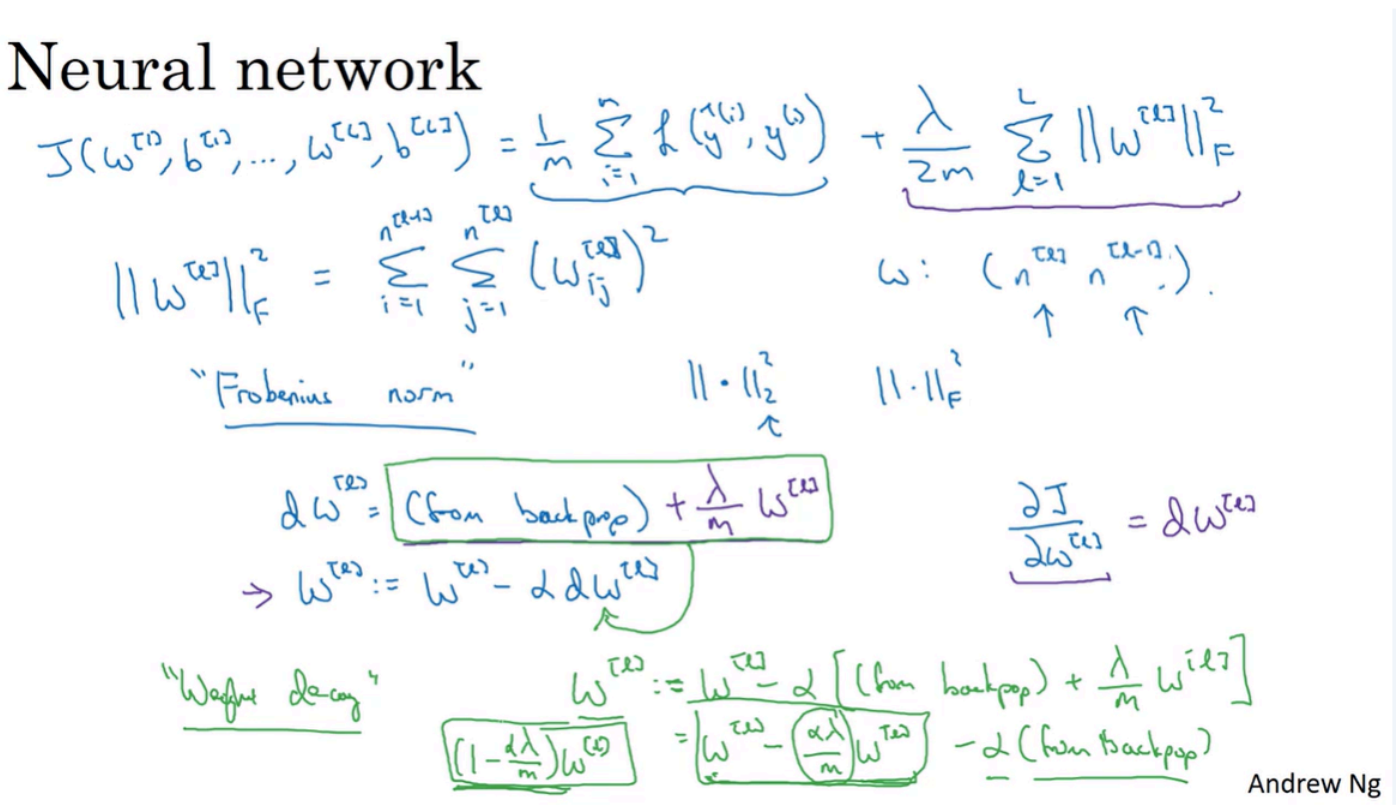
And if you add this last term, in practice, it won't make much of a difference, because b is just one parameter over a very large number of parameters. In practice, I usually just don't bother to include it. But you can if you want. So L2 regularization is the most common type of regularization. You might have also heard of some people talk about L1 regularization. And that's when you add, instead of this L2 norm, you instead add a term that is lambda over m of sum over, of this. And this is also called the L1 norm of the parameter vector w, so the little subscript 1 down there, right?

And I guess whether you put m or 2m in the denominator, is just a scaling constant. If you use L1 regularization, then w will end up being sparse. And what that means is that the w vector will have a lot of zeros in it. And some people say that this can help with compressing the model, because the set of parameters are zero, then you need less memory to store the model. Although, I find that, in practice, L1 regularization, to make your model sparse, helps only a little bit. So I don't think it's used that much, at least not for the purpose of compressing your model. And when people train your networks, L2 regularization is just used much, much more often.

(Sorry, just fixing up some of the notation here). So, one last detail. Lambda here is called the regularization parameter. And usually, you set this using your development set, or using hold-out cross validation. When you try a variety of values and see what does the best, in terms of trading off between doing well in your training set versus also setting that two normal of your parameters to be small, which helps prevent over fitting. So lambda is another hyper parameter that you might have to tune. And by the way, for the programming exercises, lambda is a reserved keyword in the Python programming language.

So in the programming exercise, we will have l-a-m-b-d, without the a, so as not to clash with the reserved keyword in Python. So we use l-a-m-b-d to represent the lambda regularization parameter. So this is how you implement L2 regularization for logistic regression.

Let’s move to neural network



In a neural network, you have a cost function that's a function of all of your parameters, w[1], b[1] through w[capital L], b[capital L], where capital L is the number of layers in your neural network. And so the cost function is this, sum of the losses, sum over your m training examples. And so to add regularization, you add lambda over 2m, of sum over all of your parameters w, your parameter matrix is w, of their, that's called the squared norm.

Where, this norm of a matrix, really the squared norm, is defined as the sum of i, sum of j, of each of the elements of that matrix, squared. And if you want the indices of this summation, this is sum from i=1 through n[l minus 1]. Sum from j=1 through n[l], because w is a n[l] by n[l minus 1] dimensional matrix, where these are the number of hidden units or number of units in layers [l minus 1] in layer l. So this matrix norm, it turns out is called the Frobenius norm of the matrix, denoted with a F in the subscript. So for arcane linear algebra technical reasons, this is not called the, you know, l2 norm of a matrix. Instead, it's called the Frobenius norm of a matrix. I know it sounds like it would be more natural to just call the l2 norm of the matrix, but for really arcane reasons that you don't need to know, by convention, this is called the Frobenius norm.

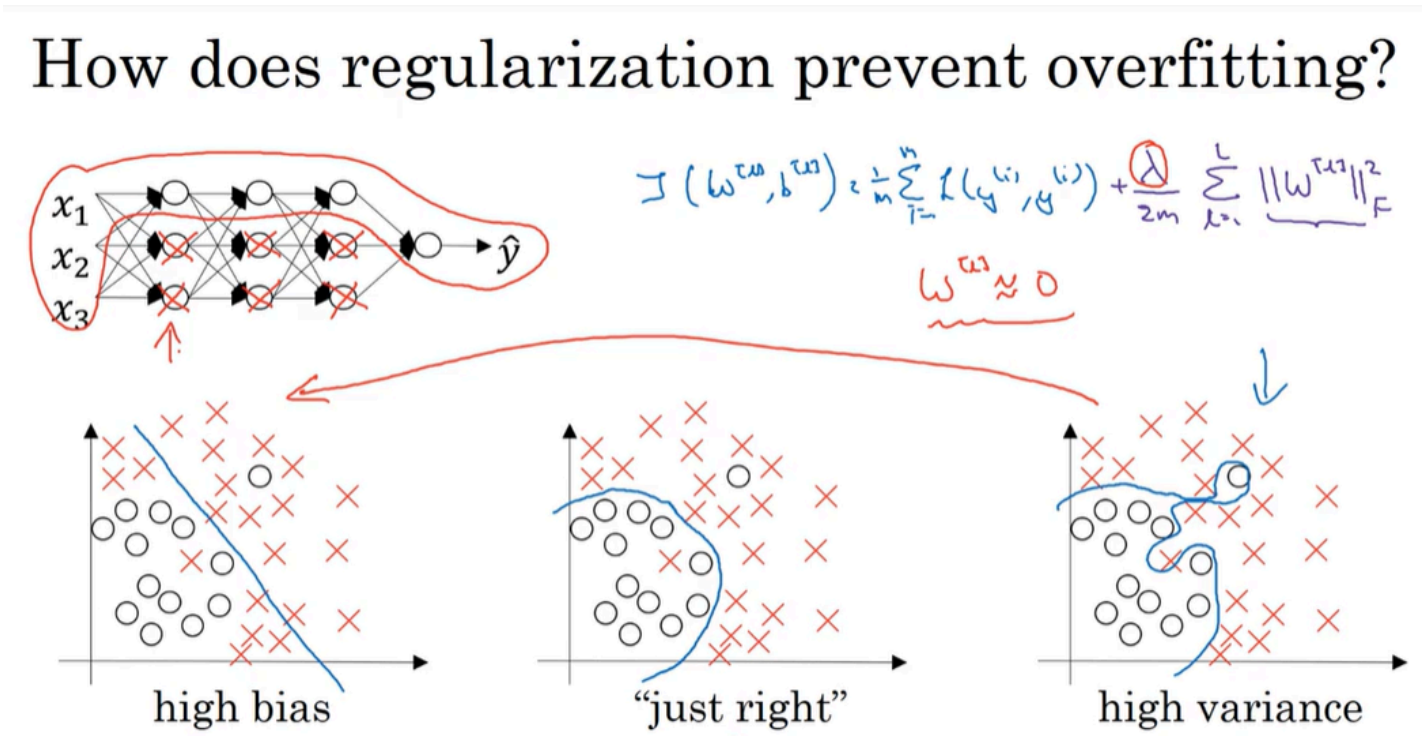
It just means the sum of square of elements of a matrix. So how do you implement gradient descent with this? Previously, we would complete dw, you know, using backprop, where backprop would give us the partial derivative of J with respect to w, or really w for any given [l]. And then you update w[l], as w[l] minus the learning rate, times d. So this is before we added this extra regularization term to the objective. Now that we've added this regularization term to the objective, what you do is you take dw and you add to it, lambda over m times w. And then you just compute this update, same as before.

And it turns out that with this new definition of dw[l], this is still, you know, this new dw[l] is still a correct definition of the derivative of your cost function, with respect to your parameters, now that you've added the extra regularization term at the end. And it's for this reason that L2 regularization is sometimes also called weight decay. So if I take this definition of dw[l] and just plug it in here, then you see that the update is w[l] gets updated as w[l] times the learning rate alpha times, you know, the thing from backprop, plus lambda over m, times w[l]. Let's move the minus sign there. And so this is equal to w[l] minus alpha, lambda over m times w[l], minus alpha times, you know, the thing you got from backprop. And so this term shows that whatever the matrix w[l] is, you're going to make it a little bit smaller, right? This is actually as if you're taking the matrix w and you're multiplying it by 1 minus alpha lambda over m.

You're really taking the matrix w and subtracting alpha lambda over m times this. Like you're multiplying the matrix w by this number, which is going to be a little bit less than 1. So this is why L2 norm regularization is also called weight decay. Because it's just like the ordinary gradient descent, where you update w by subtracting alpha, times the original gradient you got from backprop. But now you're also, you know, multiplying w by this thing, which is a little bit less than 1. So the alternative name for L2 regularization is weight decay. I'm not really going to use that name, but the intuition for why it's called weight decay is that this first term here, is equal to this.

So you're just multiplying the weight matrix by a number slightly less than 1. So that's how you implement L2 regularization in a neural network

Why Regularization Reduces Overfitting?

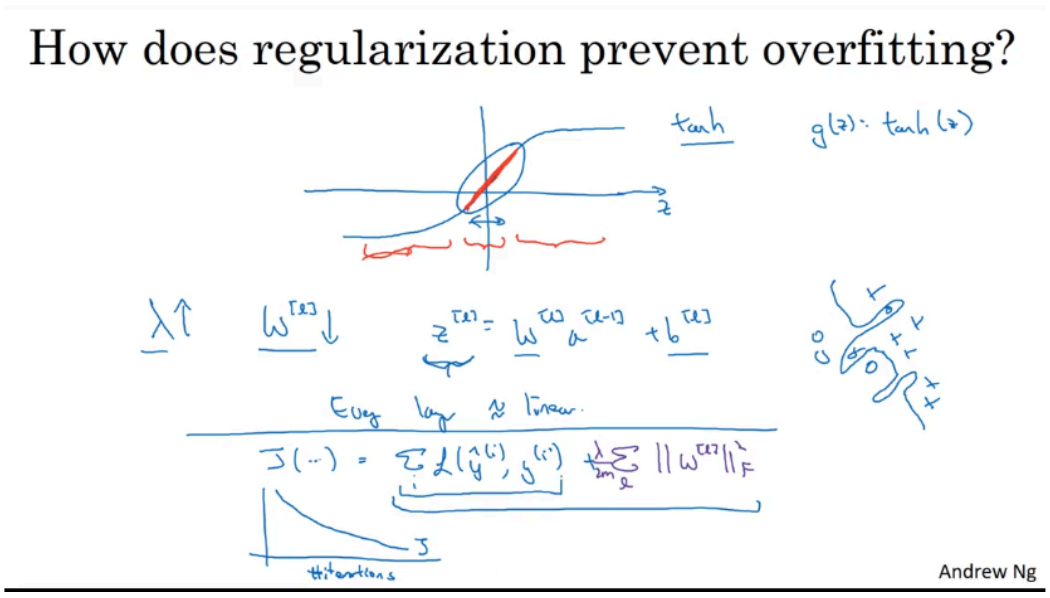


what we did for regularization was add this extra term that penalizes the weight matrices from being too large. And we said that was the Frobenius norm. So why is it that shrinking the L2 norm, or the Frobenius norm with the parameters might cause less overfitting? One piece of intuition is that if you, you know, crank your regularization lambda to be really, really big, that'll be really incentivized to set the weight matrices, W, to be reasonably close to zero. So one piece of intuition is maybe it'll set the weight to be so close to zero for a lot of hidden units that's basically zeroing out a lot of the impact of

these hidden units. And if that's the case, then, you know, this much simplified neural network becomes a much smaller neural network. In fact, it is almost like a logistic regression unit, you know, but stacked multiple layers deep.

And so that will take you from this overfitting case, much closer to the left, to the other high bias case. But, hopefully, there'll be an intermediate value of lambda that results in the result closer to this "just right" case in the middle. But the intuition is that by cranking up lambda to be really big, it'll set W close to zero, which, in practice, this isn't actually what happens. We can think of it as zeroing out, or at least reducing, the impact of a lot of the hidden units, so you end up with what might feel like a simpler network, that gets closer and closer as if you're just using logistic regression. The intuition of completely zeroing out a bunch of hidden units isn't quite right. It turns out that what actually happens is it'll still use all the hidden units, but each of them would just have a much smaller effect. But you do end up with a simpler network, and as if you have a smaller network that is, therefore, less prone to overfitting.

So I'm not sure if this intuition helps, but when you implement regularization in the program exercise, you actually see some of these variance reduction results yourself.



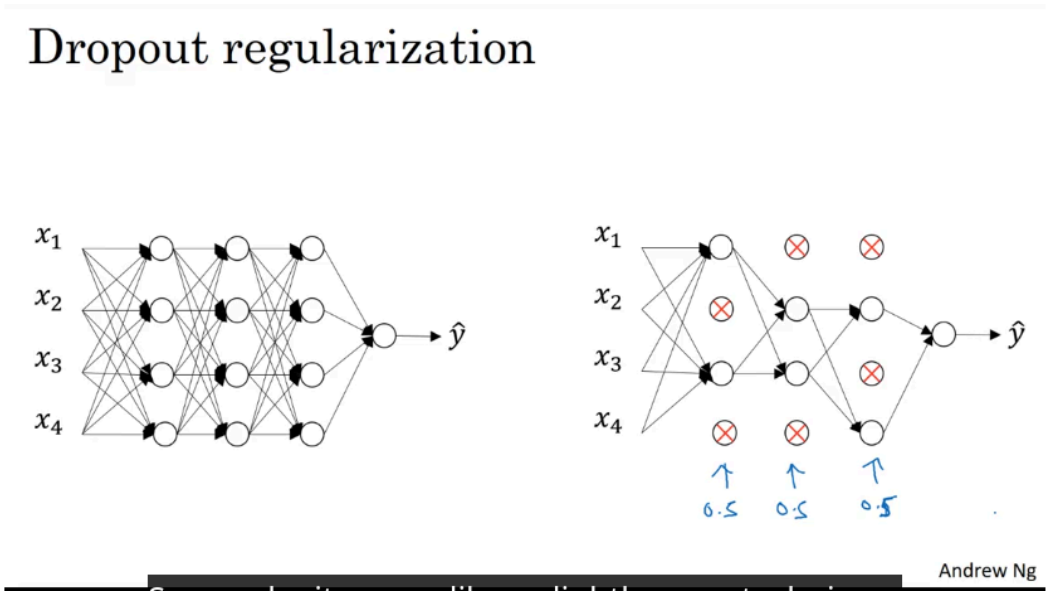
if the regularization parameters are very large, the parameters W very small, so z will be relatively small, kind of ignoring the effects of b for now, but so z is relatively, so z will be relatively small, or really, I should say it takes on a small range of values.

And so the activation function if it's tan h, say, will be relatively linear. And so your whole neural network will be computing something not too far from a big linear function, which is therefore, pretty simple function, rather than a very complex highly non-linear function. And so, is also much less able to overfit, ok? And again, when you implement regularization for yourself in the program exercise, you'll be able to see some of these effects yourself. Before wrapping up our discussion on regularization, I just want to give you one implementational tip, which is that, when implementing regularization, we took our definition of the cost function J and we actually modified it by adding this extra term that penalizes the weights being too large.

And so if you implement gradient descent, one of the steps to debug gradient descent is to plot the cost function J, as a function of the number of elevations of gradient descent, and you want to see that the cost function J decreases monotonically after every elevation of gradient descent. And if you're implementing regularization, then please remember that J now has this new definition.

Dropout Regularization

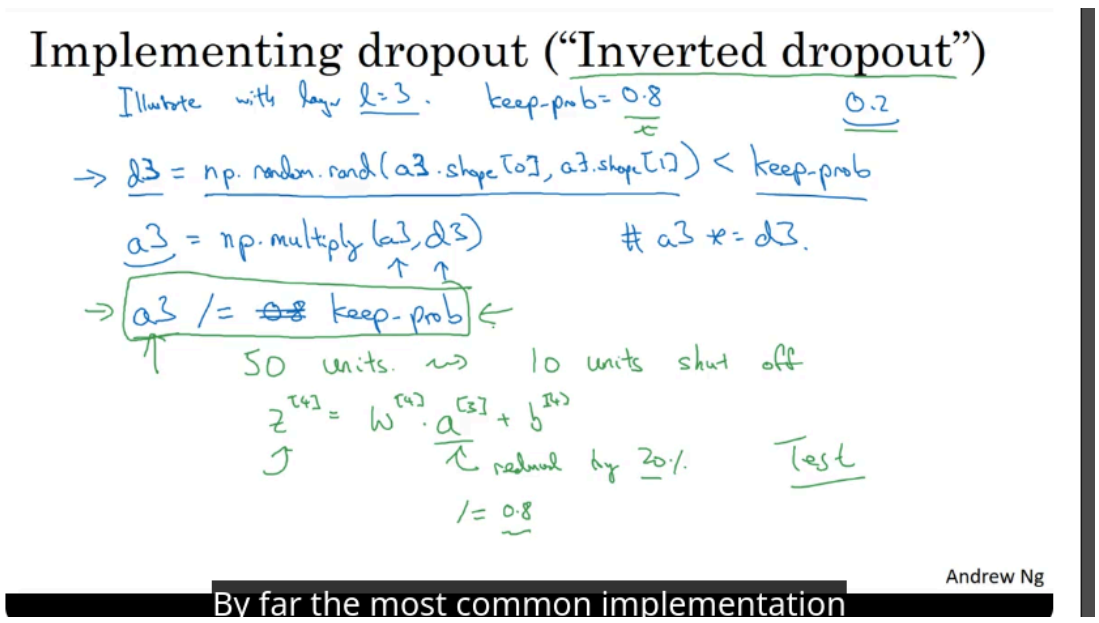
In addition to L2 regularization, another very powerful regularization techniques is called "dropout." Let's see how that works. Let's say you train a neural network like the one on the left and there's over-fitting. Here's what you do with dropout. Let me make a copy of the neural network. With dropout, what we're going to do is go through each of the layers of the network and set some probability of eliminating a node in neural network. Let's say that for each of these layers, we're going to- for each node, toss a coin and have a 0.5 chance of keeping each node and 0.5 chance of removing each node.



So, after the coin tosses, maybe we'll decide to eliminate those nodes, then what you do is actually remove all the outgoing things from that no as well. So you end up with a much smaller, really much diminished network. And then you do back propagation training. There's one example on this much diminished network.

And then on different examples, you would toss a set of coins again and keep a different set of nodes and then dropout or eliminate different than nodes. And so for each training example, you would train it using one of these neural based networks. So, maybe it seems like a slightly crazy technique.

Implementing dropout ("Inverted dropout")



Init:

1. The keep_probability = ...
2. create a vector d3 with True/False values
3. Element wise product a3 with that d3 vector
4. we're going to take a3/0.8 or really dividing by our keep.prob parameter. So, let me explain what this final step is doing. Let's say for the sake of argument that you have 50 units or 50 neurons in the third hidden layer. So maybe a3 is 50 by one dimensional or if you- factorization maybe it's 50 by m dimensional. So, if you have a 80% chance of keeping them and 20% chance of eliminating them. This means that on average, you end up with 10 units shut off or 10 units zeroed out.

```
Inverted dropout
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multply(a3, d3)
a3 /= keep_prob
```

Ở đây:

- `d3` là *mask* gồm 0 và 1 (1: giữ, 0: tắt)
- `a3 = a3 * d3` có nghĩa là những nơ-ron bị "tắt" sẽ có giá trị bằng 0.

⚖️ Vấn đề nếu chỉ làm vậy

Giả sử ban đầu giá trị trung bình của `a3` là 1.
Khi dropout 20% nơ-ron, chỉ còn 80% hoạt động.
=> Giá trị trung bình sau dropout sẽ **giảm xuống còn 0.8** (nhỏ hơn trước).

Điều này khiến mạng **học sai lệch** vì khi huấn luyện giá trị nhỏ hơn lúc chạy thử.

✅ Cách khắc phục — Inverted Dropout

Để giữ cho **giá trị kỳ vọng (expected value)** của các nơ-ron **không đổi**, ta chia cho `keep_prob`:

```python
a3 /= keep_prob
```

Tức là:

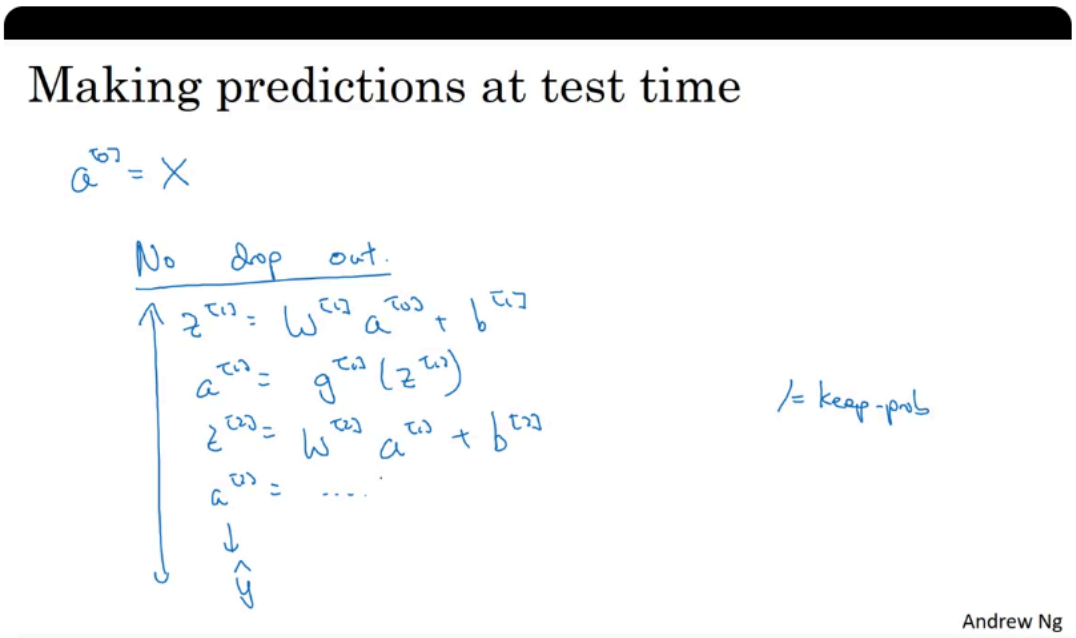
$$a3' = \frac{a3 \times d3}{keep\_prob}$$

Ví dụ:

- `keep_prob = 0.8`
- Nếu một nơ-ron được giữ lại, ta nhân nó với `1 / 0.8 = 1.25`
  → Bù lại việc 20% nơ-ron bị tắt, giúp **trung bình toàn mạng vẫn như cũ**.

🔍 Tóm tắt:

Bước	Mục đích	Giải thích
`d3 = np.random.rand(...) < keep_prob`	Tạo mask	Xác định nơ-ron nào bị tắt
`a3 = np.multiply(a3, d3)`	Áp dụng dropout	Tắt 1 phần nơ-ron
`a3 /= keep_prob`	Chuẩn hóa giá trị	Giữ giá trị kỳ vọng của mạng không đổi



But notice that the test time you're not using dropout explicitly and you're not tossing coins at random, you're not flipping coins to decide which hidden units to eliminate. 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's computationally inefficient and will give you roughly the same result; very, very similar results to this different procedure as well. And just to mention, the inverted dropout thing, you remember the step on the previous line when we divided by the cheap.prob. The effect of that was to ensure that even when you don't see men

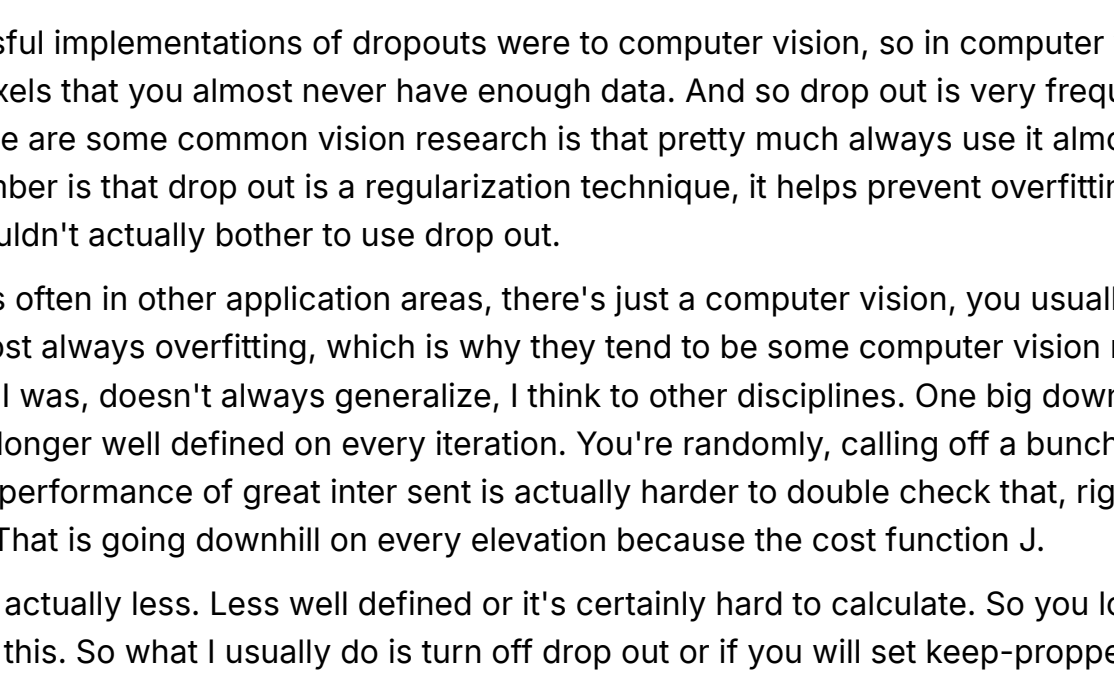
dropout at test time to the scaling, the expected value of these activations don't change. So, you don't need to add in an extra funny scaling parameter at test time. That's different than when you have that training time.

Clarification about Upcoming Understanding Dropout Video

Please note that in the next video from around 2:40 - 2:50, the dimension of $w^{[1]}$ should be 7x3 instead of 3x7, and $w^{[3]}$ should be 3x7 instead of 7x3.

In general, the number of neurons in the previous layer gives us the number of columns of the weight matrix, and the number of neurons in the current layer gives us the number of rows in the weight matrix.

Understanding Dropout



Understanding Dropout

Save note

summarize if you're more worried about some layers of fitting than others, you can set a lower key prop for some layers than others.

The downside 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 drop out and then just have one hyper parameter which is a key prop for the layers for which you do apply drop out and before we wrap up just a couple implantation all tips.

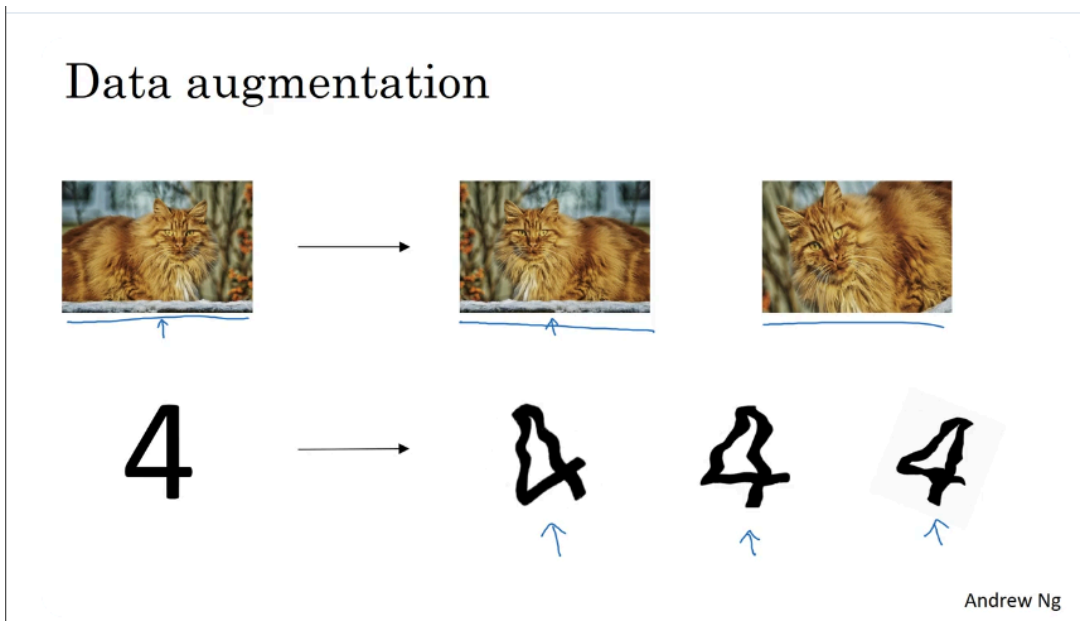
Many of the first successful implementations of dropouts were to computer vision, so in computer vision, the input sizes so big in putting all these pixels that you almost never have enough data. And so drop out is very frequently used by the computer vision and there are some common vision research is that pretty much always use it almost as a default. But really, the thing to remember is that drop out is a regularization technique, it helps prevent overfitting. And so unless my avram is overfitting, I wouldn't actually bother to use drop out.

So as you somewhat less often in other application areas, there's just a computer vision, you usually just don't have enough data so you almost always overfitting, which is why they tend to be some computer vision researchers swear by drop out by the intuition. I was, doesn't always generalize, I think to other disciplines. One big downside of drop out is that the cost function J is no longer well defined on every iteration. You're randomly, calling off a bunch of notes. And so if you are double checking the performance of great inter sent is actually harder to double check that, right? You have a well defined cost function J. That is going downhill on every elevation because the cost function J.

That you're optimizing is actually less. Less well defined or it's certainly hard to calculate. So you lose this debugging tool to have a plot a draft like this. So what I usually do is turn off drop out or if you will set keep-propped = 1 and run my code and make sure that it is monitored quickly decreasing J. And then turn on drop out and hope that, I didn't introduce, welcome to my code during drop out because you need other ways, I guess, but not plotting these figures to make sure that your code is working, the gradient descent is working even with drop out.

Other Regularization Methods

Data augmentation

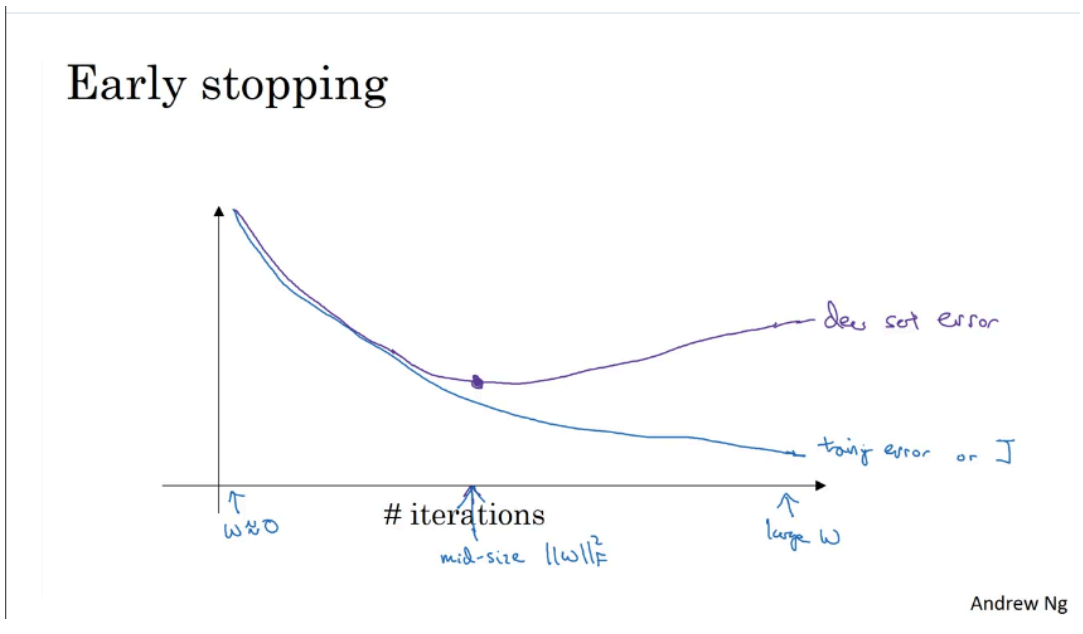


In addition to L2 regularization and drop out regularization there are few other techniques to reducing over fitting in your neural network. Let's take a look. Let's say you fitting a CAD crossfire. If you are over fitting getting more training data can help, but getting more training data can be expensive and sometimes you just can't get more data. But what you can do is augment your training set by taking image like this. And for example, flipping it horizontally and adding that also with your training set. So now instead of just this one example in your training set, you can add this to your training example.

So by flipping the images horizontally, you could double the size of your training set. Because you're training set is now a bit redundant this isn't as good as if you had collected an additional set of brand new independent examples. But you could do this Without needing to pay the expense of going out to take more pictures of cats. And then other than flipping horizontally, you can also take random crops of the image. So here we're rotated and sort of randomly zoom into the image and this still looks like a cat. So by taking random distortions and translations of the image you could augment your data set and make additional fake training examples. Again, these extra fake training examples they don't add as much information as they were to call they get a brand new independent example of a cat.

But because you can do this, almost for free, other than for some computational costs. This can be an inexpensive way to give your algorithm more data and therefore sort of regularize it and reduce over fitting. And by synthesizing examples like this what you're really telling your algorithm is that If something is a cat then flipping it horizontally is still a cat. Notice I didn't flip it vertically, because maybe we don't want upside down cats, right? And then also maybe randomly zooming in to part of the image it's probably still a cat. For optical character recognition you can also bring your data set by taking digits and imposing random rotations and distortions to it. So If you add these things to your training set, these are also still digit four.

Early Stopping

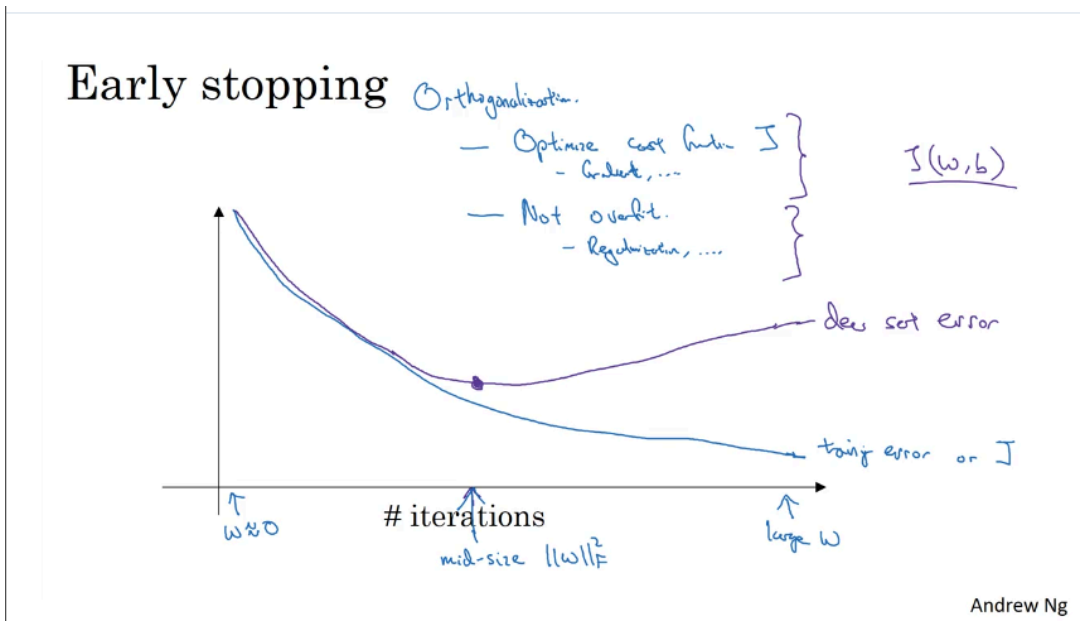


So what you're going to do is as you run gradient descent you're going to plot your, either the training error, you'll use 01 classification error on the training set. Or just plot the cost function J optimizing, and that should decrease monotonically, like so, all right?

Because as you trade, hopefully, you're trading around your cost function J should decrease. So with early stopping, what you do is you plot this, and you also plot your dev set error. And again, this could be a classification error in a development

sense, or something like the cost function, like the logistic loss or the log loss of the dev set. Now what you find is that your dev set error will usually go down for a while, and then it will increase from there. So what early stopping does is, you will say well, it looks like your neural network was doing best around that iteration, so we just want to stop trading on your neural network halfway and take whatever value achieved this dev set error. So why does this work? Well when you've haven't run many iterations for your neural network yet your parameters w will be close to zero.

Because with random initialization you probably initialize w to small random values so before you train for a long time, w is still quite small. And as you iterate, as you train, w will get bigger and bigger and bigger until here maybe you have a much larger value of the parameters w for your neural network. So what early stopping does is by stopping halfway you have only a mid-size rate w. And so similar to L2 regularization by picking a neural network with smaller norm for your parameters w, hopefully your neural network is over fitting less. And the term early stopping refers to the fact that you're just stopping the training of your neural network earlier. I sometimes use early stopping when training a neural network. But it does have one downside, let me explain.



I think of the machine learning process as comprising several different steps. One, is that you want an algorithm to optimize the cost function J and we have various tools to do that, such as grade intersect. And then we'll talk later about other algorithms, like momentum and RMS prop and Atom and so on. But after optimizing the cost function J, you also wanted to not over-fit. And we have some tools to do that such as your regularization, getting more data and so on. Now in machine learning, we already have so many hyper-parameters it surge over. It's already very complicated to choose among the space of possible algorithms.

And so I find machine learning easier to think about when you have one set of tools for optimizing the cost function J, and when you're focusing on authorizing the cost function J. All you care about is finding w and b, so that J(w,b) is as small as possible. You just don't think about anything else other than reducing this. And then it's completely separate task to not over fit, in other words, to reduce variance. And when you're doing that, you have a separate set of tools for doing it. And this principle is sometimes called **orthogonalization**. And there's this idea, that you want to be able to think about one task at a time.

Rather than using early stopping, one alternative is just use L2 regularization then you can just train the neural network as long as possible. I find that this makes the search space of hyper parameters easier to decompose, and easier to search over. But the downside of this though is that you might have to try a lot of values of the regularization parameter lambda. And so this makes searching over many values of lambda more computationally expensive. And the advantage of early stopping is that running the gradient descent process just once, you get to try out values of small w, mid-size w, and large w, without needing to try a lot of values of the L2 regularization hyperparameter lambda. If this concept doesn't completely make sense to you yet, don't worry about it. We're going to talk about orthogonalization in greater detail in a later video, I think this will make a bit more sense.